Comment développer une loi de comportement avec MFront

Jean-Michel Proix[(1)] Thomas Helfer [(2)]

[(1)]EDF R&D AMA T64

[(2)]CEA DEN Cad DEC SESC LSC

mai 2014

Titre du slide : Plan de la présentation

les équations

La loi de Norton est définie en 3D par :

$$\begin{cases} \underline{\epsilon}^{\text{to}} = \underline{\epsilon}^{\text{el}} + \underline{\epsilon}^{\text{vis}} \\ \underline{\sigma} = \underline{\underline{\mathbf{D}}} : \underline{\epsilon}^{\text{el}} \\ \underline{\dot{\epsilon}}^{\text{vis}} = \dot{p}\,\underline{n} \\ \dot{p} = A\,\sigma_{\text{eq}}^{m} \end{cases}$$

- $\underline{\epsilon}^{\text{to}}$, $\underline{\epsilon}^{\text{el}}$, $\underline{\epsilon}^{\text{vis}}$ tenseurs déf. totale, élastique et visqueuse ;
- $\underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{\text{eq}}}$ est le tenseur direction d'écoulement ;
- $\underline{s}$ est le tenseur déviateur des contraintes ;
- σ_{eq} est la norme de VON MISES.

$\underline{\underline{\mathbf{D}}}$ est déduit du module d'YOUNG E et du coef. de POISSON ν

discrétisation implicite

Discrétisation en temps : ici, implicite

- Les quantités sont écrites à l'instant t_i
- les dérivées en temps sont remplacées par leurs incréments sur l'intervalle $\Delta t = t_i - t_{i-1}$

Pour la loi de Norton, on obtient : $\begin{cases} \Delta \underline{\epsilon}^{\mathrm{el}} - \Delta \underline{\epsilon}^{\mathrm{to}} + \Delta p \, \underline{n} = 0 \\ \Delta p - \Delta t \, A \, \sigma_{\mathrm{eq}}^{m} = 0 \end{cases}$

avec :

- $\underline{n} = \dfrac{3}{2} \dfrac{\underline{s}\left(t_i\right)}{\sigma_{\mathrm{eq}}\left(t_i\right)}$.

Système de 7 équations à 7 inconnues : $\Delta \underline{\epsilon}^{\mathrm{el}}, \Delta p$

premier développement avec mfront

```
@Parser Implicit;
@Behaviour Norton;
@Algorithm NewtonRaphson_NumericalJacobian;
@RequireStiffnessTensor;
@MaterialProperty real A;
@MaterialProperty real m;
@StateVariable real    p;
@ComputeStress{  sig = D*eel;}
@Integrator{
  real seq = sigmaeq(sig);
  Stensor n = Stensor(0.);
  if(seq > 1.e-15){
    n   = 1.5*deviator(sig)/seq;
  }
  feel = deel + dp*n-deto;
  fp   = dp - dt*A*pow(seq,m);
}
```

Newton, calcul de J

$$\sigma = \underline{\underline{\mathbf{D}}} : \underline{\epsilon}^{\mathrm{el}}$$

$$\underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{\mathrm{eq}}}$$

$$\Delta\,\underline{\epsilon}^{\mathrm{el}} + \Delta\, p\,\underline{n} - \Delta\,\underline{\epsilon}^{\mathrm{to}} = 0$$
$$\Delta\, p - \Delta\, t\, A\,\sigma_{\mathrm{eq}}^{m} = 0$$

mtest

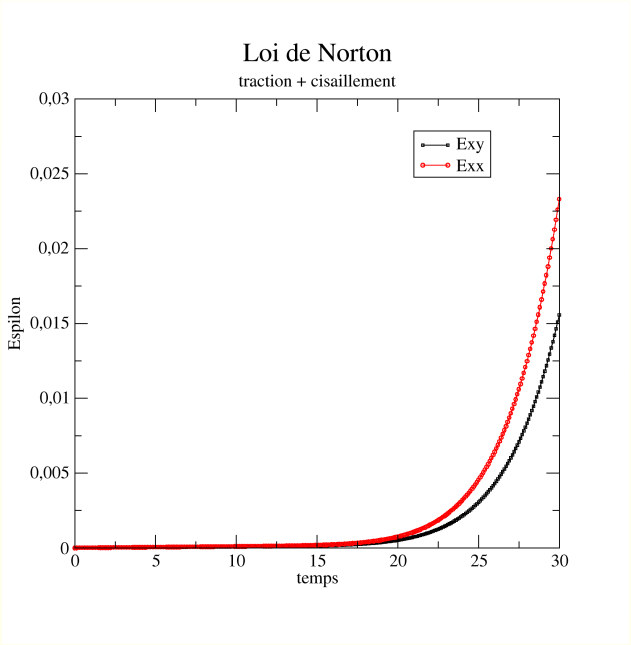
compilation...

```
mfront -obuild -interface=aster norton.mfront
```

test...

```
mtest norton.mtest
```

```
@Behaviour<aster> './src/
    libAsterBehaviour.so' 'asternorton';
@MaterialProperty<constant> 'YoungModulus
    '     2.E11 ;
@MaterialProperty<constant> 'PoissonRatio
    '      0.3 ;
@MaterialProperty<constant> 'A'    8.e-67;
@MaterialProperty<constant> 'm'    8.2;
@ExternalStateVariable 'Temperature'
    293.15;
@StiffnessMatrixType 'Elastic' ;
@ImposedStress 'SXX' {0.:0.,30.:40.e6};
@ImposedStress 'SXY' {0.:0.,30.:40.e6};
@Times {0.,  30. in 300};
```



utilisation avec Code_Aster—1/2

compilation du comportement ;

- soit avant le calcul Code_Aster : `mfront -obuild -interface=aster norton.mfront => src/libAsterBehaviour.so`
- dans le fichier de commandes :
  - `import os;`
  - `os.system("mfront -obuild -interface=aster norton.mfront".`

dans DEFI_MATERIAU sous UMAT :

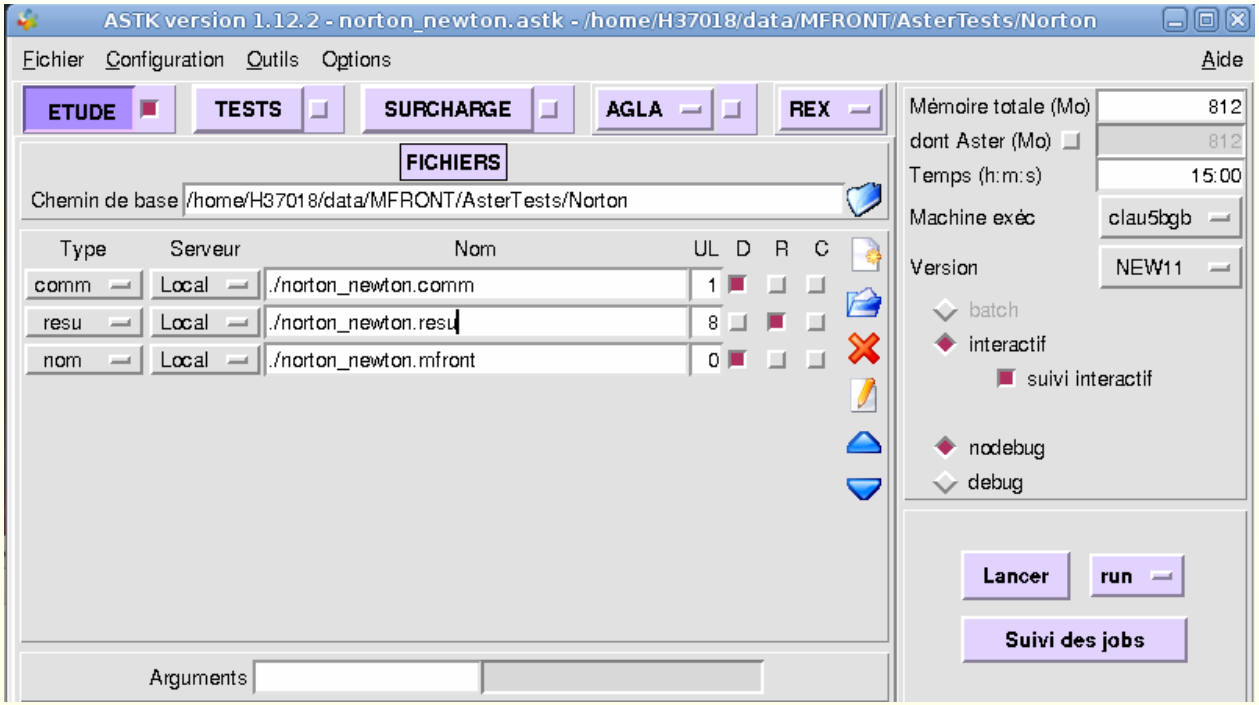
```
C1=178600.0E6;
C2=0.3;
C3=8...;
```

... ; comme définis dans le fichier norton.mfront.

utilisation avec Code_Aster—2/2

dans STAT_NON_LINE sous COMPORTEMENT ;

- RELATION='MFRONT'
- LIBRAIRIE='libAsterBehaviour.so'
- NOM_ROUTINE='asternorton.so'
- NB_VARI=7



fonctionnalités de mfront

avantages de mfront

mfront a été développé par le CEA (pleiades) pour :

- **permettre** l'écriture de connaissances matériau :
  - les propriétés matériau ; voir ▸ matériaux
  - les comportements mécaniques (plasticité,endommagement) ;
- **mutualiser** ces connaissances matériau :
  - entre différentes applications de la plate-forme pleiades,
  - maintenant interfacé avec Code_Aster
  - la base de données siriusutilise des fichiers mfront en interne (+ de 100 matériaux différents) ;
- **simplifier** le travail des utilisateurs :
  - numérique : on écrit les équations, pas les algos ;
  - informatique peu de lignes à écrire ;
  - donc minimiser le risque d'erreur
- mfront produit un code efficace :
  - opérations tensorielles optimisées ;
  - benchmarks avec Code_Aster de 18 lois depuis 12 mois

avantages de mfront

mfront a été développé par le CEA (pleiades) pour :

- **permettre** l'écriture de connaissances matériau :
  - les propriétés matériau ; voir ▸ matériaux
  - les comportements mécaniques (plasticité,endommagement) ;
- **mutualiser** ces connaissances matériau :
  - entre différentes applications de la plate-forme pleiades,
  - maintenant interfacé avec Code_Aster
  - la base de données siriusutilise des fichiers mfront en interne (+ de 100 matériaux différents) ;
- **simplifier** le travail des utilisateurs :
  - numérique : on écrit les équations, pas les algos ;
  - informatique peu de lignes à écrire ;
  - donc minimiser le risque d'erreur
- mfront produit un code efficace :
  - opérations tensorielles optimisées ;
  - benchmarks avec Code_Aster de 18 lois depuis 12 mois

avantages de mfront

mfront a été développé par le CEA (pleiades) pour :

- **permettre** l'écriture de connaissances matériau :
  - les propriétés matériau ; voir ▸ matériaux
  - les comportements mécaniques (plasticité,endommagement) ;
- **mutualiser** ces connaissances matériau :
  - entre différentes applications de la plate-forme pleiades,
  - maintenant interfacé avec Code_Aster
  - la base de données siriusutilise des fichiers mfront en interne (+ de 100 matériaux différents) ;
- **simplifier** le travail des utilisateurs :
  - numérique : on écrit les équations, pas les algos ;
  - informatique peu de lignes à écrire ;
  - donc minimiser le risque d'erreur
- mfront produit un code efficace :
  - opérations tensorielles optimisées ;
  - benchmarks avec Code_Aster de 18 lois depuis 12 mois

avantages de mfront

mfront a été développé par le CEA (pleiades) pour :

- **permettre** l'écriture de connaissances matériau :
  - les propriétés matériau ; voir ▸ matériaux
  - les comportements mécaniques (plasticité,endommagement) ;
- **mutualiser** ces connaissances matériau :
  - entre différentes applications de la plate-forme pleiades,
  - maintenant interfacé avec Code_Aster
  - la base de données siriusutilise des fichiers mfront en interne (+ de 100 matériaux différents) ;
- **simplifier** le travail des utilisateurs :
  - numérique : on écrit les équations, pas les algos ;
  - informatique peu de lignes à écrire ;
  - donc minimiser le risque d'erreur
- mfront produit un code efficace :
  - opérations tensorielles optimisées ;
  - benchmarks avec Code_Aster de 18 lois depuis 12 mois

algorithmes disponibles dans mfront

Connaissant en un point et à un instant t le tenseur $\varepsilon_{t+\Delta t}$, et toutes les quantités à l'instant t, mfront a pour fonction de calculer :

- les contraintes $\underline{\sigma}_{t+\Delta t}$ et les variables internes $\alpha_{t+\Delta t}$;
- l'opérateur tangent cohérent : $\dfrac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\epsilon}^{\text{to}}}$.
- pour plus de détail, voir algo global

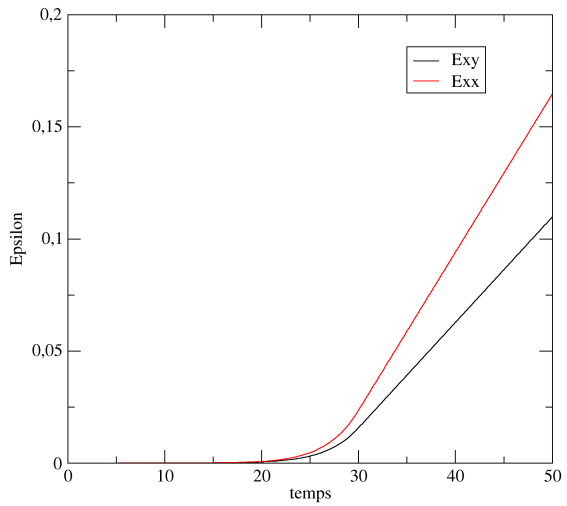
en intégrant le système d'équations régissant la loi de comportement locale à l'aide de divers algorithmes :

- spécifiques (élasto-(visco)-plasticité incompressible) ;
- explicites (méthodes de Runge-Kutta) ;
- implicites (méthode de Newton-Raphson et variantes) ;
- libre (l'utilisateur définit l'intégration).

quel algorithme choisir ?

- si un intégrateur spécifique existe, l'utiliser :
  - réduction du nombre d'équations et méthode implicite ;
- si l'on doit recourir à un autre intégrateur, **préférer l'intégration implicite** :
  - les temps de calculs sont souvent **très** avantageux ;
  - on a (plus facilement) la **tangente cohérente** ;
- utiliser une méthode de RUNGE-KUTTA si :
  - **rien d'autre** n'est possible (grand nombre de variables) ;
- exemple sur le petit test de Norton :

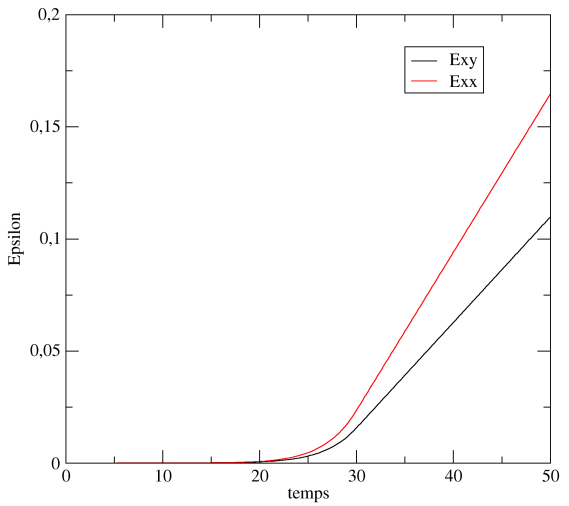
traction	spec	impl	rk
30 MPa	0.032s	0.128s	0.372s
50 MPa	0.218s	0.228s	10.37s



quel algorithme choisir ?

- si un intégrateur spécifique existe, l'utiliser :
  - réduction du nombre d'équations et méthode implicite ;
- si l'on doit recourir à un autre intégrateur, **préférer l'intégration implicite** :
  - les temps de calculs sont souvent **très** avantageux ;
  - on a (plus facilement) la **tangente cohérente** ;
- utiliser une méthode de RUNGE-KUTTA si :
  - **rien d'autre** n'est possible (grand nombre de variables) ;
- exemple sur le petit test de Norton :

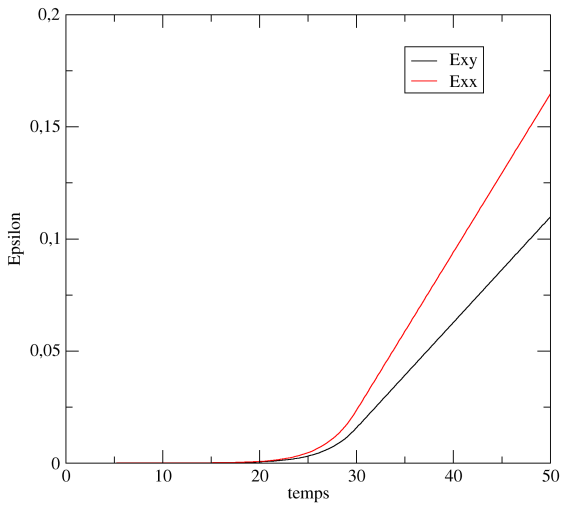
traction	spec	impl	rk
30 MPa	0.032s	0.128s	0.372s
50 MPa	0.218s	0.228s	10.37s



quel algorithme choisir ?

- si un intégrateur spécifique existe, l'utiliser :
  - réduction du nombre d'équations et méthode implicite ;
- si l'on doit recourir à un autre intégrateur, **préférer l'intégration implicite** :
  - les temps de calculs sont souvent **très** avantageux ;
  - on a (plus facilement) la **tangente cohérente** ;
- utiliser une méthode de RUNGE-KUTTA si :
  - **rien d'autre** n'est possible (grand nombre de variables) ;
- exemple sur le petit test de Norton :

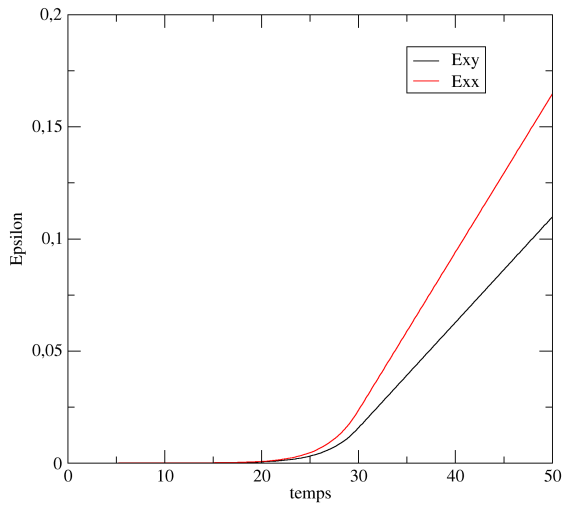
traction	spec	impl	rk
30 MPa	0.032s	0.128s	0.372s
50 MPa	0.218s	0.228s	10.37s



quel algorithme choisir ?

- si un intégrateur spécifique existe, l'utiliser :
  - réduction du nombre d'équations et méthode implicite ;
- si l'on doit recourir à un autre intégrateur, **préférer l'intégration implicite** :
  - les temps de calculs sont souvent **très** avantageux ;
  - on a (plus facilement) la **tangente cohérente** ;
- utiliser une méthode de RUNGE-KUTTA si :
  - **rien d'autre** n'est possible (grand nombre de variables) ;
- exemple sur le petit test de Norton :

traction	spec	impl	rk
30 MPa	0.032s	0.128s	0.372s
50 MPa	0.218s	0.228s	10.37s



algorithmes spécifiques

- quatre intégrateurs spécifiques :
  - `IsotropicMisesCreep`, écoulement viscoplastique isotrope $dp = f\left(\sigma_{\text{eq}}\right)$;
  - `IsotropicStrainHardeningMisesCreep`, écoulement viscoplastique isotrope avec écrouissage $dp = f\left(\sigma_{\text{eq}}, p\right)$;
  - `IsotropicPlasticMisesFlow`, écoulement plastique isotrope $f\left(\sigma_{\text{eq}}, p\right) <= 0$;
  - `MultipleIsotropicMisesFlows`, une somme des différents écoulements précédents ;
- l'élasticité est élastique isotrope :
  - les coefficients d'élasticité sont donnés par le code aux éléments finis ;
- il suffit de donner la (ou les) fonction(s) f et ses dérivées ;
- algorithme optimisé, réduction à une équation scalaire.

algorithmes spécifiques

- quatre intégrateurs spécifiques :
  - `IsotropicMisesCreep`, écoulement viscoplastique isotrope $dp = f\left(\sigma_{\text{eq}}\right)$;
  - `IsotropicStrainHardeningMisesCreep`, écoulement viscoplastique isotrope avec écrouissage $dp = f\left(\sigma_{\text{eq}}, p\right)$;
  - `IsotropicPlasticMisesFlow`, écoulement plastique isotrope $f\left(\sigma_{\text{eq}}, p\right) <= 0$;
  - `MultipleIsotropicMisesFlows`, une somme des différents écoulements précédents ;
- l'élasticité est élastique isotrope :
  - les coefficients d'élasticité sont donnés par le code aux éléments finis ;
- il suffit de donner la (ou les) fonction(s) f et ses dérivées ;
- algorithme optimisé, réduction à une équation scalaire.

algorithmes spécifiques

- quatre intégrateurs spécifiques :
  - `IsotropicMisesCreep`, écoulement viscoplastique isotrope $dp = f\left(\sigma_{\text{eq}}\right)$;
  - `IsotropicStrainHardeningMisesCreep`, écoulement viscoplastique isotrope avec écrouissage $dp = f\left(\sigma_{\text{eq}}, p\right)$;
  - `IsotropicPlasticMisesFlow`, écoulement plastique isotrope $f\left(\sigma_{\text{eq}}, p\right) <= 0$;
  - `MultipleIsotropicMisesFlows`, une somme des différents écoulements précédents ;
- l'élasticité est élastique isotrope :
  - les coefficients d'élasticité sont donnés par le code aux éléments finis ;
- il suffit de donner la (ou les) fonction(s) f et ses dérivées ;
- algorithme optimisé, réduction à une équation scalaire.

algorithmes spécifiques

- quatre intégrateurs spécifiques :
  - `IsotropicMisesCreep`, écoulement viscoplastique isotrope $dp = f\left(\sigma_{\text{eq}}\right)$;
  - `IsotropicStrainHardeningMisesCreep`, écoulement viscoplastique isotrope avec écrouissage $dp = f\left(\sigma_{\text{eq}}, p\right)$;
  - `IsotropicPlasticMisesFlow`, écoulement plastique isotrope $f\left(\sigma_{\text{eq}}, p\right) <= 0$;
  - `MultipleIsotropicMisesFlows`, une somme des différents écoulements précédents ;
- l'élasticité est élastique isotrope :
  - les coefficients d'élasticité sont donnés par le code aux éléments finis ;
- il suffit de donner la (ou les) fonction(s) f et ses dérivées ;
- algorithme optimisé, réduction à une équation scalaire.

exemple d'algorithme spécifique

```
@Parser IsotropicMisesCreep ;
@Behaviour Norton;

@MaterialProperty stress A;
@MaterialProperty stress m;

@FlowRule{
  real tmp=A*pow(seq,m-1.);
  df_dseq = m*tmp ;
  f       = seq*tmp;
}
```

Intégration par une méthode explicite (RUNGE-KUTTA)

- la loi de comportement est réduite à un système différentiel :
  $\dot{Y} = G\left(Y, t\right)$ avec : $\left[\Delta\, Y\right]^T = \left[\Delta\underline{\epsilon}^{\mathrm{el}}\ ,\ \Delta\alpha\right]$
  où t représente symboliquement l'évolution des variables externes et de la déformation totale ;
- le système différentiel s'écrit dans un bloc `@Derivative` ;
- pour toute variable interne ou externe`X`, `dX` représente la vitesse dans `@ComputeStress` et `@Derivative`
  - ce n'est pas l'incrément !
- le code du bloc `@UpdateAuxiliaryStateVariables` peut être appelé plusieurs fois. Il faut utiliser la variable locale `dt_` pour connaître le pas de temps effectivement utilisé (`dt` désigne toujours le pas de temps total)

Intégration par une méthode explicite (RUNGE-KUTTA)

- la loi de comportement est réduite à un système différentiel :
  $\dot{Y} = G\left(Y, t\right)$ avec : $\left[\Delta\, Y\right]^T = \left[\Delta\underline{\epsilon}^{\mathrm{el}}\ ,\ \Delta\alpha\right]$
  où t représente symboliquement l'évolution des variables externes et de la déformation totale ;
- le système différentiel s'écrit dans un bloc `@Derivative` ;
- pour toute variable interne ou externe`X`, `dX` représente la vitesse dans `@ComputeStress` et `@Derivative`
  - ce n'est pas l'incrément !
- le code du bloc `@UpdateAuxiliaryStateVariables` peut être appelé plusieurs fois. Il faut utiliser la variable locale `dt_` pour connaître le pas de temps effectivement utilisé (`dt` désigne toujours le pas de temps total)

Intégration par une méthode explicite (RUNGE-KUTTA)

- la loi de comportement est réduite à un système différentiel :
  $\dot{Y} = G\left(Y, t\right)$ avec : $\left[\Delta\, Y\right]^T = \left[\Delta\underline{\epsilon}^{\mathrm{el}}\ ,\ \Delta\alpha\right]$
  où t représente symboliquement l'évolution des variables externes et de la déformation totale ;
- le système différentiel s'écrit dans un bloc `@Derivative` ;
- pour toute variable interne ou externe`X`, `dX` représente la vitesse dans `@ComputeStress` et `@Derivative`
  - ce n'est pas l'incrément !
- le code du bloc `@UpdateAuxiliaryStateVariables` peut être appelé plusieurs fois. Il faut utiliser la variable locale `dt_` pour connaître le pas de temps effectivement utilisé (`dt` désigne toujours le pas de temps total)

Intégration par une méthode explicite (RUNGE-KUTTA)

- la loi de comportement est réduite à un système différentiel :
  $\dot{Y} = G\left(Y, t\right)$ avec : $\left[\Delta\, Y\right]^T = \left[\Delta\underline{\epsilon}^{\mathrm{el}}\ ,\ \Delta\alpha\right]$
  où t représente symboliquement l'évolution des variables externes et de la déformation totale ;
- le système différentiel s'écrit dans un bloc `@Derivative` ;
- pour toute variable interne ou externe`X`, `dX` représente la vitesse dans `@ComputeStress` et `@Derivative`
  - ce n'est pas l'incrément !
- le code du bloc `@UpdateAuxiliaryStateVariables` peut être appelé plusieurs fois. Il faut utiliser la variable locale `dt_` pour connaître le pas de temps effectivement utilisé (`dt` désigne toujours le pas de temps total)

Loi de Norton avec Runge_Kutta

```
@Parser RungeKutta;
@Algorithm rk54;
@Behaviour Norton;
@RequireStiffnessTensor;
@MaterialProperty real A;
@MaterialProperty real m;
@StateVariable real     p;
@ComputeStress{ sig = D*eel;}
@TangentOperator{  Dt=D;}
@Derivative{
  real sigeq = sigmaeq(sig);
  Stensor  n(0.);
  if(sigeq > 1.e-15){
    n     = 1.5*deviator(sig)/sigeq;
  }
  dp    = A*pow(sigeq,m);
  deel = deto - dp*n;
```

Runge-Kutta
ordres 4 et 5

$\underline{\sigma} = \underline{\underline{\mathbf{D}}} : \underline{\epsilon}^{\mathrm{el}}$
eel défini par défaut

$$\underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{\mathrm{eq}}}$$

$$d\,p = A\,\sigma_{\mathrm{eq}}^{m}$$
$$d\,\underline{\epsilon}^{\mathrm{el}} = d\,\underline{\epsilon}^{\mathrm{to}} - d\,p\,\underline{n}$$

quelques notations et définitions

- propriété matériau `@MaterialProperty` :
  - fournie par le code appelant !
- variable local `@LocalVariable` :
  - calcul de certains termes avant l'intégration (exemple de termes d'ARRHENIUS) ;
- variable interne `@StateVariable` ;
- variable auxiliaire `@AuxiliaryStateVariable` : permet de réduire la taille des systèmes à intégrer ;
- variable externe `@ExternalStateVariable` ;
- notations ;
  - mots réservés : `eel, eto, sig,` ;
  - explicite : pour toute variable `a`, `da` est la vitesse ;
  - implicite : `da` est l'incrément, `fa` est l'équation, `dfa_dda` la dérivée ;

Intégration implicite

- Deux analyseurs :
  - `Implicit` qui déclare automatiquement la déformation élastique ;
  - `ImplicitII` qui ne déclare pas automatiquement la déformation élastique ;
- différents algorithmes :
  - `NewtonRaphson` (jacobienne calculée par l'utilisateur) ;
  - `NewtonRaphson_NumericalJacobian` (jacobienne calculée par différence finie centrée) ;
  - `Broyden` (jacobienne partielle) ;
  - `Powell DogLeg` (méthode de Powell combinant Gauss et Newton) ;

Intégration implicite : principe

- le système différentiel devient un système non-linéaire : $[F(\Delta Y) = \Delta Y - \Delta t\, G(Y_t + \theta \Delta Y, t + \theta\, \Delta t) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\text{el}}\ ,\ \Delta\alpha\right]$
- pour les lois indépendantes du temps, on annule directement la surface de charge !
- on résout ce système par un NEWTON-RAPHSON
  - il faut la jacobienne $J = \dfrac{\partial F}{\partial \Delta Y}$
- la jacobienne peut être calculée par blocs :

$$J = \frac{\partial F}{\partial Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial y_1} & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \frac{\partial f_{y_N}}{\partial y_N} \end{pmatrix}$$

- on peut demander une vérification numérique !

```
@CompareToNumericalJacobian true;
```

Intégration implicite : principe

- le système différentiel devient un système non-linéaire : $[F(\Delta Y) = \Delta Y - \Delta t\, G(Y_t + \theta \Delta Y, t + \theta\, \Delta t) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\text{el}}\ ,\ \Delta\alpha\right]$
- pour les lois indépendantes du temps, on annule directement la surface de charge !
- on résout ce système par un NEWTON-RAPHSON
  - il faut la jacobienne $J = \dfrac{\partial F}{\partial \Delta Y}$
- la jacobienne peut être calculée par blocs :

$$J = \frac{\partial F}{\partial Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial y_1} & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \frac{\partial f_{y_N}}{\partial y_N} \end{pmatrix}$$

- on peut demander une vérification numérique !

```
@CompareToNumericalJacobian true;
```

Intégration implicite : principe

- le système différentiel devient un système non-linéaire : $[F(\Delta Y) = \Delta Y - \Delta t\, G(Y_t + \theta \Delta Y, t + \theta\, \Delta t) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\text{el}}\ ,\ \Delta\alpha\right]$
- pour les lois indépendantes du temps, on annule directement la surface de charge !
- on résout ce système par un NEWTON-RAPHSON
  - il faut la jacobienne $J = \dfrac{\partial F}{\partial \Delta Y}$
- la jacobienne peut être calculée par blocs :

$$J = \frac{\partial F}{\partial Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial y_1} & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \frac{\partial f_{y_N}}{\partial y_N} \end{pmatrix}$$

- on peut demander une vérification numérique !

```
@CompareToNumericalJacobian true;
```

Intégration implicite : principe

- le système différentiel devient un système non-linéaire : $[F(\Delta Y) = \Delta Y - \Delta t\, G(Y_t + \theta \Delta Y, t + \theta\, \Delta t) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\text{el}}\ ,\ \Delta\alpha\right]$
- pour les lois indépendantes du temps, on annule directement la surface de charge !
- on résout ce système par un NEWTON-RAPHSON
  - il faut la jacobienne $J = \dfrac{\partial F}{\partial \Delta Y}$
- la jacobienne peut être calculée par blocs :

$$J = \frac{\partial F}{\partial Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial y_1} & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \frac{\partial f_{y_N}}{\partial y_N} \end{pmatrix}$$

- on peut demander une vérification numérique !

@CompareToNumericalJacobian true;

Intégration implicite : principe

- le système différentiel devient un système non-linéaire : $[F(\Delta Y) = \Delta Y - \Delta t\, G(Y_t + \theta \Delta Y, t + \theta\, \Delta t) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\text{el}}\ ,\ \Delta\alpha\right]$
- pour les lois indépendantes du temps, on annule directement la surface de charge !
- on résout ce système par un NEWTON-RAPHSON
  - il faut la jacobienne $J = \dfrac{\partial F}{\partial \Delta Y}$
- la jacobienne peut être calculée par blocs :

$$J = \frac{\partial F}{\partial Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial y_1} & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \frac{\partial f_{y_N}}{\partial y_N} \end{pmatrix}$$

- on peut demander une vérification numérique !

```
@CompareToNumericalJacobian true;
```

Exemple : loi de Norton avec intégration implicite

Le système à résoudre étant toujours défini par :

- `feel`=$\Delta \underline{\epsilon}^{\mathrm{el}} + \Delta p \, \underline{n} - \Delta \underline{\epsilon}^{\mathrm{to}}$
- `fp`=$\Delta p - \Delta t \, A \, \sigma_{\mathrm{eq}}^{m}$

Listons les différentes dérivées à calculer :

- $dfeel\_deel = \dfrac{\partial}{\partial \Delta \underline{\epsilon}^{\mathrm{el}}} \left(\Delta \underline{\epsilon}^{\mathrm{el}} + \Delta p \, \underline{n} - \Delta \varepsilon \right) = \tilde{I} + \Delta p \dfrac{\partial \underline{n}}{\partial \Delta \underline{\epsilon}^{\mathrm{el}}}$

  avec $\tilde{I}_{ijkl} = \dfrac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$
- $dfeel\_ddp = \underline{n}$
- $dfp\_ddp = 1$
- $dfp\_ddeel = -\Delta t \, A \, m \, \sigma_{eq}^{m-1} \dfrac{\partial (\sigma)_{\mathrm{eq}}}{\partial \Delta \underline{\epsilon}^{\mathrm{el}}} = -\Delta t \, A \, m \, \sigma_{eq}^{m-1} \, \theta \, \underline{n} | D$

Les dérivées de la normale $\underline{n}$ sont : $\dfrac{\partial \underline{n}}{\partial \Delta \underline{\epsilon}^{\mathrm{el}}} = \dfrac{2\mu}{(\sigma)_{\mathrm{eq}}} (M - \underline{n} \otimes \underline{n})$

Le tenseur $\underline{\underline{\mathbf{M}}}$ étant défini par : $\underline{\underline{\mathbf{M}}} = \dfrac{3}{2}\underline{\underline{\mathbf{I}}} - \dfrac{1}{2}\underline{I} \otimes \underline{I}$

Exemple : loi de Norton avec intégration implicite —1/2

La première partie de `norton.mfront` est inchangée. On a oté : Algorithm NewtonRaphson_NumericalJacobian ;

```
@Parser Implicit;
@Behaviour Norton;
@RequireStiffnessTensor;
@Epsilon 1.e-12;
@MaterialProperty stress young;
@MaterialProperty real nu;
@MaterialProperty real A;
@MaterialProperty real m;
@StateVariable real     p;
@TangentOperator{
  Stensor4 Je;
  getPartialJacobianInvert(Je);
  Dt = D*Je;
}
@ComputeStress{   sig = D*eel;}
```

Exemple : loi de Norton avec intégration implicite —2/2

La matrice jacobienne J est programmée :

```
@Integrator{
  real seq = sigmaeq(sig);
  real inv_seq=0. ;
  if(seq > 1.e-15){
    inv_seq = 1./seq ;
  }
  Stensor n = 1.5*deviator(sig)*inv_seq ;
  feel += dp*n-deto;
  fp   -= dt*A*pow(seq,m);

  // jacobienne
  const Stensor4 Jmn  = Stensor4::M() - (n^n);
  const real mu = young/2/(1.+nu) ;
  dfeel_ddeel += 2.*mu*theta*dp*Jmn*inv_seq ;
  dfeel_ddp    = n;
  dfp_ddeel    = - dt*A*m*pow(seq,m-1)*theta*(n| D);
  dfp_ddp      = 1 ;
}
```

opérateurs tangents

- L'intégration de la loi de comportement est effectuée soit en explicite, soit en implicite ;
- On obtient les contraintes $\underline{\sigma}_{t+\Delta t}$ et les variables internes $\alpha_{t+\Delta t}$;
- Mais pour qu'un calcul de structure converge vite, il faut calculer l'opérateur tangent cohérent :
$$\frac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}}$$

l'opérateur tangent en mfront

Dans le cas d'une intégration explicite, on utilise l'opérateur d'élasticité D

```
@TangentOperator{    Dt=D;}
```

Dans le cas implicite, on peut souvent construire facilement l'opérateur tangent cohérent :

```
@TangentOperator{
  Stensor4 Je;
  getPartialJacobianInvert(Je);
  Dt = D*Je;
}
```

Il est extrait de l'inverse de la jacobienne J
soit programmée, soit estimée numériquement par mfront(pour plus de détail, voir ▸ KtgtFromJ).

fonctionnalités de mtest

- pour simuler la réponse d'un point matériel (comme `SIMU_POINT_MAT`) ;
- piloter en contraintes ou/en déformations ou de manière mixte ;
- l'algorithme de résolution peut être paramétré :
  - matrice de prédiction, matrice tangente cohérente (interface `Code_Aster`) ;
  - sous-découpage du pas de temps ;
  - etc...
- possibilité de comparer les résultats à une solution analytique ou des fichiers de références (non régression) ;
- les lois `mfront` peuvent générer des fichiers `mtest` en cas de non convergence d'un calcul de structure

exemple de fichier mtest

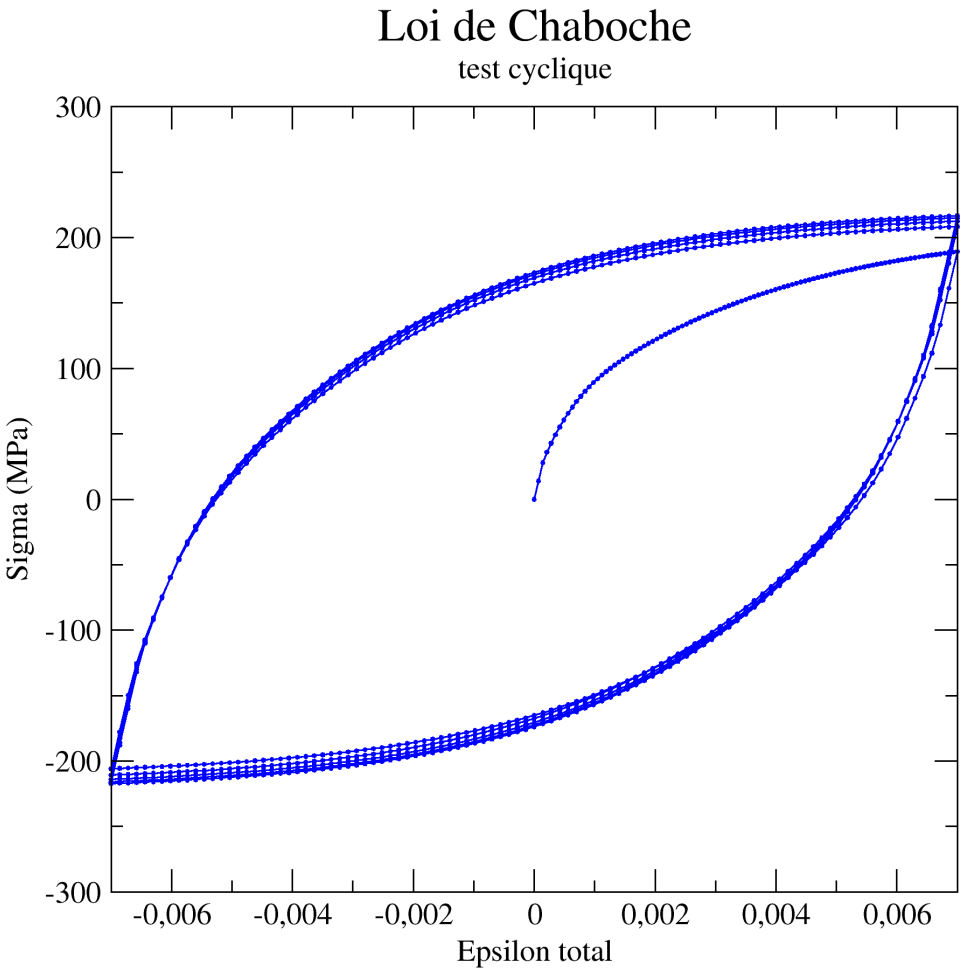
```
@Behaviour<aster> 'src/libAsterBehaviour.so' '
    asterburger';
@MaterialProperty<constant> 'young' 31000. ;
@MaterialProperty<constant> 'nu'     0.3 ;
@MaterialProperty<constant> 'KRS'    2.0E5 ;
@MaterialProperty<constant> 'NRS'    4.0E10 ;
```

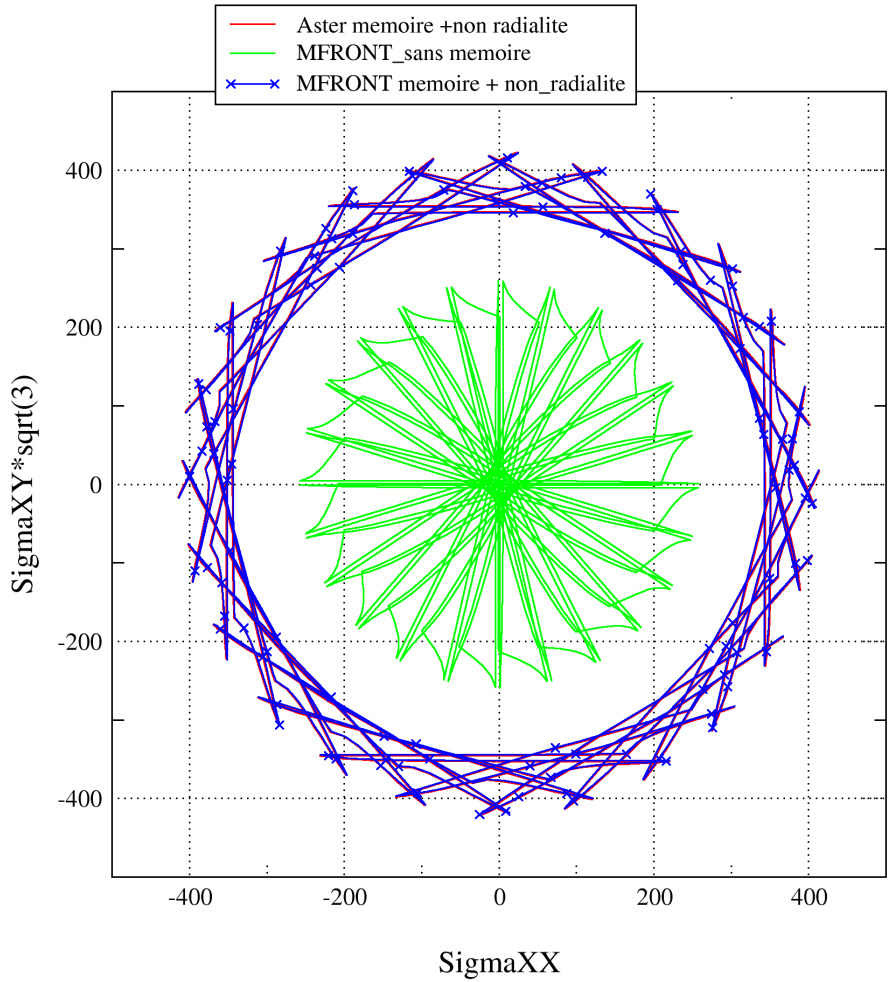
...

```
@ExternalStateVariable 'Temperature' 293.15;
@ExternalStateVariable 'C' 1.;
@ImposedStress 'SZZ' {0.: 0., 1.: -1., 31536010. : -1.};
@Times {0.,1. in 1, 138629.44 in 10,8640000. in 10 };
@Test<file> 'burger.ref' {'EZZ':4} 1.e-8 ;
```

Utilisation : `mtest fichier.mtest`

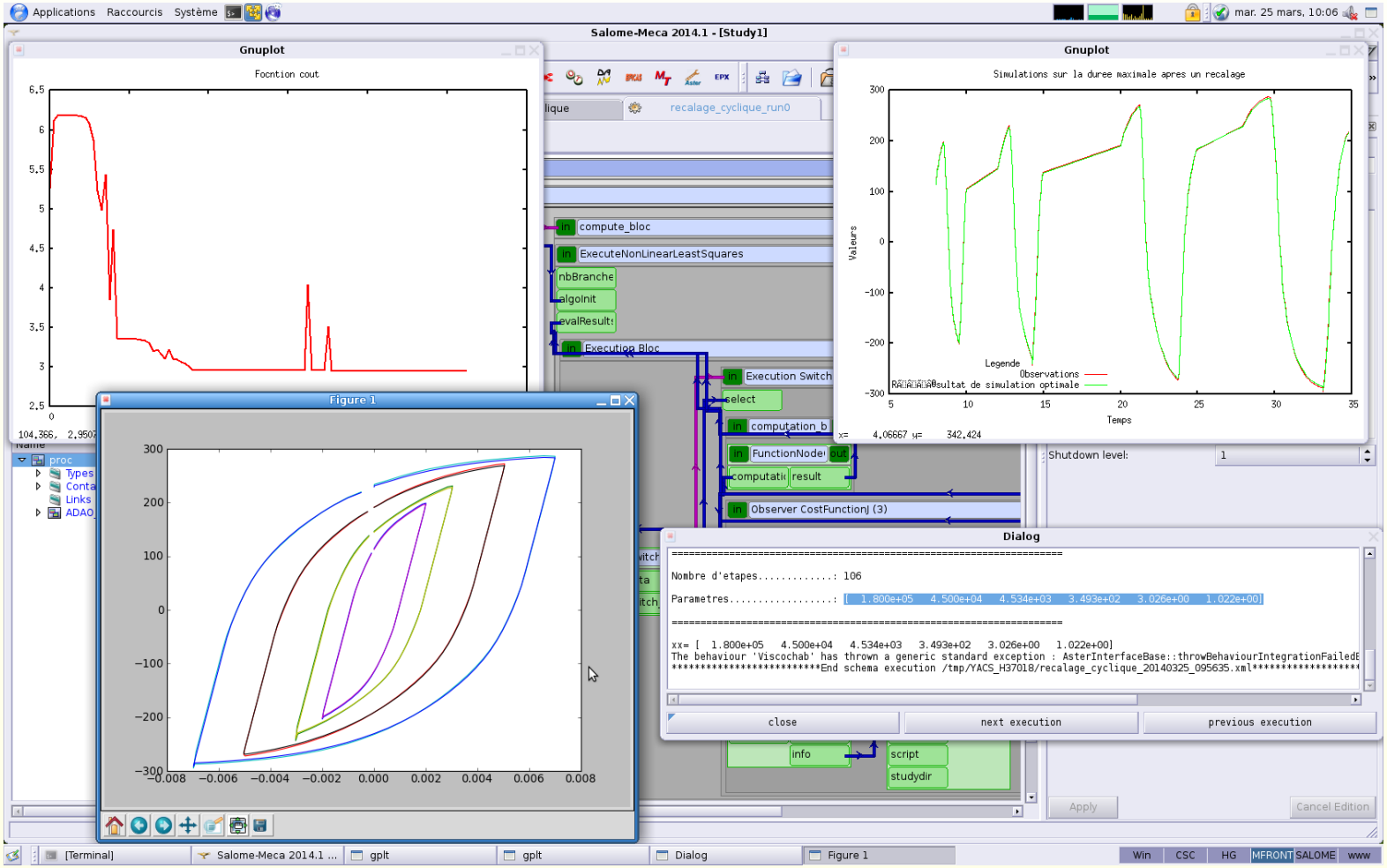
exemples de simulations mtest





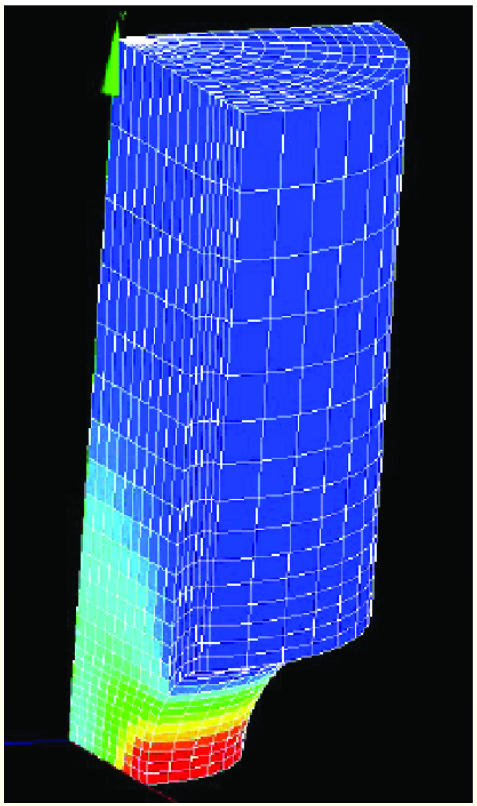
adao

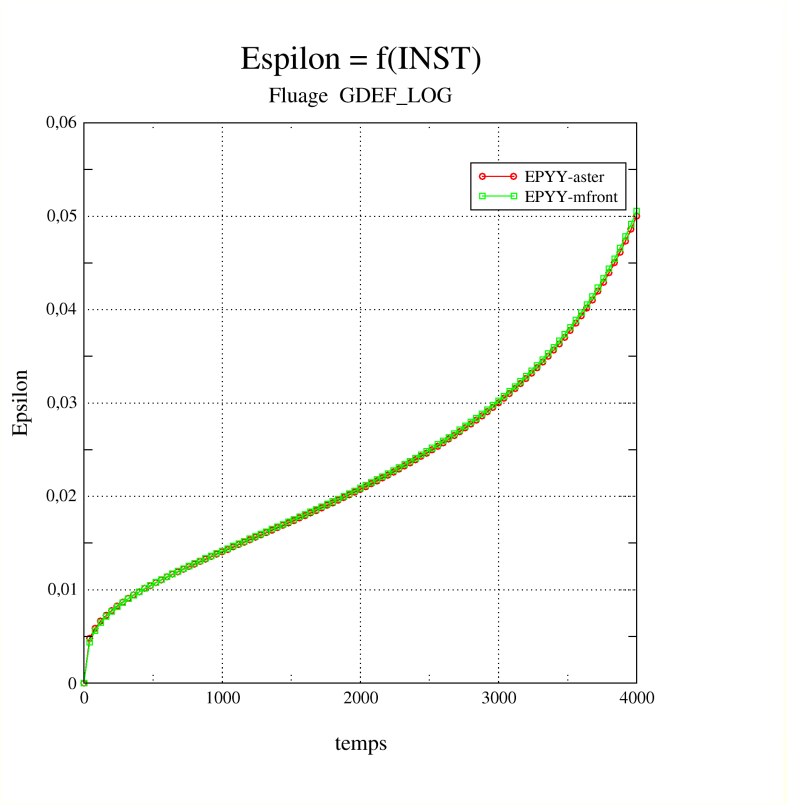
- recalage des propriétés matériau ;
- utilise `Code_Aster` + `mfront` ou directement `mtest` ;



(pour plus de détail, voir adao).

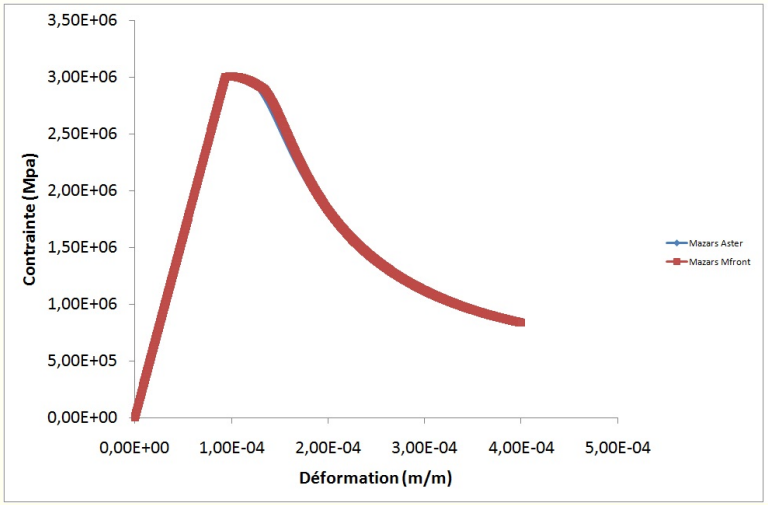
Hayhurst

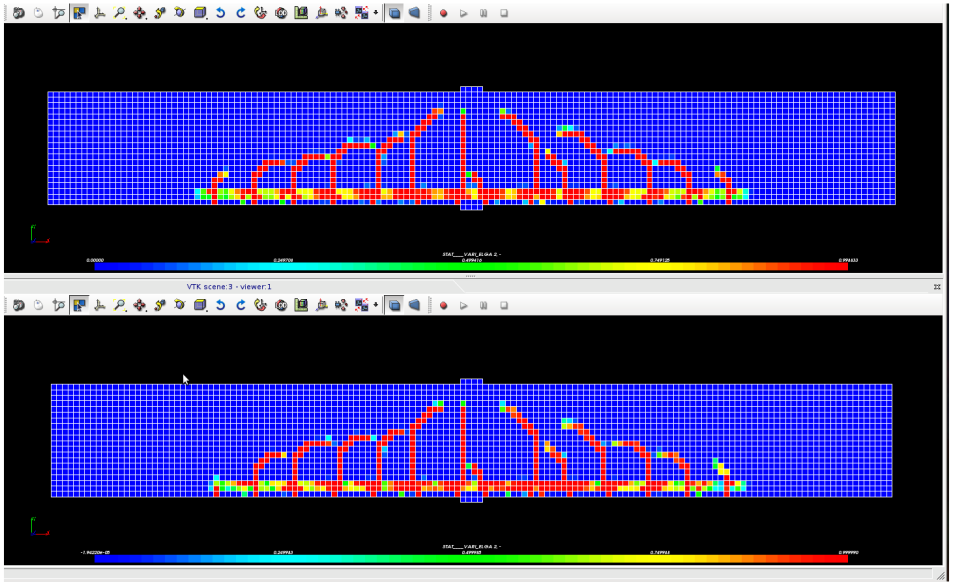




- loi de fluage tertiaire (avec endommagement) ▸ Hayhurst.mfront

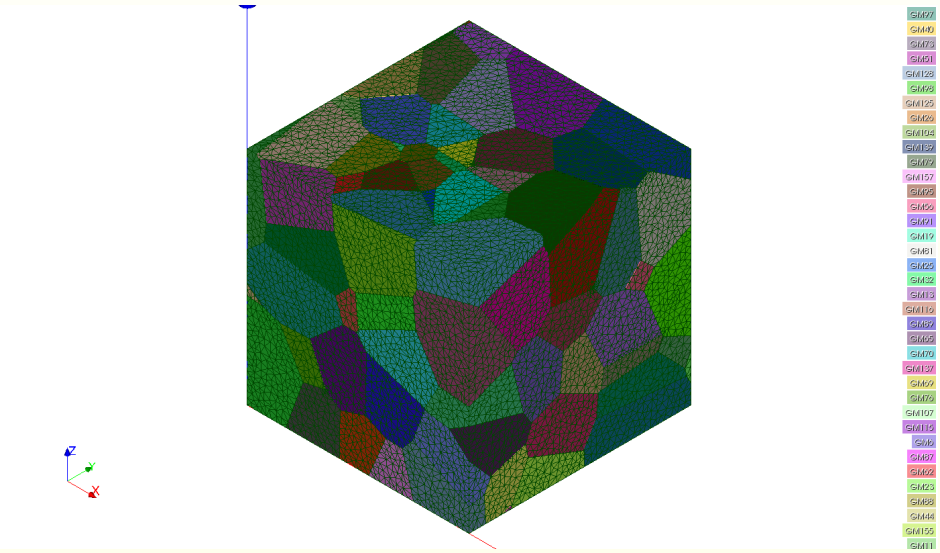
Mazars

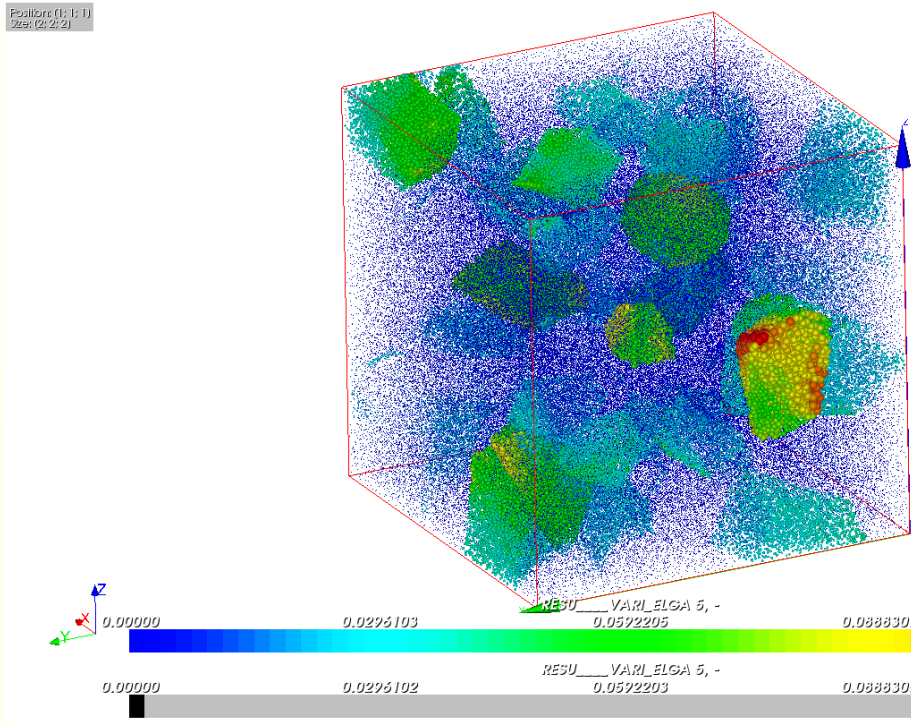




- loi de Mazars ▸ Mazars.mfront

lois monocristallines - MC—DD-CFC—DD-CC





- loi monocristalline, 172 grains ▸ monocrystal.mfront

à vos claviers !

formulation de la loi élastoplastique de Chaboche

Les équations du modèle sont résumées brièvement :

- contraintes déformations élastiques :
  $\underline{\sigma} = \underline{\mathbf{D}} : \ (\underline{\epsilon}^{\mathrm{to}} - \underline{\epsilon}^{\mathrm{p}})$
- Critère de plasticité :
  $F(\underline{\sigma}, \underline{X}) = (\sigma - \underline{X})_{\mathrm{eq}} - R(p) \leq 0$
- loi d'écoulement normale au critère :
  $\underline{\dot{\epsilon}}^{\mathrm{p}} = \dot{p}\,\underline{n}$ avec $\underline{n} = \dfrac{3}{2}\dfrac{\underline{\sigma}^{dev} - \underline{X}}{(\sigma - \underline{X})_{\mathrm{eq}}}$
- $\underline{X}$ écrouissages cinématiques : $\underline{X} = \underline{X}_1 + \underline{X}_2 + \ldots$;
- L'évolution de $\underline{X}_i$:
  $\underline{X}_i = \dfrac{2}{3} C_i \underline{\alpha}_i$ avec : $\underline{\dot{\alpha}}_i = \underline{\dot{\epsilon}}^{\mathrm{p}} - \gamma_i\,\underline{\alpha}_i\,\dot{p}$;
- l'écrouissage isotrope $R(p)$ est défini par :
  $R(p) = R^{\infty} + \left(R^0 - R^{\infty}\right)\exp(-b\,p)$;
- paramètres E, ν, R^0, R^{∞}, b, C_1, C_2,..., C_n, γ_1, γ_2,..., γ_n

discrétisation de la loi élastoplastique de Chaboche

Les inconnues sont : $\Delta\,\underline{\epsilon}^{\mathrm{el}}$; $\Delta\,p$; $\Delta\,\underline{\alpha}_i$;

- Test : si
  $F^{el}\left(\underline{\sigma},\underline{X}\right) = \left(\underline{\sigma}^{tr} - \underline{X}|_t\right)_{\mathrm{eq}} - R\left(p|_t\right) < 0$
  avec :
  $\underline{\sigma}^{tr} = \underline{D} : \left(\underline{\epsilon}^{\mathrm{el}}\big|_t + \Delta\,\underline{\epsilon}^{\mathrm{to}}\right)$
  alors la solution est élastique :
  $\Delta\,\underline{\epsilon}^{\mathrm{p}} = 0 \quad \Delta\,p = 0 \quad \Delta\,\underline{\alpha}_i = \underline{0}$
- Sinon, il faut résoudre le système suivant :
  $F(\sigma, X) = 0 \quad \Leftrightarrow$
  $$\begin{cases} \underline{\sigma}|_{t+\Delta\,t} - \underline{X}|_{t+\Delta\,t})_{\mathrm{eq}} - R(p(t+\Delta\,t)) = 0 \\ \Delta\,\underline{\alpha}_i - \Delta\,\underline{\epsilon}^{\mathrm{p}} + \gamma_i(\underline{\alpha}_i + \theta\Delta\underline{\alpha}_i)\Delta\,p = \underline{0} \\ \Delta\,\underline{\epsilon}^{\mathrm{el}} - \Delta\,\underline{\epsilon}^{\mathrm{to}} + \Delta\,\underline{\epsilon}^{\mathrm{p}} = \underline{0} \end{cases}$$
  où $\Delta\,\underline{\epsilon}^{\mathrm{p}} = \Delta\,p\ \ \underline{n}|_{t+\Delta\,t}$

intégration implicite dans `mfront`—1/3

```
@Parser Implicit;
@Behaviour Chaboche;
@Algorithm NewtonRaphson_NumericalJacobian;
@RequireStiffnessTensor;
@Theta 1.;
@MaterialProperty stress young;
@MaterialProperty real nu;
@MaterialProperty real R_inf;
@MaterialProperty real R_0;
@MaterialProperty real b;
@MaterialProperty real C[2];
@MaterialProperty real g[2];
@StateVariable real    p;
@StateVariable Stensor a[2];
@LocalVariable real Fel;
@InitLocalVars{ Stensor sigel=D*(eel+deto);
  for(unsigned short i=0;i!=2;++i){
      sigel-=C[i]*a[i]/1.5;}
  const real seqel = sigmaeq(sigel);
  const real Rpel  = R_inf + (R_0-R_inf)*exp(-b*p);
  Fel   = seqel - Rpel; // prediction elastique}
@ComputeStress{  sig = D*eel;}
```

intégration implicite dans mfront—2/3

```
@Integrator{
  if(Fel > 0){          // solution plastique
      // Les variables suivies de _ sont en t+theta*dt
      const real p_  = p +theta*dp ;
      const real Rp_ = R_inf + (R_0-R_inf)*exp(-b*p_) ;
      Stensor a_[2];
      Stensor sr = deviator(sig);
      for(unsigned short i=0;i!=2;++i){
          a_[i]       = a[i]+theta*da[i];
          sr          -= C[i]*a_[i]/1.5;
      } // tenseur Sigma-X
      const stress  seq = sigmaeq(sr);
      Stensor n = 1.5*sr/seq;
      feel = deel - deto + dp*n ;
      fp     = (seq-Rp_)/young; // pour normaliser
      for(unsigned short i=0;i!=2;++i){
        fa[i] = da[i] - dp*(n-g[i]*a_[i]);
      }
  } else {
      feel = deel - deto; // solution elastique }
}
```

intégration implicite dans `mfront`—3/3

opérateurs tangents

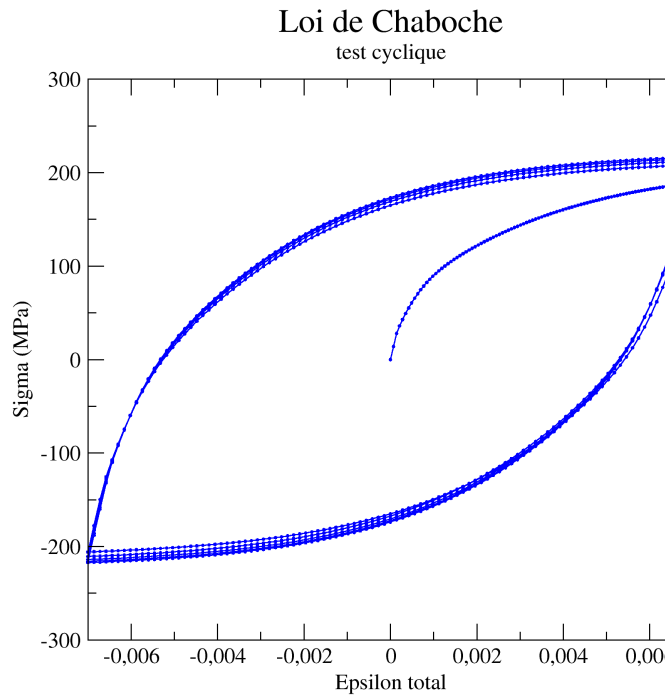
```
@TangentOperator{
  if ((smt==ELASTIC)||(smt==SECANTOPERATOR)){
    Dt=D; // matrice elastique
  } else if (smt==CONSISTANTTANGENTOPERATOR){
    Stensor4 Je;
    getPartialJacobianInvert(Je);
    Dt = D*Je; // matrice tangente coherente
  }
}
```

compilation :

```
mfront -obuild -interface=aster
Chaboche.mfront
```

petit test : cycles de traction-compression

```
@Behaviour<aster> 'src/libAsterBehaviour.
    so' 'asterchaboche';
@MaterialProperty<constant> 'young'
    200000. ;
@MaterialProperty<constant> 'nu' 0.33 ;
@MaterialProperty<constant> 'R_inf' 50. ;
@MaterialProperty<constant> 'R_0' 30. ;
@MaterialProperty<constant> 'b' 20. ;
@MaterialProperty<constant> 'C[0]'
    187000.;
@MaterialProperty<constant> 'C[1]' 45000.;
@MaterialProperty<constant> 'g[0]' 4460.;
@MaterialProperty<constant> 'g[1]' 340. ;
@ExternalStateVariable 'Temperature' 0.;
@Times {0.,5. in 1000};
@ImposedStrain 'EYY'
    {0.:0.,1.:0.007,2.:-0.007,3.:0.007,4.:-0.007,5.:0.007};
```



exercice : modèle viscoplastique

Le critère de plasticité : $F(\sigma, X) = (\sigma - X)_{\text{eq}} - R(p) \leq 0$

est remplacé par : $\dot{p} = \langle \frac{F}{K} \rangle^m$ où : $\langle F \rangle = \max(0, F)$

Dans le fichier `mfront`,

```
fp=(seq_-Rp_)/young;
```

devient :

```
fp -= pow(F*UNsurK,m)*dt;
```

Les propriétés matériau ajoutées : `UNsurK` et `m`

et si cela plante ? Comment faire ?

- erreurs de compilation souvent explicites ;
  - `Viscochab.mfront:94: error: 'F' was not declared in this scope`
  - `Viscochab.mfront:74: warning: unused variable 'Rp'`
  - `Viscochab.mfront:91: error: expected ',' or ';' before 'if'` (oubli d'un `";"` en fin de ligne)
- compilation avec `-debug` ;
- impression de variables `cout « "seq=" « seq « end; ;`
- compilation avec `CXXFLAGS='-g'` ;
- générer des fichiers mtest `@AsterGenerateMTestFileOnFailure=true;`

petit bilan

- ajouts de nouveaux comportements : "very easy !" ;
- déjà dans la base de tests de Code_Aster ! :

nom de la loi de comportement	test Code_Aster
loi élastoplastique de Chaboche	mfron01a
loi viscoplastique de Chaboche	mfron01b
loi viscoplastique de Hayhurst	mfron02a,b
loi d'endommagement de Mazars	mfron02c,d,e
loi de fluage de béton Burger	mfron02f,g
loi cristalline Méric-Cailletaud	mfron03b,c,d
loi cristalline DD_CFC (IRRA)	mfron03e,f,g
loi cristalline DD_CC (IRRA)	mfron03h,i
loi meta-lema-ani phases méta	mfron03j-n

tests de mfront : lois supplémentaires

- loi de LEMAÎTRE anisotrope : `StrainHardeningCreep.mfront`
- lois d'endommagement : `Lorentz.mfront` ,... ;
- lois CZM : `Tvergaard.mfront`,... ;
- lois monocristallines en grandes déformations `FiniteStrainMonoCristal.mfront`
- ...

72 lois de comportements dans la base de cas tests

contraintes planes (généralisées)

- déformation axiale $\underline{\epsilon}^{\text{to}}{}_z$ == variable interne supplémentaire ;
- hypothèses :
  - la déformation élastique $\underline{\epsilon}^{\text{el}}$ == variable interne ;
  - la déformation totale n'intervient que dans $f_{\underline{\epsilon}^{\text{el}}}$;
- partition des déformations :
  $f_{\underline{\epsilon}^{\text{el}}} = \Delta\underline{\epsilon}^{\text{el}} - \Delta\underline{\epsilon}^{\text{to}} - \Delta\underline{\epsilon}^{\text{to}}{}_z\, \vec{e}_z \otimes \vec{e}_z + \ldots$
- aucune des autres équations du système implicite n'est modifiée ;
- équation associée à $\underline{\epsilon}^{\text{to}}{}_z$: $f_{\underline{\epsilon}^{\text{to}}{}_z} = \dfrac{1}{E'}\sigma_z$
- contraintes planes généralisées : $f_{\underline{\epsilon}^{\text{to}}{}_z} = \dfrac{1}{E'}\left(\sigma_z - \sigma_z^{\text{équilibre}}\right)$

▸ contraintes planes

perspectives

- ajouts de nouveaux analyseurs spécifiques :
  - plasticité/viscoplasticité isotrope compressible ;
- matrice tangente cohérente :
  - facile pour algorithmes spécifiques et implicites ;
  - pour l'intégration explicite ou par BROYDEN ? :
- support de lois mécaniques générales :
  - transformations finies (presque fini) ;
  - couplages de lois de fluage, d'endommagement ;
  - autres physiques : thermique non linéaire, métallurgie ;
  - mécanique des sols : Cam-Clay, Drucker-Prager, THM,...
  - lois à gradient ;
  - ... ;

ANNEXES : algorithme global

algorithme global de mécanique quasi-statique

équilibre mécanique : trouver $\Delta\vec{u}$ tel que :

- $\vec{R}\left(\Delta\vec{u}\right) = \vec{O}$ avec $\vec{R}\left(\Delta\vec{u}\right) = \vec{F}_i\left(\Delta\vec{u}\right) - \vec{F}_e$
- force interne élémentaire : $\vec{F}_i^{elem} = \sum_{i=1}^{N_G} \left(\underline{\sigma}_{t+\Delta t}\left(\Delta\underline{\epsilon}^{\text{to}}, \Delta t\right) : \underline{\underline{\mathbf{B}}}\right) w_i$
- résolution par NEWTON-RAPHSON : $\Delta\vec{u}^{n+1} = \Delta\vec{u}^{n} - \underline{\underline{\mathbb{K}}}^{-1}.\vec{R}\left(\Delta\vec{u}^{n}\right)$
- calcul de la raideur élémentaire : $\underline{\underline{\mathbb{K}}}^{e} = \sum_{i=1}^{N^G} {}^{t}\underline{\underline{\mathbf{B}}} : \dfrac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}} : \underline{\underline{\mathbf{B}}}\, w_i$ où $\dfrac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}}$ est la *matrice tangente cohérente*.

▸ algorithmesMfront

algorithme global de mécanique quasi-statique

équilibre mécanique : trouver $\Delta\vec{u}$ tel que :

- $\vec{R}\left(\Delta\vec{u}\right) = \vec{O}$ avec $\vec{R}\left(\Delta\vec{u}\right) = \vec{F}_i\left(\Delta\vec{u}\right) - \vec{F}_e$
- force interne élémentaire : $\vec{F}_i^{elem} = \sum_{i=1}^{N_G} \left(\underline{\sigma}_{t+\Delta t}\left(\Delta\underline{\epsilon}^{\text{to}}, \Delta t\right) : \underline{\underline{\mathbf{B}}}\right) w_i$
- résolution par NEWTON-RAPHSON : $\Delta\vec{u}^{n+1} = \Delta\vec{u}^{n} - \underline{\underline{\mathbb{K}}}^{-1}.\vec{R}\left(\Delta\vec{u}^{n}\right)$
- calcul de la raideur élémentaire : $\underline{\underline{\mathbb{K}}}^{e} = \sum_{i=1}^{N^G} {}^{t}\underline{\underline{\mathbf{B}}} : \frac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}} : \underline{\underline{\mathbf{B}}}\, w_i$ où $\frac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}}$ est la *matrice tangente cohérente*.

▸ algorithmesMfront

algorithme global de mécanique quasi-statique

équilibre mécanique : trouver $\Delta\vec{u}$ tel que :

- $\vec{R}\left(\Delta\vec{u}\right) = \vec{O}$ avec $\vec{R}\left(\Delta\vec{u}\right) = \vec{F}_i\left(\Delta\vec{u}\right) - \vec{F}_e$
- force interne élémentaire : $\vec{F}_i^{elem} = \sum_{i=1}^{N_G} \left(\underline{\sigma}_{t+\Delta t}\left(\Delta\underline{\epsilon}^{\text{to}}, \Delta t\right) : \underline{\underline{\mathbf{B}}}\right) w_i$
- résolution par NEWTON-RAPHSON :

$$\Delta\vec{u}^{n+1} = \Delta\vec{u}^{n} - \left(\left.\frac{\partial\vec{R}}{\partial\Delta\vec{u}}\right|_{\Delta\vec{u}^n}\right)^{-1}.\vec{R}\left(\Delta\vec{u}^{n}\right) = \Delta\vec{u}^{n} - \underline{\underline{\mathbb{K}}}^{-1}.\vec{R}\left(\Delta\vec{u}^{n}\right)$$

- calcul de la raideur élémentaire : $\underline{\underline{\mathbb{K}}}^{e} = \sum_{i=1}^{N^G} {}^{t}\underline{\underline{\mathbf{B}}} : \frac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}} : \underline{\underline{\mathbf{B}}}\, w_i$ où $\frac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}}$ est la *matrice tangente cohérente*.

▸ algorithmesMfront

algorithme global de mécanique quasi-statique

équilibre mécanique : trouver $\Delta\vec{u}$ tel que :

- $\vec{R}\left(\Delta\vec{u}\right) = \vec{O}$ avec $\vec{R}\left(\Delta\vec{u}\right) = \vec{F}_i\left(\Delta\vec{u}\right) - \vec{F}_e$
- force interne élémentaire : $\vec{F}_i^{elem} = \sum_{i=1}^{N_G} \left(\underline{\sigma}_{t+\Delta t}\left(\Delta\underline{\epsilon}^{\text{to}}, \Delta t\right) : \underline{\underline{\mathbf{B}}}\right) w_i$
- résolution par NEWTON-RAPHSON : $\Delta\vec{u}^{n+1} = \Delta\vec{u}^{n} - \underline{\underline{\mathbb{K}}}^{-1}.\vec{R}\left(\Delta\vec{u}^{n}\right)$
- calcul de la raideur élémentaire : $\underline{\underline{\mathbb{K}}}^{e} = \sum_{i=1}^{N^G} {}^{t}\underline{\underline{\mathbf{B}}} : \dfrac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}} : \underline{\underline{\mathbf{B}}}\, w_i$ où $\dfrac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{\text{to}}}$ est la *matrice tangente cohérente*.

▸ algorithmesMfront

ANNEXES : matrice tangente

une façon générique de calculer la tangente cohérente

Il faut calculer : $\dfrac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}} = \left.\dfrac{\partial \underline{\sigma}}{\partial \underline{\epsilon}^{\mathrm{el}}}\right|_{\underline{\epsilon}^{\mathrm{el}}+\Delta \underline{\epsilon}^{\mathrm{el}}} : \dfrac{\partial \Delta \underline{\epsilon}^{\mathrm{el}}}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}}$

On a résolu : $F\left(\Delta Y, \Delta \underline{\epsilon}^{\mathrm{to}}\right) = 0$ avec : $[\Delta Y]^T = \left[\Delta \underline{\epsilon}^{\mathrm{el}} \ , \ \Delta\alpha\right]$

Par différentiation : $\dfrac{\partial F}{\partial \Delta Y}\,\mathrm{d}\,\Delta Y + \dfrac{\partial F}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}}\,\mathrm{d}\,\Delta \underline{\epsilon}^{\mathrm{to}} = 0$

$\dfrac{\partial F}{\partial \Delta Y}$ est la jacobienne J, connue après la résolution.

Hyp. l'incrément de déformation $\Delta \underline{\epsilon}^{\mathrm{to}}$ n'apparaît que dans :

$$F_\epsilon = \Delta \underline{\epsilon}^{\mathrm{el}} + \Delta \underline{\epsilon}_i^{\mathrm{p}} - \Delta \underline{\epsilon}^{\mathrm{to}} = 0$$

donc : $J\,\mathrm{d}\,\Delta Y = -\dfrac{\partial F}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}}\,\mathrm{d}\,\Delta \underline{\epsilon}^{\mathrm{to}} = \begin{pmatrix} \mathrm{d}\,\Delta \underline{\epsilon}^{\mathrm{to}} \\ 0 \end{pmatrix}$

Du 1er bloc on déduit : $\mathrm{d}\,\Delta \underline{\epsilon}^{\mathrm{el}} = J^{-1}_{\underline{\epsilon}^{\mathrm{el}}} : \mathrm{d}\,\Delta \underline{\epsilon}^{\mathrm{to}}$ où $J^{-1}_{\underline{\epsilon}^{\mathrm{el}}}$ est la partie supérieure gauche de J^{-1}.

Finalement, nous obtenons : $\dfrac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\epsilon}^{\mathrm{to}}} = D : J^{-1}_{\underline{\epsilon}^{\mathrm{el}}}$

$J^{-1}_{\underline{\epsilon}^{\mathrm{el}}}$ est calculée par getPartialJacobianInvert dans le bloc

ANNEXES : quelques lois en `mfront`

présentation de la loi de Hayhurst R5.03.13

- $\sigma = (1 - D)\ C\ \varepsilon^e$;
- $\underline{\varepsilon^e} = \underline{\varepsilon} - \dot{p}\ \underline{n}$ avec $\underline{n} = 1.5 \frac{\sigma^{dev}}{\sigma_{eq}}$;
- $\dot{p} = \varepsilon_0\ sinh\left(\frac{\sigma_{eq}(1-H)}{K(1-D)(1-\phi)}\right)$;
- $H = H_1 + H_2$;
- $\dot{H}_i = \frac{h_i}{\sigma_{eq}} \left(H_i^* - \delta_i H_i\right) \dot{p}$;
- $\dot{D} = A_0\ sinh\left(\frac{\alpha_D\ <tr(\sigma)>_+ + \sigma_{eq}\ (1-\alpha_D)}{\sigma_0}\right)$

implantation de la loi de Hayhurst —1/3

```
@Parser Implicit;
@Behaviour Hayhurst;
@IterMax 100 ;
@MaterialProperty stress young;
@MaterialProperty real nu;
@MaterialProperty real rho;
@MaterialProperty real alpha;
@MaterialProperty real K;
@MaterialProperty real epsi0;
@MaterialProperty real sigma0;
@MaterialProperty real h1;
@MaterialProperty real h2;
@MaterialProperty real H1star;
@MaterialProperty real H2star;
@MaterialProperty real A0;
@MaterialProperty real alphaD;
@MaterialProperty real delta1;
@MaterialProperty real delta2;
@MaterialProperty real sequid;
@Includes{
#include"TFEL/Material/Lame.hxx"
}
```

implantation de la loi de Hayhurst —2/3

```
@StateVariable real     p;
@StateVariable real     H1;
@StateVariable real     H2;
@StateVariable real     d;

@LocalVariable real     lambda;
@LocalVariable real     mu;
@InitLocalVars{  using namespace tfel::material::lame;
  lambda = computeLambda(young,nu);
  mu = computeMu(young,nu);}

@ComputeStress{
  if(d > 1.-1.e-8){
    sig= Stensor(0.);
  } else {
    sig = (1.-d)*(lambda*trace(eel)*Stensor::Id()+2*mu*
    eel);
  }
}
```

implantation de la loi de Hayhurst —3/3

```
@Integrator{  real seq = sigmaeq(sig);
  Stensor sig0=lambda*trace(eel)*Stensor::Id()+2*mu*eel;
   real seq0 = sigmaeq(sig0);
  if(seq > 1.e-8*young){
     real H1_=H1+theta*dH1; real H2_=H2+theta*dH2;
     real d_=d+theta*dd; const real H_=H1_+H2_;
     real shp  = sinh(seq*(1-H_)/K/(1-(d_)));
     real chp  = sqrt(1.+shp*shp) ;
     real trsig=max(trace(sig),0.); const real inv_seq =
     1/seq;
     real shd= sinh((alphaD*trsig+(1-alphaD)*seq)/sigma0
    );
     real chd= sqrt(1.+shd*shd) ; const real dtrsde=(3.*
    lambda+2.*mu)*theta*(1.-d_)*trsig/trace(sig);
    Stensor n  = 1.5*deviator(sig)*inv_seq;
    feel += dp*n-deto;
    fp   = dp-epsi0*dt*shp;
    fH1  = dH1-h1*dp*(H1star-delta1*H1_)*inv_seq ;
    fH2  = dH2-h2*dp*(H2star-delta2*H2_)*inv_seq ;
```

retour

implantation de la loi de Mazars —1/3

```
@Parser DefaultParser;
@Behaviour mazars;
@MaterialProperty stress young;
@MaterialProperty real nu;
@MaterialProperty real Ac;
@MaterialProperty real At;
@MaterialProperty real Bc;
@MaterialProperty real Bt;
@MaterialProperty real k;
@MaterialProperty real ed0;
@ProvidesSymmetricTangentOperator;
@Includes{#include"TFEL/Material/Lame.hxx"}
@StateVariable real d;
@StateVariable real Y;
@StateVariable real eeqcor;
@LocalVariable real    lambda;
@LocalVariable real    mu;
@InitLocalVars{using namespace tfel::material::lame;
  lambda = computeLambda(young,nu);
  mu = computeMu(young,nu);}
```

implantation de la loi de Mazars —2/3

```
@Integrator{ using namespace tfel::material::lame;
  real e1,e2,e3;
  real s1,s2,s3;
  real ppe1,ppe2,ppe3;
  real pns1,pns2,pns3;
  real pps1,pps2,pps3;
  const Stensor e  = eto+deto;
  const real    tr = trace(e);
  const Stensor s0 = lambda*tr*Stensor::Id()+2*mu*e;
  const real dmax=0.99999;
  e.computeEigenValues(e1,e2,e3);
  // eigen values of s0
  s1 = 2*mu*e1+lambda*tr;
  s2 = 2*mu*e2+lambda*tr;
  s3 = 2*mu*e3+lambda*tr;
  const real sn = max(abs(s1),max(abs(s2),abs(s3)));
  ppe1=max(0.,e1);
  ppe2=max(0.,e2);
  ppe3=max(0.,e3);
  pps1=max(0.,s1);
  pps2=max(0.,s2);
  pps3=max(0.,s3);
```

implantation de la loi de Mazars —3/3

```
real r=1.;
if (sn>1.e-6*young){
    r=(pps1+pps2+pps3)/(abs(s1)+abs(s2)+abs(s3));
}
real gam=1. ;
if ((min(s1,min(s2,s3))<0.)&&(r==0.)){
    pns1=min(0.,s1);
    pns2=min(0.,s2);
    pns3=min(0.,s3);
    gam=-sqrt(pns1*pns1+pns2*pns2+pns3*pns3)/(pns1+pns2
  +pns3);
}
real eeqc= sqrt(ppe1*ppe1+ppe2*ppe2+ppe3*ppe3);
eeqcor=max(gam*eeqc,eeqcor);
real A=At*(2*r*r*(1.-2*k)-r*(1-4*k))+Ac*(2*r*r-3*r+1);
real B=r*r*Bt+(1-r*r)*Bc;
real Y1=max(ed0,eeqcor);
Y=max(Y1,Y);
d=max(d,1-(1-A)*ed0/Y-A*exp(-B*(Y-ed0)));
d=min(dmax,d);
sig = (1.-d)*s0;
```

expression d'une loi cristalline

- $\Delta\underline{\varepsilon^e} = \Delta\underline{\varepsilon} - \Delta\underline{\varepsilon^p}$
- $\Delta\underline{\varepsilon^p}$ est déduit des glissements de chaque système : $\Delta\underline{\varepsilon^p} = \sum_{s=1,12} \Delta\gamma_s \underline{M_s}$
- Ceux-ci sont obtenus pour chaque système de glissement par : $\Delta\gamma_s = \Delta p_s \mathrm{sgn}(\tau_s - C\alpha_s)$ avec $\Delta p_s = \Delta t \left\langle \frac{|\tau_s - C\alpha_s| - R(p_s)}{K} \right\rangle^m$
- Ecrouissage isotrope : $R(p_s) = R_0 + Q \sum_r h_{sr} \left(1 - \exp(-bp_r)\right)$ h_{sr} matrice d'interaction entre systèmes.
- Ecrouissage cinématique : $\Delta\alpha_s = \Delta\gamma_s - D\alpha_s \Delta p_s$
- Avec : $\tau_s = \underline{\sigma} : \underline{M_s} = \underline{\sigma} : \frac{1}{2}\left(\underline{m_s} \otimes \underline{n_s} + \underline{n_s} \otimes \underline{m_s}\right)$
  - $\underline{n_s}$ et $\underline{m_s}$ sont les normales et directions de glissement.
  - L'élasticité peut être isotrope ou orthotrope : $\underline{\sigma} = \underline{\underline{D}}\left(\underline{\varepsilon^e}\right)$

implantation d'une loi cristalline —1/3

```
@Parser       Implicit;
@Behaviour  monocrystal;
@Algorithm  NewtonRaphson_NumericalJacobian;
@OrthotropicBehaviour;
@RequireStiffnessTensor;
@MaterialProperty  real m;
@MaterialProperty  real K;
@MaterialProperty  real C;
@MaterialProperty  real R0;
@MaterialProperty  real Q;
@MaterialProperty  real b;
@MaterialProperty  real d1;
@StateVariable              real g[12];
@AuxiliaryStateVariable  real p[12];
@AuxiliaryStateVariable  real a[12];
@TangentOperator{
     Stensor4 Je;
     getPartialJacobianInvert(Je);
     Dt = D*Je;   }
@Import "MonoCrystal_CFC_SlidingSystems.mfront";
@Import "MonoCrystal_InteractionMatrix.mfront";
@ComputeStress{   sig = D*eel;}
```

implantation d'une loi cristalline —2/3

```
@Integrator{  StrainStensor vepsp(real(0));
  real tau[12], vp[12], va[12], ag[12];
  real tma[12], tmR[12],Rp[12], pe[12] ;
  for(unsigned short i=0;i!=12;++i){
    ag[i]  = abs(dg[i]);
    pe[i] = Q*(1.-exp(-b*(p[i]+theta*ag[i]))) ;  }
  for(unsigned short i=0;i!=12;++i){
    Rp[i] = R0 ;
    for(unsigned short j=0;j!=12;++j){
        Rp[i] +=mh(i,j)*pe[j] ;      }
    tau[i] = mus[i] | sig ;
    va[i]  = (dg[i]-d1*a[i]*ag[i])/(1.+d1*theta*ag[i]);
    tma[i] = tau[i]-C*(a[i]+theta*va[i]) ;
    tmR[i] = abs(tma[i])-Rp[i] ;
    if (tmR[i]>0.){real sgn=tma[i]/abs(tma[i]);
      vp[i]  = dt*sgn*pow((tmR[i]/K),m);     }
    else{  vp[i]=0.;     }
    vepsp+=vp[i]*mus[i] ;  }
  feel += vepsp-deto;
  for(unsigned short i=0;i!=12;++i){
```

implantation d'une loi cristalline —3/3

```
    fg[i]  -= vp[i];  }
}
@UpdateAuxiliaryStateVars{
for(unsigned short i=0;i!=12;++i){
  p[i]+=abs(dg[i]);
  a[i]+=(dg[i]-d1*a[i]*abs(dg[i]))/(1.+d1*abs(dg[i]));}}
```

```
// MonoCristal_CFC_SlidingSystems
@LocalVariable tfel::math::tvector<12,StrainStensor> mus
    ;
@InitLocalVariables{
  const real nx[12]={ 1.0,1.0,1.0, 1.0, 1.0,
    1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0};
  const real ny[12]={ 1.0,1.0,1.0,-1.0,-1.0,-1.0, 1.0,
    1.0, 1.0,-1.0,-1.0,-1.0};
  const real nz[12]={ 1.0,1.0,1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0};
  const real mx[12]={-1.0, 0.0,-1.0,-1.0,0.0,1.0,
    0.0,1.0,1.0,-1.0,1.0,0.0};
```

retour

expression discretisée de la loi meta-lema-ani

- $\Delta\varepsilon^e - \Delta\varepsilon + \Delta p\,\underline{n} = 0$
- $\sqrt{\underline{\sigma} : \underline{\underline{\mathbf{M}}} : \underline{\sigma}} - \sum_{i=1,3} f_i[Z]\sigma_{vi} = 0$

avec :

$$\sigma_{vi} = a_i \left(e^{Q_i/T}\right)^{1/n_i} (p^- + \Delta p)^{m_i} \left(\frac{\Delta p}{\Delta t}\right)^{1/n_i}$$

meta-lema-ani —1/5

```
@Parser       Implicit;
@Behaviour  metalemani;
@Includes{  #include<TFEL/Material/Hill.hxx>
            #include<TFEL/Material/Lame.hxx>
}
@OrthotropicBehaviour;
@Algorithm  NewtonRaphson_NumericalJacobian;
@Theta  1.;  @Epsilon  1.e-10;
@MaterialProperty  real  young;
young.setGlossaryName("YoungModulus");
@MaterialProperty  real  nu;
nu.setGlossaryName("PoissonRatio");
@MaterialProperty  real  a[3];
@MaterialProperty  real  m[3];
@MaterialProperty  real  pn[3];
@MaterialProperty  real  Q[3];
@MaterialProperty  real  M1[6];
@MaterialProperty  real  M3[6];
@StateVariable  real  p;
@AuxiliaryStateVariable  real  seq;
@AuxiliaryStateVariable  real  svi[3];
```

meta-lema-ani —2/5

```
@LocalVariable stress lambda;
@LocalVariable stress mu;
@LocalVariable tfel::math::st2tost2<N,real> H;
@LocalVariable real T_ ;
@LocalVariable real invn[3], f[3], gamma[3], sv[3]  ;
// variables de commande aster
@ExternalStateVariable real SECH,HYDR,IRRA,NEUT1,NEUT2,
    CORR,ALPHPUR,ALPHBETA;
@IsTangentOperatorSymmetric true;
@TangentOperator{using namespace tfel::material::lame;
     StiffnessTensor Hooke;      Stensor4 Je;
     computeElasticStiffness<N,Type>::exe(Hooke,lambda,mu
    );
     getPartialJacobianInvert(Je);
     Dt = Hooke*Je;  }
@InitLocalVariables{
  using namespace tfel::material::lame;
  lambda = computeLambda(young,nu);
  mu = computeMu(young,nu);
  // proportion en phase alpha en milieu de pas de temps
  const real Z = min(max(ALPHPUR  + theta*dALPHPUR+
    ALPHBETA + theta*dALPHBETA,0.),1.);
```

meta-lema-ani —3/5

```
// fonctions f
if          (Z >= 0.99) { f[0]=1. ;
} else if (Z >= 0.9)  { f[0] = (Z-0.9)/0.09 ;
} else                { f[0] = 0. ;  }

if          (Z >= 0.1)  { f[2]=0. ;
} else if (Z >= 0.01) { f[2] = (0.1-Z)/0.09 ;
} else                { f[2] = 1. ;  }

if          (Z >= 0.99) { f[1]=0. ;
} else if (Z >= 0.9)  { f[1] = 1.0-(Z-0.9)/0.09 ;
} else if (Z >= 0.1)  { f[1] = 1.0 ;
} else if (Z >= 0.01) { f[1] = 1.0-(0.1-Z)/0.09 ;
} else                { f[1] = 0. ;  }
// Temperature Aster en Celsius
T_ = 273.0 + T + theta * dT ;
for(unsigned short i=0;i!=3;++i){
  invn[i] = 1.0 / pn[i] ;
  gamma[i] = a[i]* exp(Q[i]/T_*invn[i]) ;  }
```

meta-lema-ani —4/5

```
// correspondance M aster (repere x,y,z) et H
  real M[6];
  if          (Z >= 0.99) {for(unsigned short i=0;i!=6;++i
    ){M[i]=M1[i];}
  } else if (Z >= 0.01) {for(unsigned short i=0;i!=6;++i
    ){M[i]=Z*M1[i]+(1.-Z)*M3[i];}
  } else                    {for(unsigned short i=0;i!=6;++i
    ){M[i]=M3[i];}}
  const real H_F = 0.5*( M[0]+M[1]-M[2]);
  const real H_G = 0.5*(-M[0]+M[1]+M[2]);
  const real H_H = 0.5*( M[0]-M[1]+M[2]);
  const real H_L = 2.0*M[3];
  const real H_M = 2.0*M[4];
  const real H_N = 2.0*M[5];
  H = hillTensor<N,real>(H_F,H_G,H_H,H_L,H_M,H_N);
}
@ComputeStress{
  sig = lambda*trace(eel)*Stensor::Id()+2*mu*eel;}
@Integrator{
  const real sigeq = sqrt(sig|H*sig);
  real p_=p+theta*dp ;
  real sigv =  0. ;  real pm[3];  real dpn[3] ;
```

meta-lema-ani —5/5

```
  for(unsigned short i=0;i!=3;++i){
    pm[i]  = (p_ > 0.) ? pow(p_,m[i])          : 0.;
    dpn[i] = (dp > 0.) ? pow((dp/dt),invn[i]) : 0. ;
    sv[i]=gamma[i]*pm[i]*dpn[i] ;
    sigv += f[i]*sv[i] ; }
  Stensor  n(0.);
  if(sigeq > 1.e-10*young){ n= (H*sig)/sigeq; }
  feel += dp*n-deto;
  fp    = (sigeq-sigv)/young;
}
@UpdateAuxiliaryStateVars{
  for(unsigned short i=0;i!=3;++i){ svi[i]=sv[i] ; }
}
```

gestion des propriétés matériau

propriétés matériaux

- introduction en 3 étapes :
  - écriture d'une fonction `Young=f(Temperature)` ;
  - création d'une librairie `libInconel600.so` ;
  - appel depuis `Cast3M` ou `cyrano` (`Code_Aster` à venir)
- voir la documentation de `mfront`

Exemple en mfront

```
@Parser     MaterialLaw;
@Material   Inconel600;
@Law        YoungModulus  ;
@Input TK;

TK.setGlossaryName("Temperature");

@Output E;
@PhysicalBounds TK in [0:*[;
@Bounds TK in [0:*[;

@Function
{
  const real TC = TK-273.15;
  E=(-3.1636e-3*TC*TC-3.8654*TC+2.1421e+4)*1e7;
}
```

retour

gestion des contraintes planes

contraintes planes (généralisées) —1/3

- exemple de la loi de NORTON.
- nouvelle variable interne : déformation axiale ϵ_z^{to},
- contrainte imposée variable externe

```
@StateVariable<PlaneStress> strain etozz;
PlaneStress::etozz.setGlossaryName("AxialStrain");

@StateVariable<AxisymmetricalGeneralisedPlaneStress>
    strain etozz;
AxisymmetricalGeneralisedPlaneStress::etozz.
    setGlossaryName("AxialStrain");
@ExternalStateVariable<
    AxisymmetricalGeneralisedPlaneStress> stress sigzz;
AxisymmetricalGeneralisedPlaneStress::sigzz.
    setGlossaryName("AxialStress");
```

contraintes planes (généralisées) —2/3

Partie spécifique auxe contraintes planes :

```
@Integrator<PlaneStress,Append,AtEnd>{
  // the plane stress equation is satisfied at the end
    of the time
  // step
  const stress szz = (lambda+2*mu)*(eel(2)+deel(2))+
    lambda*(eel(0)+deel(0)+eel(1)+deel(1));
  fetozz    = szz/young;
  // modification of the partition of strain
  feel(2) -= detozz;
  // jacobian
  dfeel_ddetozz(2)=-1;
  dfetozz_ddetozz  = real(0);
  dfetozz_ddeel(2) = (lambda+2*mu)/young;
  dfetozz_ddeel(0) = lambda/young;
  dfetozz_ddeel(1) = lambda/young;
}
```

contraintes planes (généralisées) —3/3

Partie spécifique aux contraintes planes généralisées :

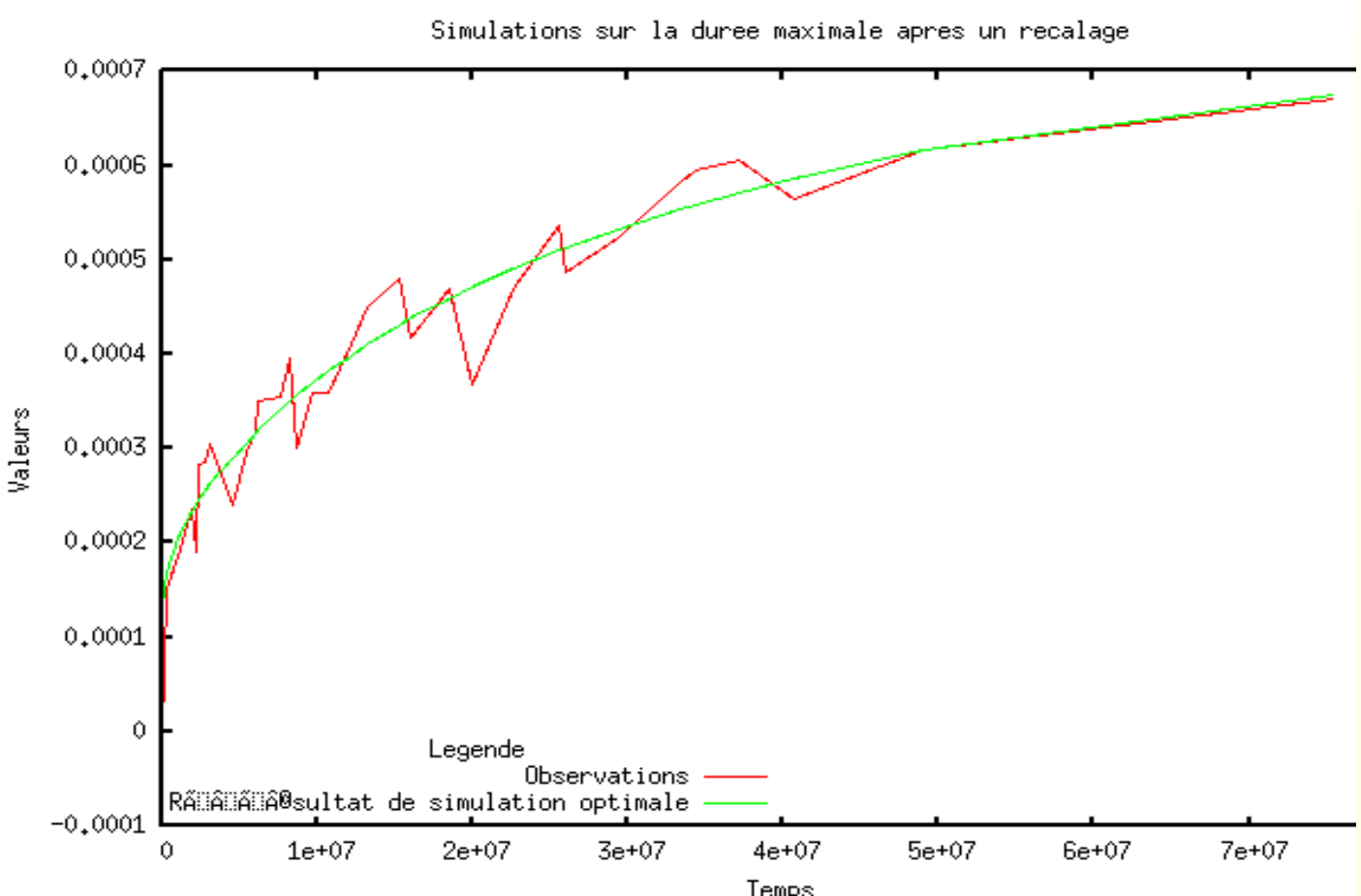
```
@Integrator<AxisymmetricalGeneralisedPlaneStress,Append,
    AtEnd>{
  // plane stress equation is satisfied at end
  const stress szz = (lambda+2*mu)*(eel(1)+deel(1))+
    lambda*(eel(0)+deel(0)+eel(2)+deel(2));
  fetozz    = (szz-sigzz-dsigzz)/young;
  // modification of the partition of strain
  feel(1) -= detozz;
  // jacobian
  dfeel_ddetozz(1)=-1;
  dfetozz_ddetozz  = real(0);
  dfetozz_ddeel(1) = (lambda+2*mu)/young;
  dfetozz_ddeel(0) = lambda/young;
  dfetozz_ddeel(2) = lambda/young;
}
```

retour

adao

principe d'Adao

- algorithmes d'optimisation pour trouver les paramètres X ;
- qui minimisent l'écart $F = Y^{obs} - H(X)$;
- Y^{obs} valeurs observées (expérimentales, ou autre) ;
- $H(X)$ valeurs simulées (par `mtest` ou `Code_Aster` ...) ;



appel de mtest(python) par Adao

```
t = temps_instants
t[1] = 1.e-1

m = MTest()
setVerboseMode(VerboseLevel.VERBOSE_QUIET)
m.setPredictionPolicy(PredictionPolicy.
LINEARPREDICTION)
```

```
m.setMaterialProperty('Cini', 1.)
m.setExternalStateVariable("Temperature", 300.)
m.setExternalStateVariable("C", 1.)
m.setImposedStress('SXX', {0.: 0., 1.: 4E6, 3.E7: 4
E6})
s = MTestCurrentState()
wk = MTestWorkSpace()
m.completeInitialisation()
m.initializeCurrentState(s)
m.initializeWorkSpace(wk)
YY1 = [0]
for i in range(0, len(t) - 1):
```

retour