Development in Z-set

Sims group

Mines ParisTech, Centre des Matériaux/CNRS UMR

March 12, 2019

Plan

1. **Behavior generalities**
2. **ZebFront behaviors**
   - Explicite integration
   - Implicit integration
3. **Behavior development in Zébulon**
   - Behavior in C++ language
   - Material object
4. **Use of Z-mat with Abaqus**

Plan

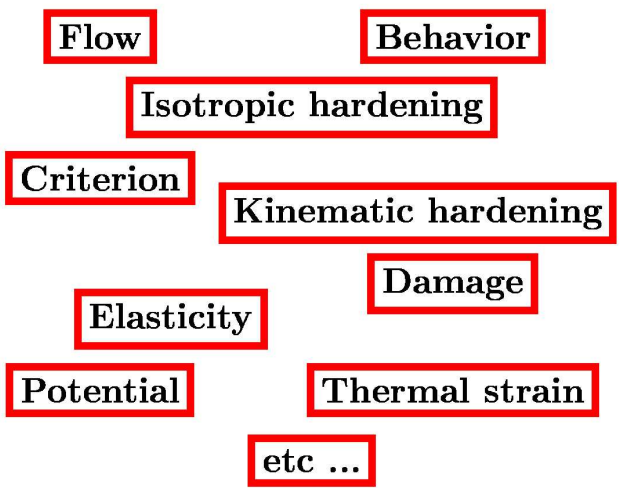
1. **Behavior generalities**
2. **ZebFront behaviors**
   - Explicite integration
   - Implicit integration
3. **Behavior development in Zébulon**
   - Behavior in C++ language
   - Material object
4. **Use of Z-mat with Abaqus**

The Zmat library

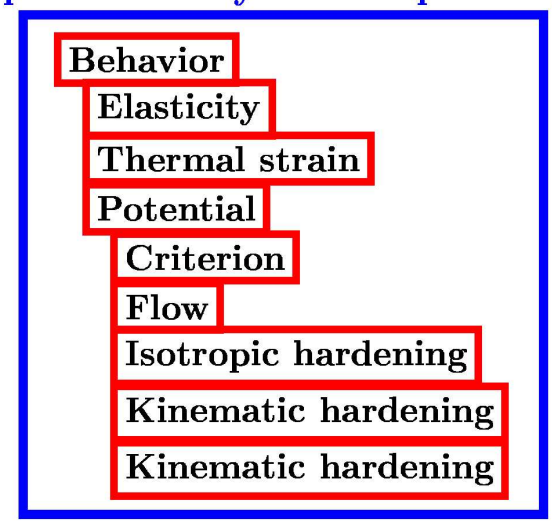
Z-MAT, librairie matériaux interfacée avec les grands codes commerciaux de mécanique des structures

- Many models
  - Elasticity, hyperelasticity
  - Plasticity, viscoplasticity
  - Linear or non-linear isotropic and kinematic hardening
  - Crystallographic plasticity
  - Multi-mat
- Many tools
  - Creation of new models in a data file by assembly of material predifined objects
  - Unlimited dependency of the coefficients on internal variables and external parameters

Material objects



Typical assembly for viscoplasticity



Introduction

Pourquoi est-il important de savoir implanter les lois de comportement dans les codes par éléments finis ?

- Peu de lois sont disponibles dans les grands codes (exemple ABAQUS: version simple de GTN).
- Traitement spécifique du comportement et de la rupture.
- ...

Introduction

Nécessité d'implanter de nouvelles lois non disponibles dans les codes de calculs (Zset, ABAQUS, ...)

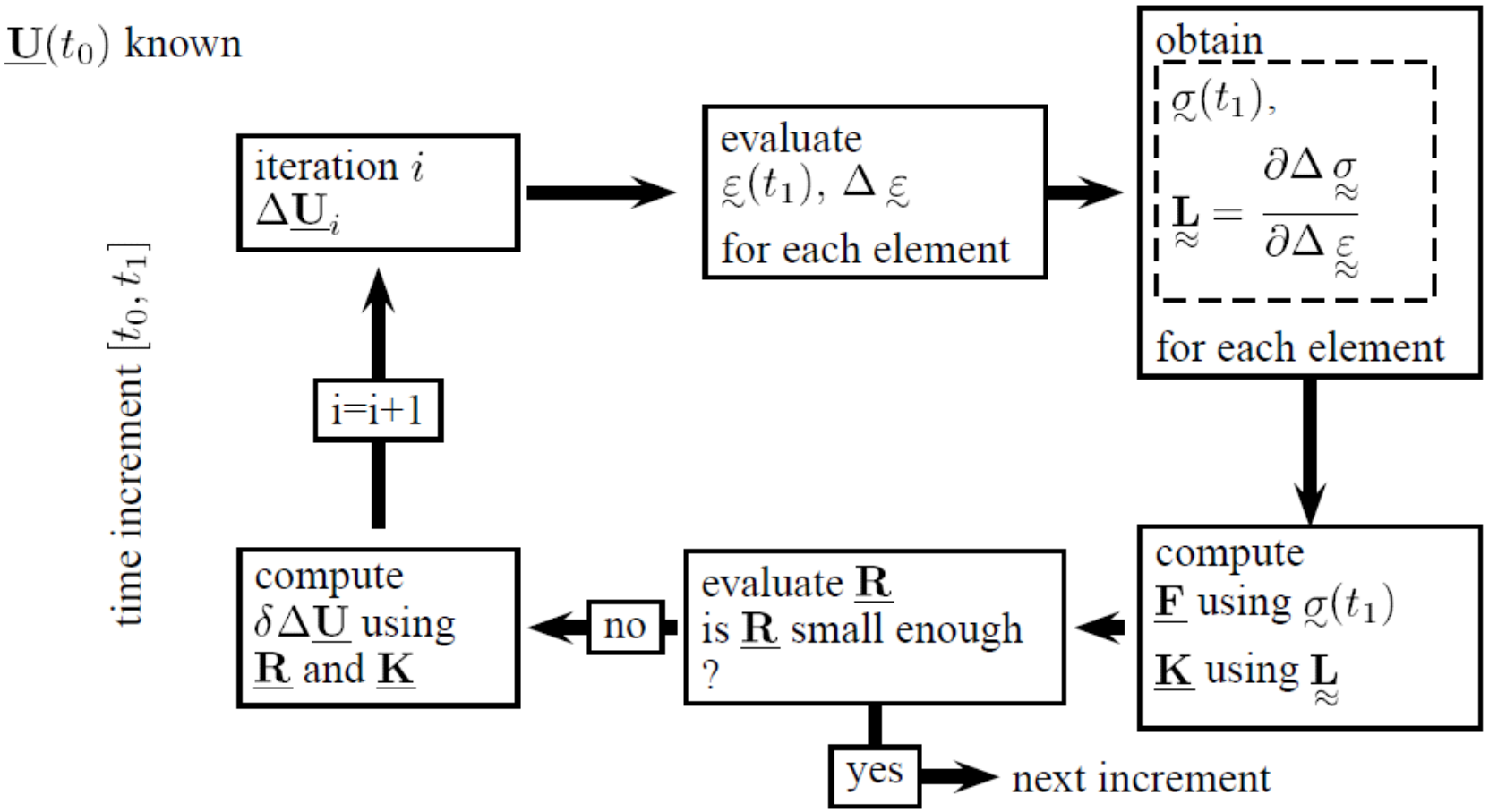
Possibilité d'implanter de nouvelles lois via des subroutines utilisateurs

- UMAT et UTHERM dans le code ABAQUS.
- *ZeBFRoNT* ou via la classe BEHAVIOR en c++. *ZeBFRoNT*: C'est le générateur de code pour loi de comportement

Difficultés rencontrées au cours de la mise en œuvre numérique:

- Nécessité d'un savoir-faire au niveau du choix de l'algorithme : algorithme spécifique pour chaque loi de comportement.
- Maîtrise du langage de programmation suggéré par le code pour le développement des subroutines.
- Connaître les entrées/ sorties des subroutines utilisateurs.

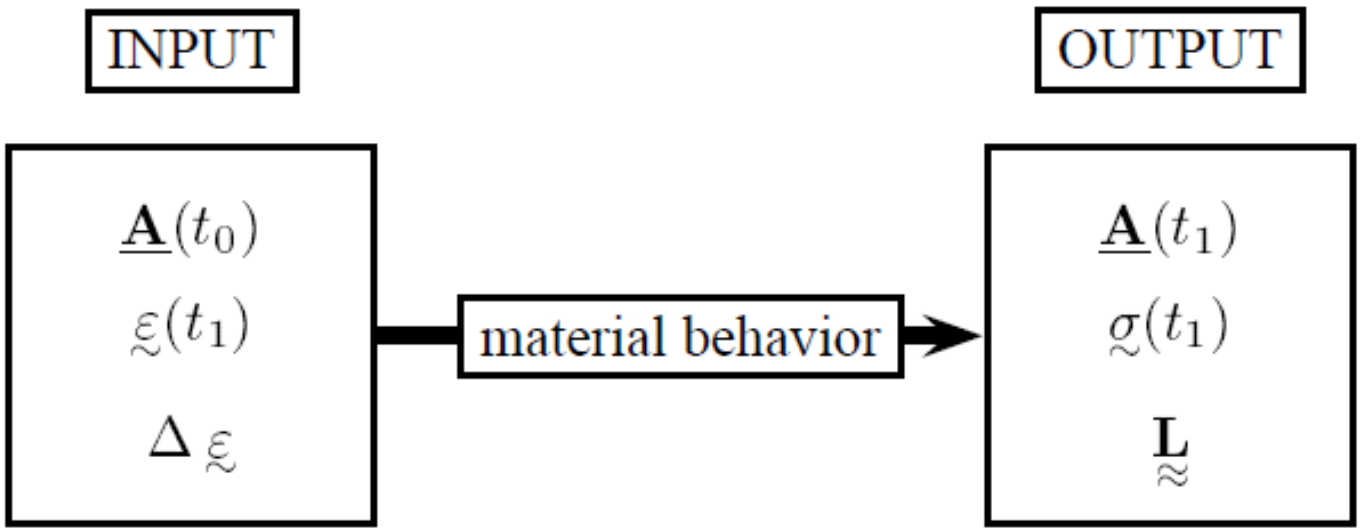
Place du comportement dans la méthode des éléments finis



Interface "générique" comportement/MEF

Pour chaque point de gauss:

$$\Delta t = t_1 - t_0$$



Behavior generalities

$$\frac{d\underline{\mathbf{A}}}{dt} = \dot{\underline{\mathbf{A}}} = \underline{\mathbf{G}}(\underline{\mathbf{A}}, t)$$

$$\frac{A_i}{dt} = \dot{\underline{\mathbf{A}}}_i = G_i(A_1, \ldots, A_n, t)$$

- Le temps (t) apparait ici pour faire figurer la déformation imposée mais aussi un **paramètre extérieur** en temps et espace comme une température par exemple ($T(\vec{x}, t)$).
- Evaluer la loi de comportement = intégrer d'équation précedente de t_0 à t_1.
- En règle générale : $\underline{\mathbf{A}} = (\underset{\sim}{\varepsilon}_e, \ldots)$ de sorte que $\underset{\sim}{\sigma}(t_1) = \underset{\approx}{\mathbf{E}}(t_1) : \underset{\sim}{\varepsilon}_e(t_1)$,
- reste le calcul de $\underset{\approx}{\mathbf{L}}$...

Behavior generalities

- Explicit integration: Runge-Kutta
  - simple à mettre en oeuvre car elle n'utilise que l'équation différentielle.
  - instable si le pas de temps n'est pas suffisamment petit.
  - L'intégration peut être gourmande en temps CPU.

- Implicit integration: $\theta - method$
  - difficle à mettre en oeuvre (calcul matriciel, inversion d'une matrice ...).
  - universellement stable.
  - Meilleur convergence.

Explicite	Implicite
facile à implanter	difficile à implanter
lent	rapide
$\underset{\approx}{\mathbf{L}}$?	calcul de $\underset{\approx}{\mathbf{L}}$

Behavior generalities

Ecriture de la matrice jacobienne . . . un travail délicat

- Ecriture par blocs

$$\underline{\mathbf{J}} = \begin{pmatrix} \dfrac{\partial R_e}{\partial \Delta \underset{\sim}{\varepsilon}_e} & \dfrac{\partial R_e}{\partial \Delta p} \\ \dfrac{\partial R_p}{\partial \Delta \underset{\sim}{\varepsilon}_e} & \dfrac{\partial R_p}{\partial \Delta p} \end{pmatrix}$$

- Matrice tangente cohérente

$$\underset{\approx}{\mathbf{L}} = \frac{\partial \Delta \underset{\sim}{\sigma}}{\partial \Delta \underset{\sim}{\varepsilon}}$$

$$\Delta \underset{\sim}{\sigma} = \underset{\approx}{\mathbf{E}} : (\Delta : \underset{\sim}{\varepsilon} - \Delta p \, \underset{\sim}{\mathbf{n}})$$

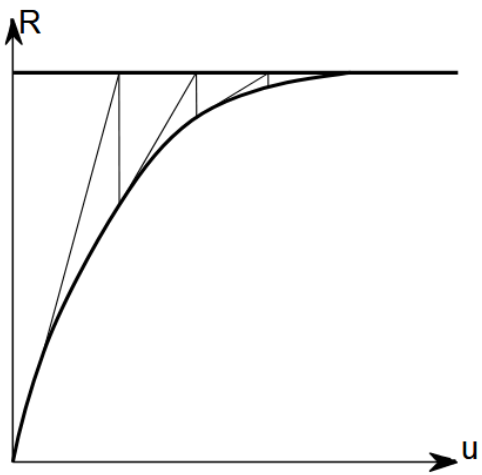


Figure 1.2.2-1: matrice tangente réactualisée à chaque itération

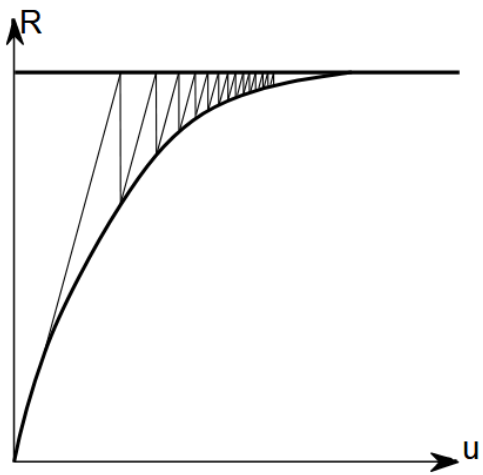


Figure 1.2.2-2: matrice élastique

Behavior generalities

- *External parameters* (`ep`) imposed as input
- *Integrated variables* (`vint`)
- *Auxiliary variables* (`vaux`), just for output
- *Coefficients* (`coef`), material parameters
  (can depend on ep, vint, vaux)
- *Primal and dual variables*, prescribed variables and associated fluxes

Behavior generalities

problem	primal	dual
mechanics, small perturbation	$\underset{\sim}{\varepsilon}$	$\underset{\sim}{\sigma}$
mechanics, large deformation	$\underset{\sim}{\boldsymbol{F}}$	$\underset{\sim}{\boldsymbol{S}}$
thermal pb	$(T, \underline{\mathbf{grad}\ T})$	$(H, \underline{\boldsymbol{q}})$
diffusion	concentration	flux
electrostatics	$\underline{\mathbf{grad}\ V}$	$\underline{\boldsymbol{E}}$
magnetostatics	$rot\ \underline{\boldsymbol{A}}$	$\underline{\boldsymbol{H}}$

$\underset{\sim}{\varepsilon}$ strain tensor, $\underset{\sim}{\boldsymbol{F}}$ deformation gradient, T temperature, V electric potential, $\underline{\boldsymbol{A}}$ potential vector, $\underset{\sim}{\sigma}$ Cauchy stress tensor, $\underset{\sim}{\boldsymbol{S}}$ second Piola–Kirchhoff stress tensor, H enthalpy, $\underline{\boldsymbol{q}}$ thermal flux , $\underline{\boldsymbol{E}}$ electric field , $\underline{\boldsymbol{H}}$ magnetic field.

Management of Zébulon development projects

- Configuration of the project: **Zsetup**
  Takes as input a "library_files" describing the project
  Generates "Makefile.dat" with architecture-independent commands

```
!DYNAMIC
!LIB_BASED
!USE_INC                                        # use standard includes $Z7PATH/include
!BFLAGS -L${Z7PATH}/PUBLIC/lib-${Z7MACHINE}     # path of standard Zebulon libraries
!INC src                                        # declares header files in src/
!SRC src src                                    # declares source files in src/
!DEBUG src
# Generates a program named Surf_$Z7MACHINE from source files in src/
# Links with standard libraries Zmat_base
!TARGET Surf src Zmat_base
!!RETURN
```

- Compilation/Link: **Zmake**
  Generates a real "makefile" for the architecture from "Makefile.dat"
  Compilation and link of the project

Plan

1. Behavior generalities

2. **ZebFront behaviors**
   - Explicite integration
   - Implicit integration

3. Behavior development in Zébulon
   - Behavior in C++ language
   - Material object

4. Use of Z-mat with Abaqus

Introdcution

- *ZeBFRoNT*: C'est le générateur de code pour loi de comportement:
- Pré-processeur, qui réutilise les classes de base
- Produit un fichier C++

  ZebFront fichier.z $\Longrightarrow$ fichier.c

- fichier.c est utilisé comme tout fichier du projet

Plan

1. Behavior generalities

2. **ZebFront behaviors**
   - Explicite integration
   - Implicit integration

3. Behavior development in Zébulon
   - Behavior in C++ language
   - Material object

4. Use of Z-mat with Abaqus

ZebFront example for explicit integration

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

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

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

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

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

ZebFront example for explicit integration

ZebFront code for explicit (runge-kutta) integration (1/2)

```
#include <Basic_nl_behavior.h>
#include <Basic_nl_simulation.h>
#include <Elasticity.h>

@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
@Name     norton;
@SubClass ELASTICITY elasticity;
@Coefs    K, n;
@Coefs    R0, Q, b;
@tVarInt  eel;     // tensorial integrated variable
@sVarInt  evcum;  // scalar integrated variable
@tVarAux  evi;
};

// This block is called at the end of the integration
@StrainPart {
evi  = eto - eel;
sig = *elasticity*eel;
// Note no consistent tg matrix !
m_tg_matrix=*elasticity;
}
```

Norton viscoplastic Model

header files (classes Zset/c++)

Créer une nouvelle classe NORTON
Nom du comportement norton
Objet matrice d'élasticité
Coeffs de Norton K et n
Coeffs de L'écrouissage isotrope R_0, Q et b
Variable interne tensorielle ε_e
Variable interne scalaire: p
Variable auxiliaire tensorielle ε_p

Après intégration:
Partition de la déformation $\underset{\sim}{\epsilon} = \underset{\sim}{\epsilon}_{el} + \underset{\sim}{\epsilon}_{v}$
Calcul de la contrainte $\underset{\sim}{\sigma} = \underset{\approx}{\boldsymbol{E}} : \underset{\sim}{\epsilon}_{el}$
Matrice tangente approchée (RK !)

ZebFront example for explicit integration

ZebFront code for explicit (runge-kutta) integration (1/2)

```
// Specify dvarint = ...
for each integrated variable named varint
//@Derivative explicit integration function

calculating variable time derivatives
@Derivative {
sig = *elasticity*eel;

TENSOR2 sprime = deviator(sig);
double  J      = sqrt(1.5*(sprime|sprime));
double  R      = R0 + Q*(1-exp(-b*evcum));
double  f      = J - R;

if (f<=0.0) {
devcum = 0.0;
deel = deto;
resolve_flux_grad(*elasticity, deel, deto);

}


else  {
devcum  = pow(f/K,n);
TENSOR2 norm = sprime*(1.5/J);
TENSOR2 dein = devcum*norm;

deel = deto - dein;
resolve_flux_grad(*elasticity, deel, deto, dein);
}
}
```

Norton viscoplastic Model

Calculation of the vector time derivative $\dot{Y}$

$\underset{\sim}{\sigma} = \underset{\sim}{\underset{\sim}{E}} : \underset{\sim}{\epsilon}_{el}$

Calcul du déviateur $\underset{\sim}{S}$

Écrouissage isotrope $R = R0 + Q(1 - e^{-bp})$

Calcul du deuxième invariant $J = \sqrt{\frac{3}{2}\underset{\sim}{s} : \underset{\sim}{s}}$

La fonction de charge $f(\underset{\sim}{\sigma}) = J - R$

Élasticité $J <= R$

$\underset{\sim}{\epsilon} = \underset{\sim}{\epsilon}_{el}$ et $\underset{\sim}{\epsilon}_{v} = p = 0$

Plasticité $J > R$

loi d'écoulement de Norton : $\dot{p} = \left\langle \frac{f(\underset{\sim}{\sigma})}{K} \right\rangle^{n}$

Direction de l'écoulement $\underset{\sim}{n} = \frac{3}{2J}\underset{\sim}{s}$

L'évolution de la déformation plastique $\underset{\sim}{\dot{\epsilon}}_{v} = \dot{p}\,\underset{\sim}{n}$

$\underset{\sim}{\dot{\epsilon}}_{el} = \underset{\sim}{\dot{\epsilon}} - \underset{\sim}{\dot{\epsilon}}_{v}$

```
if (f<=0.0) {
devcum = 0.0;
deel = deto;
resolve_flux_grad(*elasticity, deel, deto);
deel = deto;
}
else  {
devcum  = pow(f/K,n);
TENSOR2 norm = sprime*(1.5/J);
TENSOR2 dein = devcum*norm;
deel = deto - dein;
resolve_flux_grad(*elasticity, deel, deto, dein);
}
```

Note resolve_flux_grad() is only needed in simulator mode for mixed loadings:

```
// input: mat, dein, d_grad (mixed strain/stress)
// output: deel, d_grad (strain)
void BASIC_SIMULATOR::resolve_flux_grad(const SMATRIX& mat, TENSOR2& deel,
TENSOR2& dgrad, const TENSOR2& dein) {  ...  }
```

Plan

1. Behavior generalities

2. **ZebFront behaviors**
   - Explicite integration
   - Implicit integration

3. Behavior development in Zébulon
   - Behavior in C++ language
   - Material object

4. Use of Z-mat with Abaqus

ZebFront code with implicit integration

```
@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
@Name     norton;
@SubClass ELASTICITY elasticity;
@Coefs    K, n;
@Coefs    R0, Q, b;
@tVarInt  eel;
@sVarInt  evcum;
@Implicit
};
@StrainPart {
evi  = eto - eel;
sig  = *elasticity*eel;
if (integration&LOCAL_INTEGRATION::THETA_ID) {  // calculation of consistent tgmat
TENSOR4 tmp(psz,f_grad,0,0);                    // from jacobian see note
if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
else           m_tg_matrix=*elasticity;
} else { if (m_flags&CALC_TG_MATRIX) m_tg_matrix=*elasticity; }
}

@Derivative {
// Same as before ...
}
```

ZebFront code with implicit integration

ZebFront code for implicite (Theta method) integration

```
//@CalcGradF implicit integration function
//for the variable residual and Jacobian matrix
@CalcGradF {
ELASTICITY& E=*elasticity;
sig = E*eel;
double       R    = R0+Q*(1-exp(-b*evcum));
TENSOR2 sprime = deviator(sig);
double  J      = sqrt(1.5*(sprime|sprime));
double  f      = J - R;
```

```
f_vec_eel -= deto;
if ( (f>0.0 && devcum>=0) || (devcum>0.0) ) {
TENSOR2        norm             = sprime*(1.5/J);
f_vec_eel      += norm*devcum;
f_vec_evcum    -= dt*pow(f/K,n);
```

```
// predefined to 1.5 * deviator operator
SMATRIX dn_ds = (unit32 - norm^norm)/J;
SMATRIX dn_deel = dn_ds*E;
// tdt predefined to theta*dt
double  dv_df   = tdt*n*pow(f/K,n-1)/K;
TENSOR2 df_fs   = dv_df*norm;
```

Norton viscoplastic Model

CalcGradF: Residual and jacobian :

At time $t + \theta \Delta t$:

f_vec is the the residual vector, initialized to dvar

$$\mathcal{R} = (\underset{\sim}{\boldsymbol{R}}^e, R^p) == f\_vec = \{f\_vec\_eel, f\_vec\_evcum\} = \mathbf{0}$$

$$f\_vec\_eel = R_{el} = \Delta \underset{\sim}{\epsilon}_{el} - \Delta \underset{\sim}{\epsilon} - \Delta p \underset{\sim}{\boldsymbol{n}}$$

$$f\_vec\_evcum = R_p = \Delta p - \Delta t \left(\frac{J - R}{K} \right)^n$$

f_vec_var is the residual for int variable var:

$f\_vec\_eel = \Delta \underset{\sim}{\epsilon}_{el}$ et $f\_vec\_evcum = \Delta p$

$$\frac{\partial \underset{\sim}{\boldsymbol{n}}}{\partial \underset{\sim}{\boldsymbol{\sigma}}} = \frac{1}{J} \left(\frac{3}{2} \underset{\sim}{\underset{\sim}{\boldsymbol{I}}}_s - \underset{\sim}{\boldsymbol{n}} \otimes \underset{\sim}{\boldsymbol{n}} \right) \quad \underset{\sim}{\underset{\sim}{\boldsymbol{I}}}_s$$

such that $\underset{\sim}{\underset{\sim}{\boldsymbol{I}}}_s : \underset{\sim}{\boldsymbol{\sigma}} = \underset{\sim}{\boldsymbol{s}} \quad \ldots \frac{\partial \underset{\sim}{\boldsymbol{n}}}{\partial \underset{\sim}{\epsilon}} = \frac{\partial \underset{\sim}{\boldsymbol{n}}}{\partial \underset{\sim}{\boldsymbol{\sigma}}} * E$

Residual vector

Calcul des blocs ($R_e = \Delta\, \underset{\sim}{\varepsilon}_e + \Delta p\, \underset{\sim}{\mathbf{n}}^\theta - \Delta\varepsilon$)

$$\frac{\partial R_e}{\partial \Delta\, \underset{\sim}{\varepsilon}_e} = \underset{\approx}{\mathbf{1}} + \Delta p \frac{\partial\, \underset{\sim}{\mathbf{n}}}{\partial\, \underset{\sim}{\sigma}} : \frac{\partial\, \underset{\sim}{\sigma}}{\partial\, \underset{\sim}{\varepsilon}_e} : \frac{\partial\, \underset{\sim}{\varepsilon}_e}{\partial \Delta\, \underset{\sim}{\varepsilon}_e}$$

$$\underset{\approx}{\mathbf{N}} \qquad \underset{\approx}{\mathbf{E}}^\theta \qquad \theta\, \underset{\approx}{\mathbf{1}}$$

$$\underset{\approx}{\mathbf{N}} = \frac{1}{\sigma_{\text{eq}}} \left(\frac{3}{2} \underset{\approx}{\mathbf{J}} - \underset{\sim}{\mathbf{n}} \otimes \underset{\sim}{\mathbf{n}} \right)$$

$$\Rightarrow \quad \frac{\partial R_e}{\partial \Delta\, \underset{\sim}{\varepsilon}_e} = \underset{\approx}{\mathbf{1}} + \Delta p\, \underset{\approx}{\mathbf{N}}^\theta : \underset{\approx}{\mathbf{E}}^\theta$$

$$\frac{\partial R_e}{\partial \Delta p} = \underset{\sim}{\mathbf{n}}^\theta$$

ZebFront code with implicit integration

ZebFront code for implicite (Theta method) integration

```
deel_deel          += theta*devcum*dn_deel;

deel_devcum         = norm;

devcum_devcum      += dv_df*Q*b*exp(-b*evcum);

devcum_deel        -= df_fs*E;
}
}
```

Norton viscoplastic Model

Calculation of the jacobian matrix J: f_grad

At time $t + \theta\Delta t$:

$$[\mathbf{J}] = \frac{\partial \mathcal{R}}{\partial \Delta \mathbf{Z}} = \mathbf{1} - \Delta t \left.\frac{\partial \mathbf{F}}{\partial \Delta \mathbf{Z}}\right|^{t+\Theta\Delta t} = \begin{bmatrix} \frac{\partial \underset{\sim}{\boldsymbol{R}}_e}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}} & \frac{\partial \underset{\sim}{\boldsymbol{R}}_e}{\partial \Delta p} \\ \frac{\partial R_p}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}} & \frac{\partial R_p}{\partial \Delta p} \end{bmatrix}$$

diagonal blocks for the jacobian are initialized to 1, $[\mathbf{J}] = 1 * \underset{\sim}{\boldsymbol{I}}$

$$deel\_deel = \frac{\partial R_{el}}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}} = \underset{\sim}{\mathbf{1}} + \Delta p \frac{\partial \underset{\sim}{\boldsymbol{n}}}{\partial \underset{\sim}{\boldsymbol{\sigma}}} : \frac{\partial \underset{\sim}{\boldsymbol{\sigma}}}{\partial \underset{\sim}{\boldsymbol{\epsilon}}_{el}} : \frac{\partial \underset{\sim}{\boldsymbol{\epsilon}}_{el}}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}}$$

$$= \underset{\sim}{\mathbf{1}} + \theta \Delta p \frac{\partial \underset{\sim}{\boldsymbol{n}}}{\partial \underset{\sim}{\boldsymbol{\sigma}}} : \underset{\sim}{\boldsymbol{E}}$$

$$deel\_devcum = \frac{\partial R_{el}}{\partial \Delta p} = \underset{\sim}{\boldsymbol{n}}$$

$$devcum\_devcum = \frac{\partial R_p}{\partial \Delta p} = 1 - \frac{\partial (\Delta t \left(\frac{J-R}{K}\right)^n)}{\partial \Delta p}$$

$$devcum\_deel = \frac{\partial R_p}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}} = \underset{\sim}{\mathbf{1}} - \frac{\partial (\Delta t \left(\frac{J-R}{K}\right)^n)}{\partial \underset{\sim}{\boldsymbol{\sigma}}} : \frac{\partial \underset{\sim}{\boldsymbol{\sigma}}}{\partial \underset{\sim}{\boldsymbol{\epsilon}}_{el}} : \frac{\partial \underset{\sim}{\boldsymbol{\epsilon}}_{el}}{\partial \Delta \underset{\sim}{\boldsymbol{\epsilon}}_{el}}$$

Notes: Incremental consistent tangent matrix

```
if (integration&LOCAL_INTEGRATION::THETA_ID) {  // calculation of consistent tgmat
TENSOR4 tmp(psz,f_grad,0,0);                    // from jacobian see note
if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
else           m_tg_matrix=*elasticity;
} else { if (m_flags&CALC_TG_MATRIX) m_tg_matrix=*elasticity; }
```

After convergence,

$$\begin{pmatrix} d\Delta\underset{\sim}{\varepsilon} \\ 0 \end{pmatrix} = [\mathrm{J}] \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon}^e \\ d\Delta\alpha_I \end{pmatrix} \ldots \text{then} \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon}^e \\ d\Delta\alpha_I \end{pmatrix} = [\mathrm{J}]^{-1} \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon} \\ 0 \end{pmatrix}$$

$$[\mathrm{J}]^{-1} = \left(\begin{array}{c|c} H & x \\ \hline x & x \end{array}\right), \text{ with } H = \frac{\partial \Delta\underset{\sim}{\varepsilon}^e}{\partial \Delta\underset{\sim}{\varepsilon}}$$

Consistent tangent matrix:

$$\underset{\approx}{\boldsymbol{L}}_c = \frac{\partial \Delta\underset{\sim}{\boldsymbol{\sigma}}}{\partial \Delta\underset{\sim}{\varepsilon}^e} : \frac{\partial \Delta\underset{\sim}{\varepsilon}^e}{\partial \Delta\underset{\sim}{\varepsilon}} = \underset{\approx}{\boldsymbol{\Lambda}} : H$$

ZebFront code with non-linear kinematic hardening (1/2)

$$f(\underset{\sim}{\sigma}) = J(\underset{\sim}{\sigma} - \underset{\sim}{X}) - R$$

$$\underset{\sim}{X} = \frac{2}{3} C \underset{\sim}{\alpha} \qquad \dot{\underset{\sim}{\alpha}} = \dot{p}\left(\underset{\sim}{n} - D\underset{\sim}{\alpha}\right)$$

```
@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
// same as before ...
@Coefs    C1, D1;
@tVarInt  alpha1;
};
@StrainPart {
// same as before ...
}
@Derivative {
double CC1=C1/1.5;
sig = *elasticity*eel;
TENSOR2 Xv1    = CC1*alpha1;
TENSOR2 sigeff = sig - Xv1;
TENSOR2 sprime = deviator(sigeff);
double  J      = sqrt(1.5*(sprime|sprime));
// same as before
...
if(CC1>0.) dalpha1  = devcum*(norm - D1*Xv1/CC1);
...
}
```

ZebFront code with non-linear kinematic hardening (2/2)

```
@CalcGradF {
double CC1 = C1/1.5;
TENSOR2 Xv1 = CC1*alpha1;
TENSOR2 sigeff = sig - Xv1;
TENSOR2 sprime = deviator(sigeff);
double  J      = sqrt(1.5*(sprime|sprime));
// same as before
...
if(CC1>0.) {
SMATRIX dn_dal1 = dn_ds*CC1;
TENSOR2 m1     = norm - D1*Xv1/CC1;
f_vec_alpha1  -= devcum*m1;
deel_dalpha1      -= dn_dal1;
dalpha1_deel      -= dn_deel;
dalpha1_dalpha1   += dn_dal1; dalpha1_dalpha1.add_to_diagonal(tdv*D1);
dalpha1_devcum    -= m1;
devcum_dalpha1     = df_fs*CC1;
}
...
}
```

Plan

1. Behavior generalities
2. ZebFront behaviors
   - Explicite integration
   - Implicit integration
3. **Behavior development in Zébulon**
   - Behavior in C++ language
   - Material object
4. Use of Z-mat with Abaqus

Plan

1. Behavior generalities
2. ZebFront behaviors
   - Explicite integration
   - Implicit integration
3. **Behavior development in Zébulon**
   - Behavior in C++ language
   - Material object
4. Use of Z-mat with Abaqus

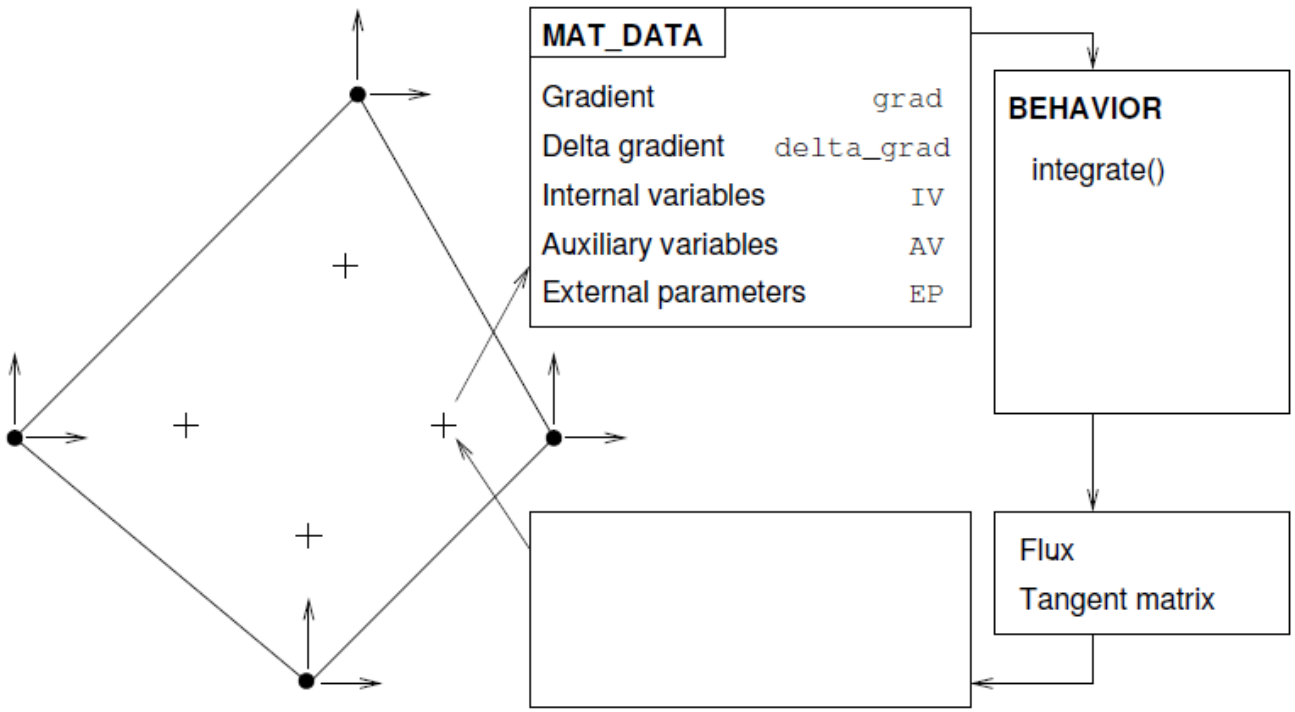
C++ example (Header file)

```
Z_START_NAMESPACE;                          //Debut du namespace Z_set
class NORTON_ISO_C : public BEHAVIOR,  //This is a base class for all behavior(constitutive equations).
                     public THETA_INTEGRATOR,
                     public RUNGE_INTEGRATOR {
 public :
     AUTO_PTR<ELASTICITY>     elasticity;
     COEFF           K, n, R0, Q, b;
     TENSOR2_VINT          eel;
     SCALAR_VINT           evcum;
     TENSOR2_VAUX          evi;
     TENSOR2_FLUX          sig;
     TENSOR2_GRAD          eto;
     TENSOR2 deto;
     AUTO_PTR<RUNGE>       runge_theta;
     AUTO_PTR<THETA>       theta_integ;
     MATRIX       m_tg_matrix;        // storage for the tangent matrix
     VECTOR       f_vec;              // residual
     VECTOR       d_chi;              // increment of integrated vars
     VECTOR       f_0;                // what f_vec is compared to f_0
     SMATRIX      f_grad;             // the jacobian matrix
     VECTOR dgrad;
  public :
    NORTON_ISO_C() ;
    virtual ~NORTON_ISO_C() ;
    virtual void initialize(ASCII_FILE&,int,LOCAL_INTEGRATION*);
    virtual void derivative(double,const  VECTOR&,VECTOR&);
   //explicit integration function calculating variable time derivatives
    virtual void calc_grad_f(VECTOR&,SMATRIX&,const VECTOR&,const VECTOR&,double,double);
 //implicit integration function for the variable residual and Jacobian matrix
    INTEGRATION_RESULT* integrate(MAT_DATA&,const VECTOR& delta_grad,MATRIX*&,int);
  //user-integrate function; used for behaviors which are not using the standard Runge-Kutta
or theta-method integration
};
Z_END_NAMESPACE;       //Fin du namespace Z_set
```

Mdata

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

//STORED_VARIABLE_SPEC : This class is used to handle the cutting up of the behavior variables into
usable matematical entities. The class has internal, auxiliary, flux, or gradient variables specific
 to the class coupling with the spacial problem.
```



C++ example

```
Z_USE_NAMESPACE;       //Utilisation du namespace Z_set
DECLARE_OBJECT(BEHAVIOR,NORTON_ISO_C,norton_iso_c)
void NORTON_ISO_C::initialize(ASCII_FILE& file, int dim, LOCAL_INTEGRATION* integ)
{
  BEHAVIOR::initialize(file,dim,integ);
  eel.initialize(this,"eel",tsz(),1); //tsz() : problem (tensor) size sym
  evcum.initialize(this,"evcum",1);
  evi.initialize(this,"evi",tsz(),1);
  sig.initialize(this,"sig",tsz(),1);
  eto.initialize(this,"eto",tsz(),1);

   for (;;) {
       STRING str=file.getSTRING();
       if (str.start_with("***") ) { file.back(); break; }
       else if (str=="**model_coef") {
            str=file.getSTRING();
            while ((!str.start_with("*")) && file.ok) {
                if      (str=="K") K.read("K",file,this);
                else if (str=="n") n.read("n",file,this);
                else if (str=="R0") R0.read("R0",file,this);
                else if (str=="Q") Q.read("Q",file,this);
                else if (str=="b") b.read("b",file,this);
                else {  INPUT_ERROR("Unknown coefficient: "+str); exit(1); }
                str=file.getSTRING();
            } file.back();
       }

       else if (str=="**elasticity" || str=="**ELASTICITY") {
            elasticity=ELASTICITY::read(file,this);
       }

       else INPUT_ERROR("Unknown command: "+str);
   }
}
```

C++ example

```
    if (integ) {
      integration=integ->type();
      if (integration&LOCAL_INTEGRATION::RUNGE_ID) {
           runge_theta=(RUNGE*)integ;
           runge_theta->set_integrator((RUNGE_INTEGRATOR*)this);
      }
      else if (integration&LOCAL_INTEGRATION::THETA_ID) {
           theta_integ=(THETA*)integ;
           theta_integ->set_integrator((THETA_INTEGRATOR*)this);
      }
      else ERROR("Integration method is not allowed");
    }
    else {
       runge_theta=(RUNGE*)Create_object(LOCAL_INTEGRATION, "runge_kutta");
       runge_theta->initialize(1.e-3, 1.e-3);
       runge_theta->set_integrator((RUNGE_INTEGRATOR*)this);
       integration=runge_theta->type();
    }
  m_tg_matrix.resize(tsz()); m_tg_matrix=0.;
}


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

C++ example

```
INTEGRATION_RESULT* NORTON_ISO_C::integrate(MAT_DATA& mdat,const VECTOR& delta_grad, MATRIX*& tg_matrix, int)
{ INTEGRATION_RESULT* ok =NULL;
  Timer_counter.start_time("Local Integration");
  tg_matrix = &m_tg_matrix;

  attach_all(mdat);

//attach_all( MAT DATA& mdat ) Used to assign all the object's STORED VARIABLE SPEC objects
onto the current integration point data. This method is usually called in the behavior
integrate method as: The method is virtual so a derived class may use the opportunity to do initialization only
required once for every integrate call (rare).
  calc_local_coefs();

//The list of COEFF* pointers for all the COEFF objects which exist in the current
behavior will automatically update every time the BEHAVIOR method calc_local_coefs is called.

  set_var_aux_to_var_aux_ini();
  set_var_int_to_var_int_ini();
//In your integration methods, theses methods should be used in place of manipulating those vectors themselves.
//These local methods only copy the portion of vectors which belong specifically to the behavior.

  dgrad = delta_grad;
  deto.reassign(eto.size(),dgrad,0);
```

C++ example

```
   if (integration&LOCAL_INTEGRATION::THETA_ID) {
       VECTOR f_0=0.;
       calc_material(theta_integ->give_theta());
       SMATRIX f_grad(int_sz()); VECTOR d_chi(int_sz()), f_vec(int_sz());
       d_chi =0.;

       ok=theta_integ->theta_method(f_vec,f_grad,curr_mat_data->var_int(vint_index), d_chi,f_0,Time_ini,Dtime);
       if (ok) return ok;
       calc_material(1.0);
       VECTOR dvar = curr_mat_data->var_int(vint_index) - curr_mat_data->var_int_ini(vint_index);
       attach(curr_mat_data->var_int(vint_index), dvar);

// attach( VECTOR& chi, VECTOR& d chi ) This alternate of the preceeding method may be
used for cases where the trial increment of the integrated variables is interesting. The default
method merely calls the above single parameter method.
       TENSOR4 tmp(tsz(),f_grad,0,0);
       if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
       else           m_tg_matrix=*elasticity;
  }
  else if (integration&LOCAL_INTEGRATION::RUNGE_ID) {
     if(Dtime!=0.) {
       dgrad/=Dtime;
       ok=runge_theta->runge_kutta(Time_ini,Dtime,curr_mat_data->var_int(vint_index));
     }
     if (CALC_TG_MATRIX) m_tg_matrix=*elasticity;
  }
  else  NOT_IMPLEMENTED_ERROR("Bad or unknown integration");

  evi=eto-eel;
  sig = *elasticity*eel;

  Timer_counter.stop_time("Local Integration");
  Timer_counter.increment_count("in Local Int");
  return ok;
}
```

C++ example

```
void NORTON_ISO_C::derivative(double tau, const VECTOR& chi, VECTOR& dchi)
{
  calc_material((tau-Time_ini)/Dtime );

  attach((VECTOR&)chi);

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

  sig = *elasticity*eel;

  TENSOR2 sprime = deviator(sig);
  TENSOR2 sigeff = sprime;
  double  J      = sqrt(1.5*(sigeff|sigeff));
  double  R      = R0 + Q*(1.-exp(-b*evcum));
  double  f      = J - R;
  TENSOR2 deto(tsz(),dgrad,0);
  TENSOR2 norm(tsz()); norm = 0.;
  TENSOR2  deel(tsz(),dchi,eel.start_pos);
  double& devcum = dchi[evcum.start_pos];
  if (f>0.0) {
     norm     = sigeff*(1.5/J);
     devcum   = pow(f/K,n);
  }  else {     devcum = 0.;     norm=0.; }
  deel = deto - devcum*norm;
}
```

C++ example

```
void NORTON_ISO_C::calc_grad_f(VECTOR& f_vec, SMATRIX& f_grad, const VECTOR& chi_vec,
                               const VECTOR& d_chi,double theta, double dt) {
   attach((VECTOR&)chi_vec, (VECTOR&)d_chi);

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

   TENSOR2  deel(tsz(),((VECTOR&)d_chi),eel.start_pos);
   double& devcum = ((VECTOR&)d_chi)[evcum.start_pos];

   VECTOR   f_vec_eel;
   f_vec_eel.reassign(eel.size(),f_vec,eel.start_pos);
   double&  f_vec_evcum=f_vec[evcum.start_pos];

   SMATRIX  deel_deel;
   MATRIX   deel_devcum, devcum_deel;
   deel_deel.reassign(eel.size(),f_grad,eel.start_pos,eel.start_pos);
   deel_devcum.reassign(eel.size(),1,f_grad,eel.start_pos,evcum.start_pos);
   double&  devcum_devcum=f_grad(evcum.start_pos,evcum.start_pos);
   devcum_deel.reassign(1,eel.size(),f_grad,evcum.start_pos,eel.start_pos);

    double tdt=theta*dt;
   f_grad=0.0; f_grad.add_to_diagonal(1.0);
   f_vec=d_chi;
   ELASTICITY& E=*elasticity;
   double tdv = theta*devcum;
   sig = E*eel;
   double       R   = R0+Q*(1-exp(-b*evcum));
   TENSOR2 sprime = deviator(sig);
   double  J      = sqrt(1.5*(sprime|sprime));
   double  f      = J - R;
```

C++ example

```
    TENSOR4         unit, unit32;   // utility dev operator s' = unit*sigma
    unit.resize(tsz());
    unit=0.0;
    double th=1.0/3.0;
    double th2=2.0/3.0;
    switch (tsz())  {
    case 6:
       unit(4,4)=unit(5,5)=1.0;
    case 4:
       unit(3,3)=1.0;
       unit(1,1)=unit(2,2)=th2;
       unit(1,0)=unit(0,1)=-th;
       unit(2,0)=unit(0,2)=-th;
       unit(2,1)=unit(1,2)=-th;
    case 1:
       unit(0,0)=th2;
    }
    unit32.resize(tsz()); unit32 = 1.5*unit;

    f_vec_eel -= deto;
    if ( (f>0.0 && devcum>=0) || (devcum>0.0) ) {
        TENSOR2        norm            = sprime*(1.5/J);
        f_vec_eel      += norm*devcum;
        f_vec_evcum    -= dt*pow(f/K,n);
        SMATRIX dn_ds = unit32; dn_ds -= norm^norm; dn_ds *= tdv/J;
        SMATRIX dn_deel = dn_ds*E;
        double  dv_df   = tdt*n*pow(f/K,n-1)/K;
        TENSOR2 df_fs   = dv_df*norm;
        deel_deel         += dn_deel;
        deel_devcum        = norm;
        devcum_devcum     += dv_df*Q*b*exp(-b*evcum);
        devcum_deel       -= df_fs*E;
    }
}
```

Plan

1. Behavior generalities
2. ZebFront behaviors
   - Explicite integration
   - Implicit integration
3. **Behavior development in Zébulon**
   - Behavior in C++ language
   - Material object
4. Use of Z-mat with Abaqus

Material object example: How to connect ?

Example for Sellars-tegart flow: $\dot{p} = A\left[sinh\left(\frac{f(\underset{\sim}{\sigma})}{K}\right)\right]^m$

Base class for FLOW and functions to overload

```
class FLOW : public MATERIAL_PIECE {
...
virtual void initialize(ASCII_FILE&,MATERIAL_PIECE* boss);
virtual double      flow_rate(double v, double crit);
virtual double      dflow_dv();
virtual double      dflow_dcrit();
...
};
```

Object factory declaration

```
DECLARE_OBJECT(FLOW,NEW_FLOW,new_flow)
```

Input file

```
***behavior
**potential ...
*flow new_flow
...
****return
```

Material object example: Source code (1/2)

```
#include <Object_factory.h>
#include <File.h>
#include <Flow.h>

class FLOW_SINH : public FLOW {
protected :
double _os;
COEFF A,K,m;
public :
FLOW_SINH() {}
virtual ~FLOW_SINH() {}
void initialize(ASCII_FILE& file, MATERIAL_PIECE*);
double flow_rate(double v, double crit);
double dflow_dv();
double dflow_dcrit();
};
DECLARE_OBJECT(FLOW,FLOW_SINH,sinh)

double FLOW_SINH::flow_rate(double, double crit)
{  _os=crit;
return A*pow(sinh(crit/K),m);
}

double FLOW_SINH::dflow_dcrit()
{ return (m*A/K)*pow( sinh(_os/K), m-1 )*cosh(_os/K); }
```

Material object example: Source code (2/2)

```
double FLOW_SINH::dflow_dv()
{ return 0.; }

void FLOW_SINH::initialize(ASCII_FILE& file, MATERIAL_PIECE* mp)
{ FLOW::initialize(file,mp);
for(;;) {
STRING str=file.getSTRING();
if( str[0]=='*' ) break;
else if(str=="A" ) A.read(str,file,mp);
else if(str=="K" ) K.read(str,file,mp);
else if(str=="m" ) m.read(str,file,mp);
else INPUT_ERROR("Unknown coefficient:"+str);
} file.back();
}
```

Plan

1. Behavior generalities
2. ZebFront behaviors
   - Explicite integration
   - Implicit integration
3. Behavior development in Zébulon
   - Behavior in C++ language
   - Material object
4. **Use of Z-mat with Abaqus**

User interface for Abaqus

- ABAQUS input file: visco.inp

```
******************************
** ABAQUS INPUT FILE:
******************************
*NODE .................
**----------------------------
*SOLID SECTION,ELSET=all,MATERIAL=
 steel
*MATERIAL,NAME=steel
*DEPVAR
  13
*USER MATERIAL,CONSTANTS=1
0.0
**----------------------------
...
```

- Zmat material file: steel

```
***material
 *integration theta_method_a
        1.0 1.e-12 150
 ***behavior  gen_evp
   **elasticity isotropic
        young   210000.0
        poisson  0.33
   **potential gen_evp ep
      *criterion  mises
      *flow  plasticity
      *isotropic constant
         R0 150.0
      *kinematic nonlinear
         D  69.31
         C  8317.77
  ***return
```

Command: $ `Zmat cyclic_abaqus`

Zpreload utility

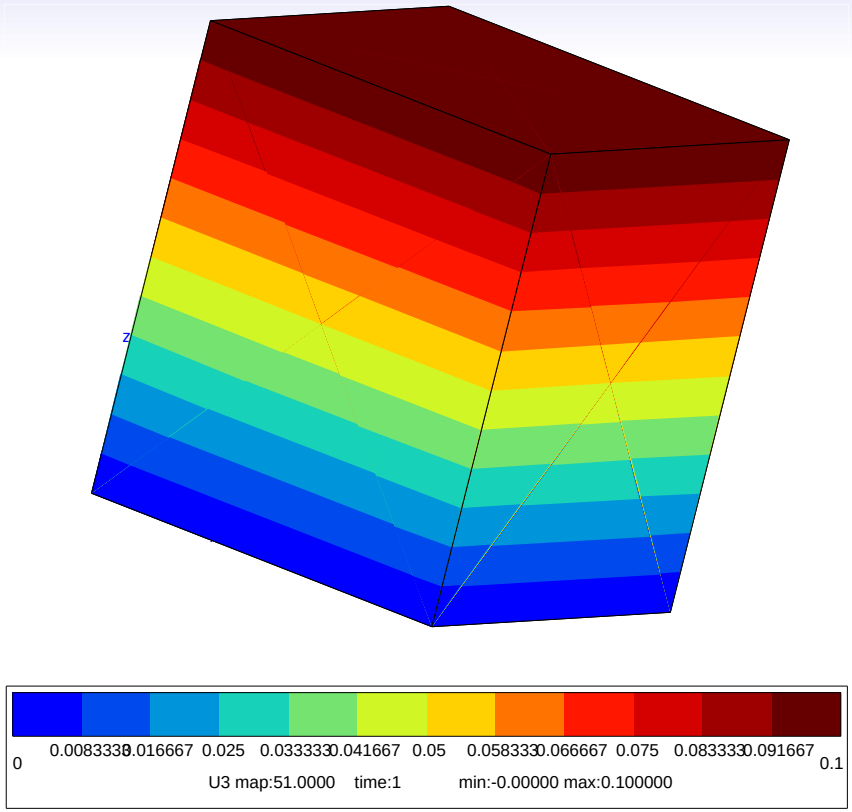
- Checks the behavior definition
- Printouts the names and number of SDVs needed by the Z-mat model
- Command: `$ Zpreload zf`
- 3D default output, for 2D use `Zpreload -d 2 ...`

```
Reading behavior in file: zf
============================================
..
var_int Name:
  eel11(sdv1) eel22(sdv2) eel33(sdv3) eel12(sdv4) eel23(sdv5)
  eel31(sdv6)
  epcum(sdv7)
  al111(sdv8) al122(sdv9) al133(sdv10) al112(sdv11) al123(sdv12)
  al131(sdv13)

var_aux Name:
  epi11(sdv14) epi22(sdv15) epi33(sdv16) epi12(sdv17) epi23(sdv18)
  epi31(sdv19)
============================================
done with material file reading...
```

Command ▸ Zpreload zmat ▸ Zpreload -d 2 zmat

Reading odb files with the Zmaster GUI



- Alternative to abaqus viewer
- Basic graphical post-treatment operations
- Preserve Z-mat's SDV names
- Iso-contours at integration points

Command ▸ Zodb -G cyclic_abaqus.odb

Running Z-post on odb files

- General batch post-processing
- Damage post-processing
- Reading/Writing of odb files

```
****post_processing
 ***data_source odb
  **open cyclic_abaqus.odb
 ***data_output odb
  **problem_name zpost_for_abaqus
 ***local_post_processing
  ...
  **process range
   *var sig
****return
```

Command ▸ Zodb -pp zpost_for_abaqus