# Invariant-based implementation of the Mohr-Coulomb elasto-plastic model in OpenGeoSys using MFront

**Technical Report** · April 2020

DOI: 10.13140/RG.2.2.34335.10403

**4 authors**, including:

Thomas Helfer
Atomic Energy and Alternative Energies Commission
**38** PUBLICATIONS **128** CITATIONS
SEE PROFILE

Thomas Nagel
Technische Universität Bergakademie Freiberg
**101** PUBLICATIONS **655** CITATIONS
SEE PROFILE

# Invariant-based implementation of the Mohr-Coulomb elasto-plastic model in OpenGeoSys using MFront

Gentien Marois, Thomas Nagel, Dmitri Naumov, Thomas Helfer

1/08/2019

This page describes how to implement a non-associated plastic behaviour based on the Mohr-Coulomb criterion. The algorithm mostly follows the work of Nagel et al. (2017) and relies on an apex smoothing introduced by Abbo and Sloan (1995).

This implementation has been introduced in OpenGeoSys.

This document shows:

- how to implement a plastic behaviour based on the third invariants of the stress tensor.
- how to simply the implementation by moving the evoluation of the stress criteria (and its first and second derivatives) in a seperate header file.
- how the implementation finally looks like once introduced in the `StandardElastoViscoplasticity` brick

## 1 Description of the behaviour

The behaviour is described by a standard decomposition of the strain $\underline{\varepsilon}^{\mathrm{to}}$ in an elastic and a plastic component, respectively denoted $\underline{\varepsilon}^{\mathrm{el}}$ and $\underline{\varepsilon}^{\mathrm{p}}$:

$$\underline{\varepsilon}^{\mathrm{to}} = \underline{\varepsilon}^{\mathrm{el}} + \underline{\varepsilon}^{\mathrm{p}}$$

### 1.1 Elastic behaviour

The stress $\underline{\sigma}$ is related to the the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ by a the orthotropic elastic stiffness $\underline{\underline{\mathbf{D}}}$:

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

### 1.2 Yield surface

The plastic part of the behaviour is described by the following yield surface:

$$F = 0$$

F is defined as follow:

$$F = \frac{I_1}{3}\sin\phi + \sqrt{J_2 K(\theta)^2 + a^2\sin^2\phi} - c\cos\phi \tag{1}$$

where

$$K(\theta) = \begin{cases} \cos\theta - \frac{1}{\sqrt{3}}\sin\phi\sin\theta & |\theta| < \theta_{\mathrm{T}} \\ A - B\sin 3\theta & |\theta| \geq \theta_{\mathrm{T}} \end{cases} \tag{2}$$

$$A = \frac{1}{3}\cos\theta_{\mathrm{T}}\left[3 + \tan\theta_{\mathrm{T}}\tan 3\theta_{\mathrm{T}} + \frac{1}{\sqrt{3}}\mathrm{sign}\theta\left(\tan 3\theta_{\mathrm{T}} - 3\tan\theta_{\mathrm{T}}\right)\sin\phi\right]$$
$$B = \frac{1}{3}\frac{1}{\cos 3\theta_{\mathrm{T}}}\left[\mathrm{sign}\theta\,\sin\theta_{\mathrm{T}} + \frac{1}{\sqrt{3}}\sin\phi\cos\theta_{\mathrm{T}}\right] \tag{3}$$

and

$$I_1 = \mathrm{tr}(\underline{\sigma}) \qquad J_2 = \frac{1}{2}\underline{\sigma}^{\mathrm{D}} : \underline{\sigma}^{\mathrm{D}} \qquad J_3 = \det\underline{\sigma}^{\mathrm{D}} \qquad \theta = \frac{1}{3}\arcsin\left(-\frac{3\sqrt{3}J_3}{2\sqrt{J_2^3}}\right)$$

$\underline{\sigma}^{\mathrm{D}}$, $c$ and $\phi$ are respectively the deviatoric part of the tensor $\underline{\sigma}$, the cohesion and friction angle.

The contribution of the smoothing is visualized in Figure 1.

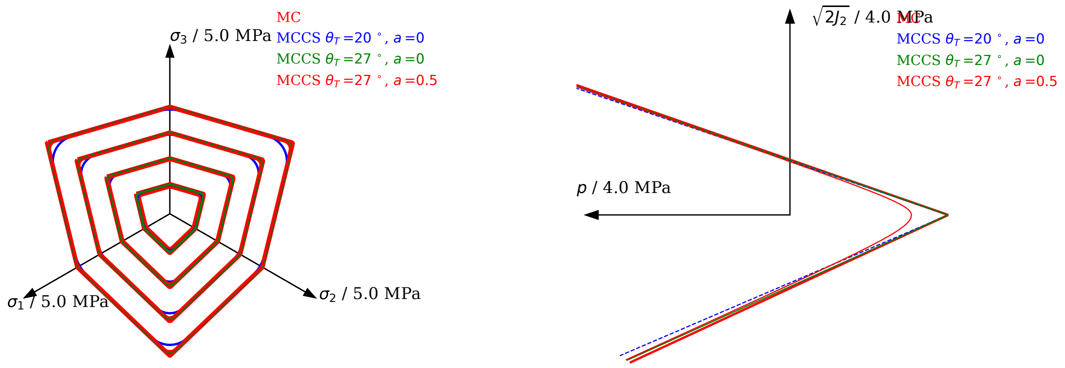


Figure 1: Effect of the smoothing of the yield surface

## 1.3 Plastic potential

The plastic potential differs from the yield surface in order to more accurately estimate dilatancy, but has an analogous structure:

$$G_{\mathrm{F}} = \frac{I_1}{3}\sin\psi + \sqrt{J_2 K_G^2 + a^2\tan^2\phi\cos^2\psi} - c\cos\psi \tag{4}$$

where $\psi$ is the dilatancy angle. $K_G$, $A_G$ and $B_G$ follow from (2) and (3) by substituting the friction angle with the dilatancy angle :

$$K_G(\theta) = \begin{cases} \cos\theta - \frac{1}{\sqrt{3}}\sin\psi\sin\theta & |\theta| < \theta_{\mathrm{T}} \\ A_G - B_G\sin 3\theta & |\theta| \geq \theta_{\mathrm{T}} \end{cases}$$

$$A_G = \frac{1}{3}\cos\theta_{\mathrm{T}}\left[3 + \tan\theta_{\mathrm{T}}\tan 3\theta_{\mathrm{T}} + \frac{1}{\sqrt{3}}\mathrm{sign}\theta\left(\tan 3\theta_{\mathrm{T}} - 3\tan\theta_{\mathrm{T}}\right)\sin\psi\right]$$
$$B_G = \frac{1}{3}\frac{1}{\cos 3\theta_{\mathrm{T}}}\left[\mathrm{sign}\theta\sin\theta_{\mathrm{T}} + \frac{1}{\sqrt{3}}\sin\psi\cos\theta_{\mathrm{T}}\right]$$

Equation ((4)) can be written as follow:

$$G_{\mathrm{F}} = \frac{I_1}{3}\sin\psi + \sqrt{J_2 K_G^2 + a_G^2\sin^2\psi} - c\cos\psi \tag{5}$$

with

$$a_G = a\frac{\tan\phi}{\tan\psi}$$

this formulation allows the use of the `StandardElastoViscoplasticity` brick (see the last part).

## 1.4 Plastic flow rule

Plastic flow follows in a general manner for a $I_1$, $J_2$, $\theta$-type yield surface as

$$\underline{n} = \frac{\partial G_F}{\partial I_1}\underline{I} + \left(\frac{\partial G_F}{\partial J_2} + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_2}\right)\underline{\sigma}^D + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_3} J_3 (\underline{\sigma}^D)^{-1}\ :\ \underline{\underline{\mathbf{P}}}^D$$

or by use of the Caley-Hamilton theorem as

$$\underline{n} = \frac{\partial G_F}{\partial I_1}\underline{I} + \left(\frac{\partial G_F}{\partial J_2} + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_2}\right)\underline{\sigma}^D + \frac{\partial G_F}{\partial \theta}\frac{\partial \theta}{\partial J_3}(\underline{\sigma}^D)^2\ :\ \underline{\underline{\mathbf{P}}}^D$$

# 2 Integration algorithm

The previous constitutive equations will be integrated using a standard implicit scheme.

## 2.1 Plastic loading case

### 2.1.1 Implicit system

Assuming a plastic loading, the system of equations to be solved is:

$$\begin{cases} \Delta\,\underline{\varepsilon}^{\mathrm{el}} - \Delta\,\underline{\varepsilon}^{\mathrm{to}} + \Delta\,p\,\underline{n}\big|_{t+\theta\,\Delta\,t} = 0 \\ F = 0 \end{cases}$$

where $X\big|_{t+\theta\,\Delta\,t}$ is the value of $X$ at $t+\theta\,\Delta\,t$, $\theta$ being a numerical parameter.

In the following, the first (tensorial) equation is noted $f_{\underline{\varepsilon}^{\mathrm{el}}}$ and the second (scalar) equation is noted $f_p$.

In practice, it is physically sound to make satisfy exactly the yield condition at the end of the time step (otherwise, stress extrapolation can lead to stress state outside the yield surface and spurious oscillations can also be observed). This leads to the choice $\theta = 1$.

### 2.1.2 Computation of the jacobian

The jacobian $J$ of the implicit system can be decomposed by blocks:

$$J = \begin{pmatrix} \dfrac{\partial f_{\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial f_{\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\,p} \\ \\ \dfrac{\partial f_p}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial f_p}{\partial \Delta\,p} \end{pmatrix} \tag{6}$$

The expression of the previous terms is given by:

$$
\begin{cases}
\dfrac{\partial f_{\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\, \underline{\varepsilon}^{\mathrm{el}}} = \underline{\underline{\mathbf{I}}} + \Delta\lambda \dfrac{\partial \underline{n}}{\partial \Delta \underline{\epsilon}_{el}} \\
\dfrac{\partial f_{\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\, p} = \underline{n} \\
\dfrac{\partial f_p}{\partial \Delta\, \underline{\varepsilon}^{\mathrm{el}}} = E^{-1} \underline{n}_F \,:\, \underline{\underline{\mathbf{C}}} \\
\dfrac{\partial f_p}{\partial \Delta\, p} = 0
\end{cases}
$$

## 2.2 Elastic loading case

Assuming an elastic loading, the system of equations to be solved is trivially:

$$
\begin{cases}
\Delta\, \underline{\varepsilon}^{\mathrm{el}} - \Delta\, \underline{\varepsilon}^{\mathrm{to}} = 0 \\
\Delta\, p = 0
\end{cases}
$$

The jacobian associated with this system is the identity matrix.

## 3 Implementation

### 3.1 Choice of domain specific language

While not mandatory (the `@DSL` keyword can be place anywhere in the file), its is convenient to start the implementation by declaring the domain specific language to be used. For an integration by a $\theta$-scheme, the `Implicit` domain specific language is choosen:

```
@DSL Implicit;
```

### 3.2 Name of the behaviour

The `@Behaviour` keyword is used to give the name of the behaviour.

```
@Behaviour MohrCoulombAbboSloan;
```

### 3.3 Metadata

The following instructions give some information about the author, the date

```
@Author Thomas Nagel;
@Date 05/02/2019;
```

### 3.4 Name of the Algorithm of resolution

The `@Algorithm` keyword is used to give the name of the algorithm.

```
@Algorithm NewtonRaphson;
```

### 3.5 Usage of the `StandardElasticity` brick

The implicit scheme used satisfies the requirements of the `StandardElasticity` brick as described here.

The `StandardElasticity` brick which provides:

- Automatic computation of the stress tensor at various stages of the behaviour integration.
- Automatic computation of the consistent tangent operator.
- Automatic support for plane stress and generalized plane stress modelling hypotheses (The axial strain is defined as an additional state variable and the associated equation in the implicit system is added to enforce the plane stess condition).
- Automatic addition of the standard terms associated with the elastic strain state variable.

The usage of the `StandardElasticity` is introduced as follows:

```
@Brick StandardElasticity;
```

## 3.6 Numerical parameters

The following instruction changes the default value of the stopping criterion $\epsilon$ used by the Newton-Raphson method and the time integration scheme parameter $\theta$ :

```
@Theta 1.0;
@Epsilon 1.e-14;
```

## 3.7 Supported modelling hypothesis

Thanks to the `StandardElasticity` brick, all the modelling hypotheses can be supported. The following statement, starting with the `@ModellingHypotheses`, enables all the modelling hypotheses:

```
@ModellingHypotheses {".+"};
```

## 3.8 RequireStiffnessTensor

The `@RequireStiffnessTensor` keyword requires the stiffness tensor to be computed by the calling code. This generally means that some extra material properties will be introduced and handled by the interface before the behaviour integration.

```
@RequireStiffnessTensor<UnAltered>;
```

## 3.9 State variable

The `@StateVariable` keyword introduces the EquivalentPlasticStrain $\lambda$.

```
@StateVariable real lam;
lam.setGlossaryName("EquivalentPlasticStrain");
```

## 3.10 Material properties

The `@MaterialProperty` keyword introduces several properties, here:

- The cohesion c
- The friction angle $\phi$
- The dilatancy angle $\psi$
- The transition angle $\theta_T$
- The tension cut-off control parameter a

```
@MaterialProperty stress c;
c.setEntryName("Cohesion");
@MaterialProperty real phi;
phi.setEntryName("FrictionAngle");
@MaterialProperty real psi;
psi.setEntryName("DilatancyAngle");
@MaterialProperty real lodeT;
```

```
lodeT.setEntryName("TransitionAngle");
@MaterialProperty stress a;
a.setEntryName("TensionCutOffParameter");
```

### 3.11 Local variable

In `MFront`, an integration variable is defined to store a variable and use it in various code block.

Here several local variables are declared such as the bolean variable F: if true, plastic loading

```
@LocalVariable Stensor np;
@LocalVariable bool F; // if true, plastic loading
@LocalVariable real sin_psi;
@LocalVariable real sin_phi;
@LocalVariable real cos_phi;
@LocalVariable real cos_lodeT;
@LocalVariable real sin_lodeT;
@LocalVariable real tan_lodeT;
@LocalVariable real cos_3_lodeT;
@LocalVariable real sin_3_lodeT;
@LocalVariable real tan_3_lodeT;
@LocalVariable real a_G;
```

### 3.12 Initialisation a the local variable

The `@InitLocalVariables` code block is called before the behaviour integration.

```
@InitLocalVariables
{
```

First, we define some variables :

```
  constexpr auto sqrt3 = Cste<real>::sqrt3;
  constexpr auto isqrt3 = Cste<real>::isqrt3;
  // conversion to rad
  phi *= pi / 180.;
  psi *= pi / 180.;
  lodeT *= pi / 180.;
  sin_psi = sin(psi);
  cos_phi = cos(phi);
  sin_phi = sin(phi);
  sin_lodeT = sin(lodeT);
  cos_lodeT = cos(lodeT);
  tan_lodeT = tan(lodeT);
  cos_3_lodeT = cos(3. * lodeT);
  sin_3_lodeT = sin(3. * lodeT);
  tan_3_lodeT = tan(3. * lodeT);
  a_G = (a * tan(phi)) / tan(psi)
```

Then the `computeElasticPrediction` method (introducted with the `StandardElasticity` brick) is used to compute $\sigma^{el}$

```
  // elastic prediction
  const auto sig_el = computeElasticPrediction();
```

The three invariant $I_1^{el}$, $J_2^{el}$ and $J_3^{el}$ corresponding to the elastic prediction are calculated:

```
  const auto s_el = deviator(sig_el);
  const auto I1_el = trace(sig_el);
  const auto J2_el = max((s_el | s_el) / 2., local_zero_tolerance);
  const auto J3_el = det(s_el);
```

The $\theta^{el}$ angle is defined:

```
const auto arg = min(max(-3. * sqrt3 * J3_el / (2. * J2_el * sqrt(J2_el)),
                         -1. + local_zero_tolerance),
                     1. - local_zero_tolerance);
const auto lode_el = 1. / 3. * asin(arg);
```

K is initiliazed as $K^{el}$ value:

```
auto K = 0.0;
if (abs(lode_el) < lodeT) {
  K = cos(lode_el) - isqrt3 * sin_phi * sin(lode_el);
} else {
  const auto sign =
      min(max(lode_el / max(abs(lode_el), local_zero_tolerance), -1.), 1.);
  const auto A = 1. / 3. * cos_lodeT *
                 (3. + tan_lodeT * tan_3_lodeT +
                  isqrt3 * sign * (tan_3_lodeT - 3. * tan_lodeT) * sin_phi);
  const auto B = 1 / (3. * cos_3_lodeT) *
                 (sign * sin_lodeT + isqrt3 * sin_phi * cos_lodeT);
  K = A - B * arg;
}
```

To finish $F^{el}$ is calculated and the normal `np` is initialized.

```
const auto sMC =
    I1_el / 3 * sin_phi + sqrt(J2_el * K * K + a * a * sin_phi * sin_phi);
F = sMC - c * cos_phi > 0.;
np = Stensor(real(0));
```

## 3.13 Implicit system implementation

The implementation of the implicit system and its derivative is done in the `@Integrator` code block:

```
@Integrator{
```

Some expressions are defined

```
constexpr auto sqrt3 = Cste<real>::sqrt3;
constexpr auto isqrt3 = Cste<real>::isqrt3;
constexpr auto id = Stensor::Id();
constexpr auto id4 = Stensor4::Id();
```

If there is no plasticity (elastic strain) there is no need for additional calculation because the various variables have already been initialized with elastic hypothesis. If there is plastic strain the rest is necessary.

```
if (F) {
```

$K$ and $\dfrac{\partial K}{\partial \theta}$ are computed:

```
    const auto s = deviator(sig);
    const auto I1 = trace(sig);
    const auto J2 = max((s | s) / 2., local_zero_tolerance);
    const auto J3 = real(det(s) < 0. ? min(det(s), -local_zero_tolerance)
                                     : max(det(s), local_zero_tolerance));
    const auto arg = min(max(-3. * sqrt3 * J3 / (2. * J2 * sqrt(J2)),
                             -1. + local_zero_tolerance),
                         1. - local_zero_tolerance);
    const auto lode = 1. / 3. * asin(arg);
    const auto cos_lode = cos(lode);
    const auto sin_lode = sin(lode);
    const auto cos_3_lode = cos(3. * lode);
    const auto sin_3_lode = arg;
    const auto tan_3_lode = tan(3. * lode);
    auto K = 0.;
    auto dK_dlode = 1.;
    if (abs(lode) < lodeT) {
      K = cos_lode - isqrt3 * sin_phi * sin_lode;
      dK_dlode = -sin_lode - isqrt3 * sin_phi * cos_lode;
    } else {
      const auto sign =
          min(max(lode / max(abs(lode), local_zero_tolerance), -1.), 1.);
      const auto A = 1. / 3. * cos_lodeT *
                     (3. + tan_lodeT * tan_3_lodeT +
                      isqrt3 * sign * (tan_3_lodeT - 3. * tan_lodeT) * sin_phi);
      const auto B = 1. / (3. * cos_3_lodeT) *
                     (sign * sin_lodeT + isqrt3 * sin_phi * cos_lodeT);
      K = A - B * sin_3_lode;
      dK_dlode = -3. * B * cos_3_lode;
    }
```

$K_G$, $\dfrac{\partial K_G}{\partial \theta}$ and $\dfrac{\partial^2 K_G}{\partial^2 \theta}$ are computed :

```
    auto KG = 0.0; // move into a function to avoid code duplication
    auto dKG_dlode = 1.;
    auto dKG_ddlode = 1.;
    if (abs(lode) < lodeT)
    {
      KG = cos_lode - isqrt3 * sin_psi * sin_lode;
      dKG_dlode = -sin_lode - isqrt3 * sin_psi * cos_lode;
      dKG_ddlode = -cos_lode + isqrt3 * sin_psi * sin_lode;
    }
    else
    {
      const auto sign =
          min(max(lode / max(abs(lode), local_zero_tolerance), -1.), 1.);
      const auto A = 1. / 3. * cos_lodeT *
                     (3. + tan_lodeT * tan_3_lodeT +
                      isqrt3 * sign * (tan_3_lodeT - 3. * tan_lodeT) * sin_psi);
      const auto B = 1. / (3. * cos_3_lodeT) *
                     (sign * sin_lodeT + isqrt3 * sin_psi * cos_lodeT);
      KG = A - B * sin_3_lode;
      dKG_dlode = -3. * B * cos_3_lode;
      dKG_ddlode = 9. * B * sin_3_lode;
    }
```

The flow direction is computed :

```
    // flow direction
    const auto dev_s_squared = computeJ3Derivative(
        sig); // replaces dev_s_squared = deviator(square(s));
    const auto dG_dI1 = sin_psi / 3.;
    const auto root = max(sqrt(J2 * KG * KG + a_G * a_G * sin_psi * sin_psi),
                          local_zero_tolerance);
    const auto dG_dJ2 = KG / (2. * root) * (KG - tan_3_lode * dKG_dlode);
    const auto dG_dJ3 = J2 * KG * tan_3_lode / (3. * J3 * root) * dKG_dlode;
    const auto n = eval(dG_dI1 * id + dG_dJ2 * s + dG_dJ3 * dev_s_squared);
```

The yield function F is computed:

```
    // yield function
    const auto rootF = max(sqrt(J2 * K * K + a * a * sin_phi * sin_phi), local_zero_tolerance);
    const auto Fy1 = I1 * sin_phi / 3 + rootF;
    const auto Fy =  Fy1 - c * cos_phi;
```

Derivatives are calculated before computing the Jacobian matrix:

$\dfrac{\partial F}{\partial I_1}, \dfrac{\partial F}{\partial J_2}, \dfrac{\partial F}{\partial J_3}, n_F, \dfrac{\partial G}{\partial \theta}, \dfrac{\partial^2 G}{\partial^2 \theta}, \dfrac{\partial^2 G}{\partial \theta \partial J_2}, \dfrac{\partial^2 G}{\partial^2 J_2}, \dfrac{\partial^2 G}{\partial^2 J_3}, \dfrac{\partial^2 G}{\partial J_2 \partial J_3}$

```
    // yield function derivative for Jacobian
    const auto dF_dI1 = sin_phi / 3.;
    const auto dF_dJ2 = K / (2. * rootF) * (K - tan_3_lode * dK_dlode);
    const auto dF_dJ3 = J2 * K * tan_3_lode / (3. * J3 * rootF) * dK_dlode;
    const auto nF = eval(dF_dI1 * id + dF_dJ2 * s + dF_dJ3 * dev_s_squared);

    // building dfeel_ddeel
    const auto Pdev = id4 - (id ^ id) / 3;

    const auto dG_dlode = KG * J2 / (root)*dKG_dlode;
    const auto dG_ddlode =
        J2 / root * (dKG_dlode * dKG_dlode * (1. - J2 * KG * KG / (root * root)) + KG * dKG_ddlode);
    const auto dG_ddlodeJ2 = KG / root * dKG_dlode * (1. - J2 * KG * KG / (2 * root * root));
    const auto dG_ddJ2 =
        -KG * KG * KG * KG / (4. * root * root * root) + dG_dlode * tan_3_lode / (2 * J2 * J2) -
        tan_3_lode / (2 * J2) * (2 * dG_ddlodeJ2 - tan_3_lode / (2 * J2) * dG_ddlode -
                                 3 / (2 * J2 * cos_3_lode * cos_3_lode) * dG_dlode);
    const auto dG_ddJ3 = -tan_3_lode / (3 * J3 * J3) * dG_dlode +
                         tan_3_lode / (3 * J3) * (dG_ddlode * tan_3_lode / (3 * J3) +
                                                  dG_dlode * 1. / (J3 * cos_3_lode * cos_3_lode));
    const auto dG_ddJ2J3 = dG_ddlodeJ2 * tan_3_lode / (3 * J3) -
                           tan_3_lode / (2 * J2) * (dG_ddlode * tan_3_lode / (3 * J3) +
                                                    dG_dlode * 1. / (J3 * cos_3_lode * cos_3_lode));
```

$f^{el}, \dfrac{\partial f^{el}}{\partial \Delta\, \underline{\varepsilon}^{\text{el}}}, \dfrac{\partial f^{el}}{\partial \Delta\, \lambda}$ are computed:

```
    // elasticity
    feel += dlam * n;
    dfeel_ddeel += theta * dlam * (dG_dJ2 * Pdev + dG_dJ3 * computeJ3SecondDerivative(sig) +
                                   dG_ddJ2 * (s ^ s) + dG_ddJ3 * (dev_s_squared ^ dev_s_squared) +
                                   dG_ddJ2J3 * ((dev_s_squared ^ s) + (s ^ dev_s_squared))) *
                   D;
    dfeel_ddlam = n;
```

These equations are equivalent to:

$$
\begin{aligned}
& f^{el} = \Delta\, \underline{\varepsilon}^{\text{el}} - \Delta\, \underline{\varepsilon}^{\text{to}} \ + \Delta\, \lambda\, \underline{n} = 0 \\
& \frac{\partial f^{el}}{\partial \Delta\, \underline{\varepsilon}^{\text{el}}} = 1 + \Delta\lambda \frac{\partial \underline{n}}{\partial \Delta \underline{\varepsilon}^{\text{el}}} = 1 + \Delta\lambda \frac{\partial \underline{n}}{\partial \underline{\sigma}} \frac{\partial \underline{\sigma}}{\partial \Delta \underline{\varepsilon}^{\text{el}}} \\
& \frac{\partial f^{el}}{\partial \Delta\, \lambda} = \underline{n}
\end{aligned}
$$

because this implementation takes into account the fact that the `Implicit` DSL automatically initializes `feel` to the current estimation of $\Delta\, \underline{\varepsilon}^{\text{el}}$ and the Jacobian to identity. Moreover, the `StandardElasticity` brick automatically subtracts $\Delta\, \underline{\varepsilon}^{\text{to}}$ to `feel`.

And to finish $f^{\lambda}, \dfrac{\partial f^{\lambda}}{\partial \Delta\, \lambda}, \dfrac{\partial f^{\lambda}}{\partial \Delta\, \underline{\varepsilon}^{\text{el}}}$ are computed:

```
    // plasticity
    flam = Fy / D(0, 0);
    dflam_ddlam = strain(0);
    dflam_ddeel = theta * (nF | D) / D(0, 0);
    np = n;
```

The Jacobian can now be computed (see (6))

## 4 Verification

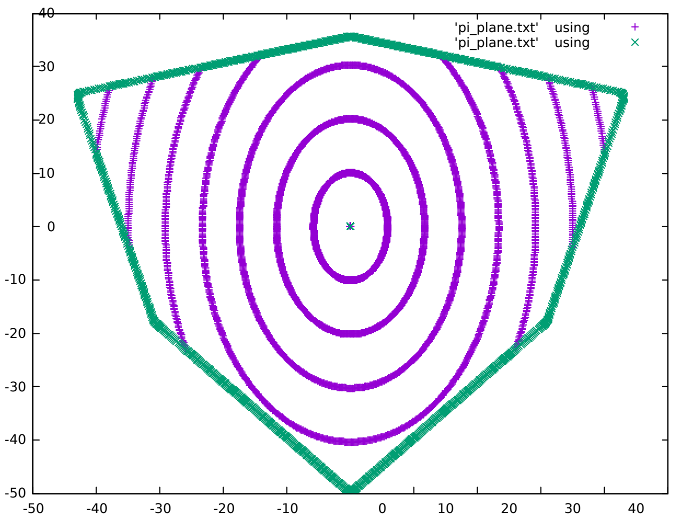
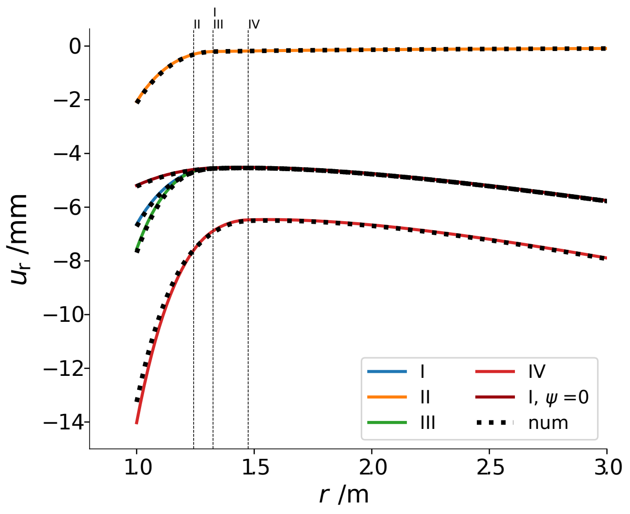


Figure 2: Trace of the yield surface (green) when approached on different stress paths (purple)

The yield function was approached along different stress paths, shown in Figure 2 within the $\pi$-plane. This shows that a) yield is correctly detected, and b) the stress-state is correctly pulled back onto the yield surface.



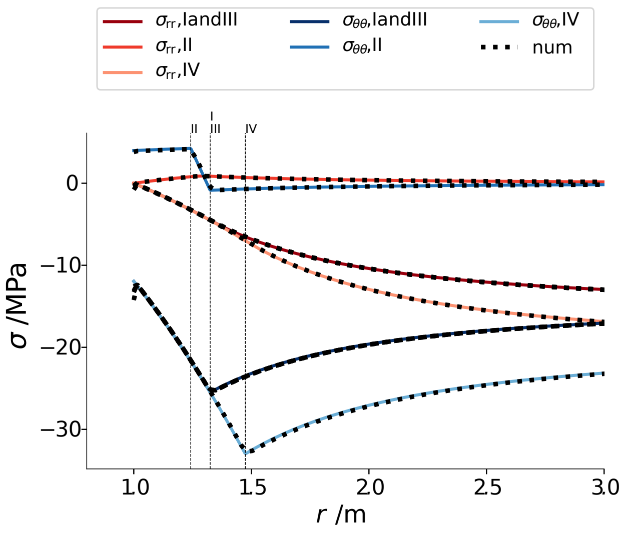


Figure 3: Verification against analytical example. Description in Nagel et al. (2017).

The second test (included as a benchmark) follows an analytical solution for a stress-free cavity in an infinite medium under a variable far-field stress. The solution computes stress- and displacement fields as well as the location of the plastified zone. It has been used in Nagel et al. (2017) where the test is described in more detail (with a partial extension towards non-associated flow).

## 5 Simplification of the MFront file : use of `TFEL/Material/MohrCoulombYieldCrite` file

Because F et G have an analogous structure, it's possible to simplify the MFront file and to do the calculation in the hxx `TFEL/Material/MohrCoulombYieldCriterion.hxx` instead of the MFront file.

Severals simplifications are done:

- parameters and local variables such as `sin_phi`, `sin_psi`, ... are now defined in the hxx file
- the calculation of $F^{el}$ is done by `computeMohrCoulombStressCriterion` function
- the calculation of $F$ and this normal $\underline{n}_F$ are done by `computeMohrCoulombStressCriterionNormal` function
- the calculation of $G$, $\underline{n}_G$ and $\dfrac{\partial \underline{n}_G}{\partial \underline{\sigma}}$ are done by `computeMohrCoulombStressCriterionSecondDerivative` function

Except for some name changes (for example p instead lam for the EquivalentPlasticStrain) and the functions previously introduced (`computeMohrCoulombStressCriterion`,`computeMohrCoulombStressCriterionNormal` and `computeMohrCoulombStressCriterionSecondDerivative`) the rest of the MFront file is identical to that described above in the section `Implementation`).

The new MFront file is however much shorter and clearer:

```
@DSL Implicit;
@Behaviour MohrCoulombAbboSloan;
@Author Thomas Nagel;
@Date 05 / 02 / 2019;

@Algorithm NewtonRaphson;

@Brick StandardElasticity;

@Includes{
#include "TFEL/Material/MohrCoulombYieldCriterion.hxx"
}

@Theta 1.0;     // time integration scheme
@Epsilon 1e-14; // tolerance of local stress integration algorithm
@ModellingHypotheses{".+"};

@RequireStiffnessTensor<UnAltered>;
@Parameter pi = 3.14159265359;

@StateVariable real p;
p.setGlossaryName("EquivalentPlasticStrain");

@MaterialProperty stress c;
c.setEntryName("Cohesion");
@MaterialProperty real phi;
phi.setEntryName("FrictionAngle");
@MaterialProperty real psi;
psi.setEntryName("DilatancyAngle");
@MaterialProperty real lodeT;
lodeT.setEntryName("TransitionAngle");
@MaterialProperty stress a;
a.setEntryName("TensionCutOffParameter");

@LocalVariable Stensor ngp;

@LocalVariable bool F; // if true, plastic loading
@LocalVariable MohrCoulombParameters<StressStensor> pf;
@LocalVariable MohrCoulombParameters<StressStensor> pg;
```

```
@LocalVariable real a_G;

@InitLocalVariables {
  a_G = (a * tan((phi*pi)/180)) / tan((psi*pi)/180);
  constexpr const auto u = MohrCoulombParameters<StressStensor>::DEGREE;
  pf = makeMohrCoulombParameters<StressStensor, u>(c, phi, lodeT, a);
  pg = makeMohrCoulombParameters<StressStensor, u>(c, psi, lodeT, a_G);
  const auto sel = computeElasticPrediction();
  const auto smc = computeMohrCoulombStressCriterion(pf, sel);
  F = smc > stress(0);
  ngp = Stensor(real(0));
}

@Integrator {
  if (F) {
    // in C++11:
    auto Fy = stress{};
    auto nf = Stensor{};
    auto Fg = stress{};
    auto ng = Stensor{};
    auto dng = Stensor4{};
    std::tie(Fy, nf) = computeMohrCoulombStressCriterionNormal(pf, sig);
    std::tie(Fg, ng, dng) = computeMohrCoulombStressCriterionSecondDerivative(pg, sig);
    // elasticity
    feel += dp * ng;
    dfeel_ddeel += theta * dp * dng * D;
    dfeel_ddp = ng;
    // plasticity
    fp = Fy / D(0, 0);
    dfp_ddp = strain(0);
    dfp_ddeel = theta * (nf | D) / D(0, 0);
    ngp = ng;
  }
}
```

## 6 Simplification of the MFront file : use of the `StandardElastoViscoPlasticity` brick

The Mohr-Coulomb model has been introduced in the `StandardElastoViscoPlasticity` brick.

The `MFront` file is then very short: the whole model is contained in the brick and can be called by the keyword `MohrCoulomb`

The MFront file is now:

```
@DSL Implicit;

@Behaviour MohrCoulomAbboSloan3;
@Author HELFER Thomas 202608;
@Date 10 / 12 / 2019;
@Description {
}

@Algorithm NewtonRaphson;
@Epsilon 1.e-14;
@Theta 1;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
    young_modulus : 150.e3,
    poisson_ratio : 0.3
  },
  inelastic_flow : "Plastic" {
    criterion : "MohrCoulomb" {
      c : 3.e1,      // cohesion
      phi : 0.523598775598299,    // friction angle or dilatancy angle
      lodeT : 0.506145483078356,  // transition angle as defined by Abbo and Sloan
      a : 1e1       // tension cuff-off parameter
    },
    flow_criterion : "MohrCoulomb" {
      c : 3.e1,      // cohesion
      phi : 0.174532925199433,    // friction angle or dilatancy angle
      lodeT : 0.506145483078356,  // transition angle as defined by Abbo and Sloan
      a : 3e1       // tension cuff-off parameter
    },
    isotropic_hardening : "Linear" {R0 : "0"}
  }
};
```

## 7 References

Abbo, A.J., and S.W. Sloan. 1995. "A smooth hyperbolic approximation to the Mohr-Coulomb yield criterion." *Computers & Structures* 54 (3): 427–41. https://doi.org/10.1016/0045-7949(94)00339-5.

Nagel, Thomas, Wolfgang Minkley, Norbert Böttcher, Dmitri Naumov, Uwe-Jens Görke, and Olaf Kolditz. 2017. "Implicit numerical integration and consistent linearization of inelastic constitutive models of rock salt." *Computers & Structures* 182 (April): 87–103. https://doi.org/10.1016/j.compstruc.2016.11.010.