incrementalDriver
programmer's manual

A.Niemunis

March 27, 2020

1 Introduction

incrementalDriver is a program for testing constitutive models. incrementalDriver enables *element* tests. It deals with homogeneous fields only (i.e. no consolidation, no spatial changes of stress, strain and state variables, no spatial change of material constants, no gradients of deformation are allowed for). incrementalDriver calls a material routine (constitutive relations) with the syntax of the user material subroutine of Abaqus^{TM} [1]. Given stress $\mathbf{T}$, with other internal state variables $\boldsymbol{\alpha}$ (all at the beginning of an increment) and strain increment $\Delta\boldsymbol{\epsilon}$, `umat` updates $\mathbf{T}$ and $\boldsymbol{\alpha}$ returning their values at the end of the increment. Moreover `umat` calculates a tangential stiffness matrix or a Jacobian matrix $\mathsf{E} = \partial\Delta\mathbf{T}/\partial\Delta\boldsymbol{\epsilon}$. The main task of incrementalDriver is to force `umat` to follow a prescribed loading path formulated in strain or stress components. The complementary components are calculated and stored (in an output file) together with the constitutive state variables. The main difficulty is to follow a prescribed stress path or a combined stress/strain path (= loading). The problem arises from the fact that `umat` accepts only *strain increment* $\Delta\boldsymbol{\epsilon}$ as input. For a prescribed stress increment $\breve{\Delta}\mathbf{T}$ the components of $\Delta\boldsymbol{\epsilon}$ will be determined iteratively in a procedure similar to the equilibrium iteration (EI) in the FEM.

In order to use incrementalDriver you may omit $\star$ sections. They provide additional information for programmers.

2 Notation

Matrix notation and a fixed orthogonal Cartesian coordinate system is used. The components of second rank tensors are written as 6×1 column matrix, in particular

$$\begin{aligned} \mathbf{1} &= [1,1,1,0,0,0]^T\,, \qquad \mathbf{0} = [0,0,0,0,0,0]^T \\ \Delta\mathbf{T} &= [\Delta T_{11}, \Delta T_{22}, \Delta T_{33}, \Delta T_{12}, \Delta T_{13}, \Delta T_{23}]^T \\ \Delta\boldsymbol{\epsilon} &= [\Delta\epsilon_{11}, \Delta\epsilon_{22}, \Delta\epsilon_{33}, \Delta\gamma_{12}, \Delta\gamma_{13}, \Delta\gamma_{23}]^T \end{aligned} \qquad \mathsf{J} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

The accent $\breve{\sqcup}$, reads "prescribed", e.g. $\breve{\Delta}T_i$ is a prescribed component of stress and $\Delta\epsilon_i$ is an unknown one of strain.
The upper index $\sqcup^i$ at a boldface variable denotes its value after the step number i.
The lower index $\sqcup_k$ at a boldface variable denotes its value after th equilibrium iteration step number k.

3 $\star$ Flowchart

The following description of the algorithm of incrementalDriver is slightly simplified. It leaves aside transformed variables and rigid rotations of the material. The flowchart is organized as follows:

1. Choose a `umat`.
2. Read the material constants

3. Read an initial state = initial stress $\mathbf{T}^0$, and an initial deformation $\boldsymbol{\epsilon}^0$. Usually, $\boldsymbol{\epsilon}^0 = \mathbf{0}$ is the starting point of the strain path. Internal state variables $\boldsymbol{\alpha}^0$ must be initialized too. Print this initial state to the output file.

4. Call `umat` with a zero strain increment[1] $\Delta\boldsymbol{\epsilon} = \mathbf{0}$ to get an estimate of the tangential stiffness E_0. In the case of a stress or mixed control E_0 will be needed to make the first guess about the strain increment $\Delta\boldsymbol{\epsilon}$. The lower index is the number of equilibrium iteration (EI).

5. Start an loop over load increments (values *after* the increment are denoted by the number of increment written as the upper index, e.g. after first increment we have $\mathbf{T}^1 = \mathbf{T}^0 + \Delta\mathbf{T}^1$. The upper index (=number of increment) is often omitted if the current increment is meant.

6. Read an increment from the prescribed stress/strain path:

   - Mixed load increments may be prescribed, i.e. some components $\breve{\Delta}\epsilon_{ij}$ of the strain increment and the complementary components $\breve{\Delta}T_{kl}$ (with $ij \neq kl$) of the stress increment are given.
   - Transformed components may be used in prescribing the load increments. This means that linear combinations of strain components like $\breve{\Delta\epsilon}_v$ with $\epsilon_v = -\mathrm{tr}\,\boldsymbol{\epsilon}$ or linear combinations of stress components like $\breve{\Delta}p$ with $p = -\mathrm{tr}\,\mathbf{T}$ may be prescribed. Very useful are increments of Roscoe's invariants instead of the conventional cartesian components.

7. Start the "equilibrium" iteration (EI) within an increment. EI is necessary unless all load components are deformations[2]. The corrections $\mathbf{c}_T$ are added to stress increments, e.g. $\Delta\mathbf{T}^i_k = \Delta\mathbf{T}^i_{k-1} + \mathbf{c}_T$ so that the prescribed stress increments can be better approximated.

8. Make the first guess of the strain increment $\Delta\boldsymbol{\epsilon}_1$ using the initial jacobian E_0, i.e., solve $\Delta\mathbf{T}_1 = \mathsf{E}.\Delta\boldsymbol{\epsilon}_1$ with some unknown components in $\Delta\boldsymbol{\epsilon}_1$ and some in $\Delta\mathbf{T}_1$. The lower index k (at bold face symbol, say $\mathbf{T}^i_k$) tells the number of EI and the upper index i tells the number of increment in step. If the upper index is omitted then the quantity pertains to the current increment.

9. Call `umat` with $\Delta\boldsymbol{\epsilon}_1$ to get the approximated stress $\Delta\mathbf{T}$ and a better estimation of the tangential stiffness E_1

10. For prescribed stress components $\breve{\Delta}T_{ij}$ (`ddstress`[3]the approximation ΔT_{ij} (`a_dstress` [4] calculated with the current guess of the strain increment $\Delta\boldsymbol{\epsilon}$ may have an error $u_i = \breve{\Delta}T_i - \Delta T_i$ (`u_dstress`[5]. This error corresponds to the "out of balance forces" in a true EI of a FE program and must be reduced (iteratively). We can have two cases:

$\|\mathbf{u}\| >$ toler: EI is not finished. The error is considerable, so it must be reduced. For this purpose

   (a) Calculate the corrected strain components where the stress components are prescribed using

$$\Delta\boldsymbol{\epsilon}_2 = \Delta\boldsymbol{\epsilon}_1 + \mathbf{c}_\epsilon \quad \text{with} \quad \mathbf{c}_\epsilon = (\mathsf{E}_1)^{-1} : \mathbf{u} \tag{1}$$

   We must not modify the prescribed strain components. Hence, the above $\Delta\boldsymbol{\epsilon}_2$ must be overridden by setting $\Delta\boldsymbol{\epsilon}_2 = \breve{\Delta}\boldsymbol{\epsilon}$ wherever the strain control applies[6].

   (b) Undo all updates (of stress, strain and state variables) performed by `umat`. For this purpose the stress and the internal state variables from the beginning of the current increment must be memorized. INCREMENTALDRIVER stores them as `r_stress` and `r_statev`[7], respectively.

   (c) Repeat the EI i.e. call `umat` with the improved $\Delta\boldsymbol{\epsilon}_2$ and with $\mathbf{T}$ and $\boldsymbol{\alpha}$ from the beginning of the increment.

$\|\mathbf{u}\| <$ toler : EI is finished. The error $\|\mathbf{u}\|$ is acceptable

   (a) Accept the updates done by `umat` (= do nothing).

   (b) Write out everything that matters to the output file (to be plotted by another programs).

   (c) Update the total strain, the time and continue with the next increment.

[1] If your `umat` crashes when called with $\Delta\boldsymbol{\epsilon} = \mathbf{0}$ and $\Delta t = 0$ it will crash in Abaqus too.

[2] No iterative process (no EI) is needed for fully strain-controlled loading (all components of $\Delta\boldsymbol{\epsilon}$ are prescribed), because `umat` itself can calculate the exact stress increments as the material response. If stress increments are prescribed as loads then strain increments must be found iteratively by making a guess of $\Delta\boldsymbol{\epsilon}$, calculating the corresponding increment $\Delta\mathbf{T}$, finding the desired stress correction, finding (linearly) the corresponding correction in strain etc.

[3] for 'desired delta stress'

[4] for 'approximated delta stress'

[5] for 'undesired delta stress'

[6] The corrections $\mathbf{c}_\epsilon$ are called `c_dstran` in the program

[7] `r_` stands for 'remembered before increment'

4 ⋆ How artificial deformation cycles are avoided

Undoing of inaccurate approximations $\Delta\mathbf{T}$ and $\Delta\boldsymbol{\alpha}$ during EI means that `umat` is repeatedly given the same initial stress and the same initial state in all iteration. Only strain increments are modified (updated according to $\Delta\boldsymbol{\epsilon}_2 = \Delta\boldsymbol{\epsilon}_1 + \mathbf{c}_\epsilon$ where $\mathbf{c}_\epsilon = (\mathsf{E}_1)^{-1} : \mathbf{u}$) during the EI.

Such approach prevents an artificial (numerical) zigzag-like evolution of $\mathbf{T}$ and $\boldsymbol{\alpha}$ in the EI. If stress and state increments done in `umat` were kept by INCREMENTALDRIVER after each EI, then the subsequent contributions $\mathbf{c}_\mathbf{T}$ and $\mathbf{c}_{\boldsymbol{\alpha}}$ were successively added to the total values of $\mathbf{T}$ and $\boldsymbol{\alpha}$. Such process is not exact because it is not physical. The sequence of updates (with possible oscillations, unloadings, reloadings, zigzag trajectories) would be dictated by the convergence procedure of EI rather than by physical phenomena. "Numerical" oscillations of state variables might influence the accuracy of the constitutive response and therefore should be avoided.

5 Rigid rotation

Symbolic and index notation of tensors are used in this section (not matrix notation)

The strain increments should be calculated from the Hencky strain with respect to the material axes

$$\boldsymbol{\epsilon} = \ln \mathbf{U}, \tag{2}$$

wherein $\mathbf{U}$ is the right stretch tensor from the polar decomposition $\mathbf{F} = \mathbf{R} \cdot \mathbf{U}$.

In a single `*LinearLoad` step we apply the total strain change $\boldsymbol{\epsilon}$ which is decomposed into `ninc` equal increments $\Delta\boldsymbol{\epsilon}$. In a 1D oedometric compression from the height H_0 to H_n we would have

$$\epsilon = \ln \frac{H_n}{H_0} = \sum_{i=1}^{n} \ln \frac{H_i}{H_{i-1}} \tag{3}$$

so it is evident that equal strain increments $\ln(H_i/H_{i-1}) = \text{const}$ do not imply equal increments of displacement, i.e. $H_i - H_{i-1} \neq \text{const}$.

In some tests (for example simple shearing) we may wish to define a deformation path using the gradient of deformation $\mathbf{F}$ rather than the strain $\boldsymbol{\epsilon}$ per step. For this purpose after the keyword `*DeformationGradient` all nine partial derivatives $\breve{\mathbf{F}} = (\partial\mathbf{x}(t_E)/\partial\mathbf{x}(t_B))$ of the position vector at the end (=time t_E) of the step with respect to the position vector at the beginning (=time t_B) of the step must be specified in the following sequence

$$\left\{\breve{F}_{11}, \breve{F}_{22}, \breve{F}_{33}, \breve{F}_{12}, \breve{F}_{21}, \breve{F}_{13}, \breve{F}_{31}, \breve{F}_{23}, \breve{F}_{32}\right\} \tag{4}$$

Note that $\mathbf{F} = \mathbf{1}$ at the beginning (=time t_B) of the step. The *displacement* gradient $\breve{\mathbf{F}} - \mathbf{1}$ is decomposed *additively* into $n =$ `ninc` equal (and sufficiently small) increments

$$\breve{\Delta}\mathbf{F} = \frac{1}{n}(\breve{\mathbf{F}} - \mathbf{1}) \tag{5}$$

This decomposition corresponds to $H_i - H_{i-1} = \text{const}$ in the 1D example discussed above so this decomposition is slightly different than the one of `*LinearLoad`.

Let us denote $\Delta\mathbf{F} = \mathbf{F}_E - \mathbf{F}_B = \mathbf{F}_E - \mathbf{1}$ the difference of deformation gradients at the beginning ($\mathbf{F}_B$) and at the end ($\mathbf{F}_E$) of the current increment. After Hughes and Winget [3] we calculate the following incremental approximations

$$\Delta\mathbf{L} = \Delta\mathbf{F} \cdot \left[\mathbf{F}_B + \frac{1}{2}\Delta\mathbf{F}\right]^{-1} \quad \text{and} \tag{6}$$

$$\mathbf{D}\Delta t = \Delta\boldsymbol{\epsilon} = \frac{1}{2}\left(\Delta\mathbf{L} + \Delta\mathbf{L}^T\right) \tag{7}$$

$$\mathbf{W}\Delta t = \frac{1}{2}\left(\Delta\mathbf{L} - \Delta\mathbf{L}^T\right) \tag{8}$$

$$\Delta\mathbf{R} = \left(\mathbf{1} - \frac{1}{2}\mathbf{W}\Delta t\right)^{-1} \cdot \left(\mathbf{1} + \frac{1}{2}\mathbf{W}\Delta t\right) \tag{9}$$

The stress is updated using the equation

$$\mathbf{T}^{i+1} = \mathbf{T}^i + \Delta\mathbf{R} \cdot (\mathbf{T}^i + \Delta\mathbf{T}) \cdot \Delta\mathbf{R}^T \tag{10}$$

in which $\Delta\mathbf{T}$ is the constitutive (co-rotational) stress increment calculated with $\Delta\boldsymbol{\epsilon}$ in the subroutine `umat`. Analogously we rotate the total strain tensor

$$\boldsymbol{\epsilon}^{i+1} = \boldsymbol{\epsilon}^i + \Delta\mathbf{R} \cdot \Delta\boldsymbol{\epsilon} \cdot \Delta\mathbf{R}^T \tag{11}$$

`Is the the original form better ?`

$$\mathbf{T}^{i+1} = \mathbf{T}^i + \Delta\mathbf{R} \cdot \mathbf{T}^i \cdot \Delta\mathbf{R}^T + \Delta\mathbf{T} \tag{12}$$

After Rashid [5] the algorithm by Hughes and Winget [3] is only weakly objective (objective only for the special cases of pure rotation or pure stretching but not for their combinations). For two input deformation increments that differ only by a rotation a strongly objective stress update algorithm should guarantee that the output stress also differ only by a rotation. Weak objectivity may cause artificial cumulative effects during cyclic shearing.

5.1 ⋆ Material or spatial strain

Abaqus passes to `umat` the information about the total strain in rotated configuration, i.e. first rotated then stretched

$$\boldsymbol{\epsilon} = \ln \mathbf{V}, \tag{13}$$

and not in the material configuration $\boldsymbol{\epsilon} = \ln \mathbf{U}$ (first stretched and then rotated). For most geotechnical models it should not be of importance because strain should not be treated as a state variable (the initial value indeterminable). In some cases, however, we do need $\boldsymbol{\epsilon} = \ln \mathbf{U}$, for example in order to evaluate a multiaxial strain amplitude during out-of-phase cycles.

The definition of $\boldsymbol{\epsilon} = \ln \mathbf{V}$ makes it impossible to recognize rigid rotation from the deformation with the principal axes rotation. The following example illustrates the problem. Suppose, the material was first uniaxially stretched with $\mathbf{F} = \mathbf{U} = \text{ diag}(\lambda, 1, 1) = \text{const}$ (uniformly with $\lambda \neq 1$). Let us observe $\mathbf{V}$ during a subsequent rigid rotated with $\mathbf{R}(t)$. Using a time-like parameter t and the abbreviations $c = \cos(\dot{\omega}t), s = \sin(\dot{\omega}t)$ we may consider a rigid rotation about the x_3 axis:

$$\mathbf{R} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{and hence} \qquad \mathbf{F} = \mathbf{R} \cdot \mathbf{U} = \begin{bmatrix} c\lambda & -s & 0 \\ s\lambda & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{14}$$

The left stretch tensor obtained from $\mathbf{F} = \mathbf{V} \cdot \mathbf{R}$ as $\mathbf{V} = \mathbf{F} \cdot \mathbf{R}^T = \mathbf{R} \cdot \mathbf{U} \cdot \mathbf{R}^T$ in the matrix form

$$\mathbf{V} = \begin{bmatrix} c^2\lambda + s^2 & -cs + cs\lambda & 0 \\ -cs + cs\lambda & c^2 + \lambda s^2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

Despite the rigid rotation the components of $\mathbf{V}$ oscillate.

`why do they use the spatial strain in Abaqus ?`

6 ⋆ Solving a mixed problem

One may define a loading path prescribing several components of strain inclrements $\breve{\Delta}\epsilon_j$ and several complementary components of stress increments $\breve{\Delta}T_i$ with $i \neq j$. A geotechnical example of mixed control is an oedometric test with the prescribed axial stress and the prescribed zero radial strain.

INCREMENTALDRIVER uses a routine `usolver( )` to solves a system of linear equations $\mathbf{y} = \mathsf{E}.\mathbf{x}$ in which 6×6 stiffness matrix E may be unsymmetric. The unknown components may lie on both sides of the equation. Six unknowns are partly components $\Delta\epsilon_i$ of strain increments and partly components ΔT_j of stress increment. The prescribed components $\breve{\Delta}\epsilon_j$ and $\breve{\Delta}T_i$ have the complementary indices $i \neq j$. Geotechnical materials are often nonlinear and the stress increment returned by `umat` is not a linearly related to the strain increment $\Delta\mathbf{T} \neq \mathsf{E}.\Delta\boldsymbol{\epsilon}$. Therefore an iterative correction of the

unknown components is necessary. Corrections to ΔT_j are calculated directly by `umat` but correction to $\Delta\epsilon_i$ must be found iteratively using the given Jacobian matrix. For this purpose a so-called undesired stress increment $\breve{\Delta}T_i - \Delta T_i^k$ is calculated where the components $\breve{\Delta}T_i$ are given. The Remaining values ΔT_j will be ignored by solver. The old increment $\Delta\mathbf{T}^k$ is taken from the latest iteration (calculated with $\Delta\boldsymbol{\epsilon}^k$). The correction to the strain increment

$$\mathbf{c}^\epsilon = \mathsf{E}^{-1}.(\breve{\Delta}\mathbf{T} - \Delta\mathbf{T}^k) \quad \text{with} \quad \Delta\boldsymbol{\epsilon}^{k+1} = \Delta\boldsymbol{\epsilon}^k + \mathbf{c}^\epsilon \tag{16}$$

is added to the unknown components of the strain increment only. Components j with prescribed strain increments $\breve{\Delta}\epsilon_j$ have $c_j^\epsilon = 0$, of course. Using well programmed `umat`s this Newton iteration should converge very fast. Our linear solver `usolver` can deal with unknowns lying on both sides of equation $(16)_1$. The zero-one column matrix `ifstress` points to components with the prescribed stress increment.

The formal parameters of `usolver` are
`usolver(ddsdde,c_dstran,u_dstress,ifstress,ntens)`
wherein
`ddsdde` = stiffness E;
`c_dstran` = correction to strain increment $\mathbf{c}_\epsilon$;
`u_dstress` = undesired stress $\breve{\Delta}T_i - \Delta T_i$ (with zeros where strain components are prescribed);
`ifstress` = a list of 6 flags set to 1 or 0 if the stress or strain component is prescribed, respectively;
`ntens` = 6 number of equations.

The algorithm used in `usolver` has been described in [2].

7 ⋆ Solving a perturbation problem

One may wish to examine in detail the material response by superposing a given state $(\mathbf{T}, e, \boldsymbol{\alpha})$ with load increments in different directions. Usually we apply strain increments of identical length but different directions. The resulting stress responses form a so-called *response envelope* (or response polar) in the stress space. Conversely, one may also apply unit stress perturbations and examine the material response in strain space. This can also be done experimentally, cf. Lewin [4] or Royis and Doanh [6]. In order to produce the response polars, INCREMENTALDRIVER must undo the updates done by `umat` after each increment. In the case of stress probes we must complete the EI first (undoing the temporary end-states from inaccurate iterations), then record the accepted end-state and finally remove this state from the material memory to perform a stress probe in a different direction[8].

The strain or stress perturbations are calculated within a special mode of execution initiated by the keywords `*PerturbationsS` or `*PerturbationsE` in the `test.inp` input file. They generate stress probes or strain probes, respectively.

Having evoked perturbation as a separate step in `test.inp` file, information about the number of probes number of iterations and the time increment (for rate dependent models) must be provided. The next keyword, (usually `*Rendulic` or `*RoscoeIsomorph`) should define transformed variables and the size R of the probes must be given. For example, `*PerturbationsE` with `*RoscoeIsomorph` means that different strain probes of identical size are applied such that

$$\sqrt{(\breve{\Delta}\epsilon_P)^2 + (\breve{\Delta}\epsilon_Q)^2} = R \tag{17}$$

The size R is stored in `deltaLoad(1)` in the program.

8 Working with linearly transformed components

In some cases it is convenient to describe the stress/strain path in terms of transformed strain rate or stress increments.

[8] Alternatively one could perform a series of small linear cycles in various directions each consisting of two strain increments with opposite sense. However, such cycles do not reproduce the perturbation *exactly* because of small residual effects which may remain in the state variables after each cycle.

8.1 ⋆ Roscoe's variables

The most popular transformed components are known as Roscoe's variables:

$$p = -(T_1+T_2+T_3)/3 \qquad \text{and} \quad q = -T_1 + (T_2+T_3)/2 \quad \text{with} \tag{18}$$

$$\epsilon_v = -(\epsilon_1+\epsilon_2+\epsilon_3) \qquad \text{and} \quad \epsilon_q = -\frac{2}{3}\epsilon_1 + \frac{1}{3}(\epsilon_2+\epsilon_3) \tag{19}$$

defined with principal stresses and strains for axially symmetric states. Generally, we admit any *linear* combination of strain rate components to be a transformed strain rate, and analogously for the transformed stress rate. No combinations of components of stress- *and* strain rates are allowed.

The definition of p and q reflects the manner in which a triaxial test is actually controlled. In a conventional triaxial test (with the cell pressure acting also on the the upper end plate) we prescribe p and q rather than T_1 and $T_2 = T_3$.

In an undrained triaxial compression test of a fully saturated soil sample we may prescribe vertical and lateral pressures (the components of the total stress $\mathbf{T}^{tot}$,) but the pore pressure is usually a part of the material response and therefore the effective stress $\mathbf{T}$ cannot be directly controlled. Assuming incompressibility of water we control the volumetric strain ($\epsilon_v = \text{const}$) and the difference $q = -(T_1 - T_2) = -(T_1^{tot} - T_2^{tot})$ which is the deviatoric *effective* stress (= deviatoric total stress). The material response is observed through the complementary variables p (after subtraction of pore pressure) and ϵ_q. We cannot treat the effective stress components as prescribed because they are affected by the build-up of pore water pressure, i.e. by a part of the material response which we do not control. In this case choosing the transformed variables together with a mixed control seems quite natural for numerical simulation.

A transformed stress rate measure[9] can be written as $\Delta\mathbf{t} = \mathsf{M}.\Delta\mathbf{T}$, wherein M is a non-singular matrix of constant coefficients ('the regular linear substitution' or 'linear transformation'). The corresponding strain rate follows from the postulated conservation of second-order work $\Delta\mathbf{T}^T.\Delta\boldsymbol{\epsilon} = \Delta\mathbf{t}^T.\Delta\mathbf{e}$. Hence we have

$$\Delta\mathbf{t} = \mathsf{M}.\Delta\mathbf{T} \qquad \Delta\mathbf{e} = \mathsf{M}^{-T}.\Delta\boldsymbol{\epsilon} \qquad \Delta\mathbf{T} = \mathsf{M}^{-1}.\Delta\mathbf{t} \qquad \Delta\boldsymbol{\epsilon} = \mathsf{M}^{T}.\Delta\mathbf{e} \tag{20}$$

Matrix M the non-singular not necessarily symmetric nor orthogonal. Linear constitutive relation between $\Delta\mathbf{t}$ and $\Delta\mathbf{e}$ is

$$\Delta\mathbf{t} = \bar{\mathsf{E}}.\Delta\mathbf{e} \quad \text{with} \quad \bar{\mathsf{E}} = \mathsf{M}.\mathsf{L}.\mathsf{M}^T, \tag{21}$$

One says that $\bar{\mathsf{E}}$ and E are congruent. Note that the components of $\Delta\mathbf{t}$ and $\Delta\mathbf{e}$ need not be invariant. In Section 8.2 we demonstrate M for Roscoe's invariants and in Section 8.3 we explain the equilibrium iteration for incrementally nonlinear constitutive models under stress or mixed control.

8.2 Using Roscoe's invariants in IncrementalDriver

The Roscoe's invariants as components of stress and strain can be evoked by the keyword `*Roscoe` in the input file `test.inp` for IncrementalDriver. The strain and stress components are

$$\begin{Bmatrix} \Delta p \\ \Delta q \\ \Delta z \\ \Delta T_{12} \\ \Delta T_{13} \\ \Delta T_{23} \end{Bmatrix} = \begin{bmatrix} -1/3 & -1/3 & -1/3 & 0 & 0 & 0 \\ -1 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} \Delta T_{11} \\ \Delta T_{22} \\ \Delta T_{33} \\ \Delta T_{12} \\ \Delta T_{13} \\ \Delta T_{23} \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} \Delta \epsilon_v \\ \Delta \epsilon_q \\ \Delta \epsilon_z \\ \Delta \gamma_{12} \\ \Delta \gamma_{13} \\ \Delta \gamma_{23} \end{Bmatrix} = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 \\ -2/3 & 1/3 & 1/3 & 0 & 0 & 0 \\ 0 & -1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} \Delta \epsilon_{11} \\ \Delta \epsilon_{22} \\ \Delta \epsilon_{33} \\ \Delta \gamma_{12} \\ \Delta \gamma_{13} \\ \Delta \gamma_{23} \end{Bmatrix} \tag{22}$$

or briefly $\Delta\mathbf{t} = \mathsf{M}.\Delta\mathbf{T}$ and $\Delta\mathbf{e} = \mathsf{M}^{-T}.\ \Delta\boldsymbol{\epsilon}$. A somewhat arbitrary stress component Δz is introduced to insure the one-to-one relation between $\Delta\mathbf{t}$ and $\Delta\mathbf{T}$. The component $\Delta\epsilon_z$ of $\Delta\mathbf{e}$ is the strain counterpart of Δz. We may use them to impose the axial symmetry of stress or strain by setting $\Delta z = 0$ or $\Delta\epsilon_z = 0$, respectively.

8.3 ⋆ Equilibrium iteration with transformed variables

One may want to define loading prescribing some components of $\Delta\mathbf{e}$ and some the complementary components of $\Delta\mathbf{t}$. Analogously as in Section 6 we need to perform the equilibrium iteration due to nonlinearity $\Delta\mathbf{t} \neq \bar{\mathsf{E}}.\Delta\mathbf{e}$. Iteratively we try to diminish the undesired stress $u_i = \breve{\Delta}t_i - \Delta t_i^k$ calculated from components with prescribed the stress increments $\breve{\Delta}t_i$

[9] This $\mathbf{t}$ should not to be mixed up with the traction (=stress vector).

and with Δt_i^k from the latest iteration. As in Section 6, components u_j corresponding to the prescribed $\breve{\Delta} e_j$ are ignored by the solver[10]. The correction $\mathbf{c}^e$ of the transformed strain increment is found from

$$\mathbf{c}^e = \bar{\mathsf{E}}^{-1}.(\breve{\Delta}\mathbf{t} - \Delta\mathbf{t}^k) \quad \text{and} \quad \Delta\mathbf{e}^{k+1} = \Delta\mathbf{e}^k + \mathbf{c}^e \tag{23}$$

This correction pertains to the *unprescribed* transformed strain components only. The prescribed increments $\breve{\Delta} e_j$ must not be corrected and hence $c_j^e = 0$ should be considered in $(23)_1$.

8.4 Other transformed variables implemented in IncrementalDriver

The conventional set of components is used in incrementalDriver if the data describing the test path is preceded by the keyword `*Cartesian`. Apart from `*Roscoe` two other sets of transformed components can used by incrementalDriver, namely :
`*RoscoeIsomorph` with

$$\Delta\mathbf{t} \;=\; [\Delta P, \Delta Q, \Delta Z, \Delta T_{12}, \Delta T_{13}, \Delta T_{23}]^T \qquad \Delta\mathbf{e} = [\Delta\epsilon_P, \Delta\epsilon_Q, \Delta\epsilon_Z, \Delta\gamma_{12}, \Delta\gamma_{13}, \Delta\gamma_{23}]^T \tag{24}$$

and `*Rendulic` with

$$\Delta\mathbf{t} \;=\; [\Delta\sigma_1, \sqrt{2}\Delta\sigma_2, \Delta Z, \Delta T_{12}, \Delta T_{13}, \Delta T_{23}]^T \qquad \Delta\mathbf{e} = [\Delta\epsilon_1, \sqrt{2}\Delta\epsilon_2, \Delta\epsilon_Z, \Delta\gamma_{12}, \Delta\gamma_{13}, \Delta\gamma_{23}]^T \tag{25}$$

If these keywords appear before the test path data in the input file `test.inp` then incrementalDriver will use the following linear transformation matrices:

$$\mathsf{M}^{\text{Rosc}} = \begin{bmatrix} -1/\sqrt{3} & -1/\sqrt{3} & -1/\sqrt{3} & 0 & 0 & 0 \\ -2/\sqrt{6} & 1/\sqrt{6} & 1/\sqrt{6} & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \mathsf{M}^{\text{Rend}} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{26}$$

respectively. Both matrices are orthogonal so $\mathsf{M}^{-T} = \mathsf{M}$. This is not true for the M matrix corresponding to `*Roscoe`.

8.5 ⋆ Flowchart of a single step for transformed variables

In this flowchart the equilibrium iteration is explained in terms of transformed components. The term "where" should be understood as "at the components for which".

a. only in the first increment of a step: initialize stress $\mathbf{T} = \mathbf{T}_0$, strain $\boldsymbol{\epsilon} = \mathbf{0}$ and state variables $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0$
call `umat` to calculate E for $\Delta\boldsymbol{\epsilon} = \mathbf{0}$ and $\Delta t = 0$
b. read the type loading, choose M and M^T and transform stiffness to $\bar{\mathsf{E}} = \mathsf{M}.\mathsf{E}.\mathsf{M}^T$
c. read increments of transformed components $\breve{\Delta} t_i$ and $\breve{\Delta} e_j$ (distinguishing $i \neq j$ from `ifstress`)
d. make the initial guess of the stress increment, e.g. $\Delta\mathbf{t}^k = \mathbf{0}$
e. remember the end-stress $\mathbf{T}_r = \mathbf{T}$ and the end-state $\boldsymbol{\alpha}_r = \boldsymbol{\alpha}$ from the latest increment
`r_stress(:)= stress` and `r_statev(:)=statev(:)`
f. enter the equilibrium iteration -loop numbered with ^{k}
g. calculate the undesired (out-of-balance) stress $u_i = \breve{\Delta} t_i - \Delta t_i^k$ for components i with prescribed stress, wherein $\Delta\mathbf{t}^k$ is the stress increment the the latest iteration
h. set $\mathbf{c}^e = \mathbf{0}$ where $\Delta\mathbf{e}$ is prescribed and solve $\mathbf{c}^e = \bar{\mathsf{E}}.(\breve{\Delta}\mathbf{t} - \Delta\mathbf{t}^k)$ `call USOLVER(ddsdde_bar,c_dstran,u_dstress,ifstress,ntens)`
i. correct the increment $\Delta\mathbf{e}^{k+1} = \Delta\mathbf{e}^k + \mathbf{c}^e$ `where (ifstress == 1) dstran = dstran + c_dstran`
j. convert the strain increment to cartesian form $\Delta\boldsymbol{\epsilon} = \mathsf{M}^T.\Delta\mathbf{e}$ `dstran_Cart`
k. call `umat` with the corrected $\Delta\boldsymbol{\epsilon}^{k+1}$ and with $\mathbf{T}_r, \boldsymbol{\alpha}_r$ to update $\mathbf{T}$ and $\boldsymbol{\alpha}$ and to get a new E
l. if `iiter < maxiter`:
- improve the approximation of the stress inc $\Delta\mathbf{t}^{k+1} = \mathsf{M}.(\mathbf{T} - \mathbf{T}_r)$
- undo the update of stress and state performed by `umat` $\mathbf{T} = \mathbf{T}_r$, $\boldsymbol{\alpha} = \boldsymbol{\alpha}_r$
- repeat the EI with $k := k + 1$

[10] One may pad u_j them with zeroes

m. if `iiter == maxiter`:
- add `dstran_Cart(:)` to $\boldsymbol{\epsilon}$
- accept the update of stress $\mathbf{T}^{k+1}$ and state $\boldsymbol{\alpha}^{k+1}$ performed by `umat`
- reset the couter of iteration $k := 1$

n. use the most recent E and $\Delta\mathbf{t}$ as the initial guess in the next increment

9 Preparing input for incrementalDriver

incrementalDriver reads three input files: `parameters.inp`, `initialconditions.inp` and `test.inp`. For all these files the following rules apply

- No empty lines;
- No comment lines;
- The end-of-line comments are allowed but after 15 spaces;
- Integers in input must not have a decimal point;
- Strings in input must not have apostrophes. They must start from the first column (no preceding spaces). The asterisk is the first character of each keyword. The capital letters are important in the keywords.
- Only spaces can be used as separators of data in a single line (no commas or semicolons)

The above filenames can be overridden by the parameters in the command line. The default values of the command-line parameters are given in the following example:

```
incrementalDriver test=test.inp param=parameters.inp ini=initialconditions.inp verbose=true
```

If we set `verbose=false` no information about the currently calculated step and increment will be output to the screen.

9.1 Material constants

The material constants are read from the file `parameters.inp`. All should be input one item per line in the following sequence

```
cmname
nprops
props(1)
props(2)
...
props(nprops)
```

Remarks:
The current material name `cmname` and the number of material constants `nprops` is followed by a list of constants `props(1), props(2) ....` The meaning of `props` depends on the internal definitions in `umat`. Contrarily to the syntax of Abaqus™ just one material constant per line is read. The end-of-line comments after the data are allowed for. Note that the name of the material `cmname` is `character*80` so in this case the end-of-line comment should start from column 81 or farther.

9.2 Initial conditions

The initial conditions are read from the file `initialconditions.inp` in the following sequence

```
ntens
stress(1)
stress(2)
...
stress(ntens)
nstatv
statev(1)
statev(2)
...
statev(nstatv)
```

Remarks:

1. We always have `ndi` = 3 which means that the first three components `stress(1), stress(2), stress(3)` must be specified. They correspond to T_{11}, T_{22}, T_{33}
2. incrementalDriver is working internally with full six components so for `ntens` < 6 the initial values of the remaining components are just all set to zero.
3. The components of `stress` are listed in the following sequence $T_{11}, T_{22}, T_{33}, T_{12}, T_{13}, T_{23}$
4. Despite transformed variables used to define loading the initial stress is always defined with the classical Cartesian components.
5. If the input is too short the remaining components of `statev( )` will be padded with zeros. In other words, if the end of the file `initialconditions.inp` is encountered by incrementalDriver after, say, `statev(5)` and `nstatv > 5` then each of the values `statev(6)` , `statev(7)` ... `statev(nstatv)` is set to zero.

9.3 Prescribing the test path

The strain/stress path is read from the file `test.inp`. In the first line the name of the output file (`character(len=260)` must be given. If `#` is found in the first line then only the portion on the left-hand side of `#` is interpreted as the name of the output file. The portion of the first line on the right-hand side of `#` is copied to the output file as a heading line.

```
outputFileName   #  optional heading copied to the outputFile
```

The output file name is obligatory but it can be overridden by a parameter in the command line, for example:
`incrementalDriver out=output.out`
The heading (if we define one beyond `#` in the `test.inp`) will not be affected by command line.

Further lines contain the description of *steps*. Similarly as in ABAQUS^{TM} the word "step" means a sequence of similar prescribed increments. The art of step is defined by a keyword preceded with an asterisk.

9.3.1 Proportional path

For proportional increments one may use `*LinearLoad`

```
*LinearLoad
ninc maxiter deltaTime          # number of increments, max. number of EI  and total time for the step
*Cartesian                    #  other possibility are:  *Roscoe, *RoscoeIsomorph, *Rendulic
ifstress(1)  deltaLoad(1)    #  0/1 Flag  followed by an increment of stress or strain
ifstress(2)  deltaLoad(2)
...
ifstress(6)  deltaLoad(6)
```

The components of "load" pertain to the whole step i.e. depending on the flag `ifstress(i)` the i-th component is calculated as prescribed stress increment `ddstress(i) = deltaLoad/ninc` or strain increment `dstran(i) = deltaLoad/ninc`.

In the current version of the program there is *no convergence criterion*. The desired number `maxiter` of equilibrium iterations is carried out irrespectively of convergence which may have already been achieved. After the last equilibrium iteration the next load increment is applied irrespectively of whether the previous iteration was successful (small out of balance stress) or not. Controlling stress components we should carefully examine if the desired stress path was indeed applied to the material. One can prescribe an "impossible" stress increment (e.g. going outside the yield surface). In this case INCREMENTALDRIVER will do its best to achieve the required state but the calculation will not be interrupted if INCREMENTALDRIVER fails. Insufficient accuracy within the stress controlled regime may also be caused by too small number `maxiter` of iterations. The number of equilibrium iterations should be increased for inaccurate Jacobian matrices in `umat.f` and for large increments.

9.3.2 Harmonic oscillation

A useful alternative, especially for cyclic loading, is a harmonic oscillation `*CirculatingLoad` with the syntax

```
*CirculatingLoad
ninc  maxiter deltaTime # number of incr. per cycle, max. number of EI  and total time for the step
*Cartesian                #  here other possibilities are:  *Roscoe, *RoscoeIsomorph, *Rendulic
ifstress(1) deltaLoadCirc(1) phase0(1) deltaLoad(1) #  1/0Flag, amplitude, phase and linear step load
ifstress(2) deltaLoadCirc(2) phase0(2) deltaLoad(2)
...
ifstress(6) deltaLoadCirc(6) phase0(6) deltaLoad(6)
```

Each stress or strain component consists of a harmonic portion superposed by a linear part (monotonic) portion. The harmonic portion is described by the amplitude `deltaLoadCirc(i)` and by the initial phase `phase0(i)` The linear part described by `deltaLoad`. The linear portion of each increment is constant over the step and it is calculated with

$$B_i = \texttt{deltaLoad(i)}/\texttt{ninc} \tag{27}$$

The harmonic portion is varying with the step time $t =$ `deltaTime` treated as a full period. This means that the total increments of each component are calculated from

$$\Delta L_i = \dot{\omega}\Delta t A_i \cos(\dot{\omega}t + \omega_i^{(0)}) + B_i \tag{28}$$

wherein ΔL_i denotes either the increment of stress `ddstress(i)` or the increment of strain `dstran(i)`, depending on the value of `ifstress(i)` for the i-th component. Other variables are
$\dot\omega =$ `wd = 2*Pi/deltaTime`
$\Delta t =$ `dtime`
$A_i =$ `deltaLoadCirc(i)`
$B_i =$`deltaLoad(i)/ninc`
$t =$ `time(1) + dtime/2 # step time in the middle of the increment`
$\omega_i^{(0)}$ `=w0(i)= phase0(i)`

9.3.3 ⋆ On transformation matrices

Inventing new transformation matrices M we may simulate different kinds of laboratory tests. However the applicability of this technique is limited. Consider an isobaric biaxial test under condition $\Delta T_1 + \Delta T_2 + \Delta T_3 = 0$ with $\Delta\epsilon_3 = 0$ and $\Delta\epsilon_1 = -0.01$. It turns out that a linear transformation matrix M for $\Delta\mathbf{t} = \mathsf{M}.\Delta\mathbf{T}$ and $\Delta\mathbf{e} = \mathsf{M}^{-T}.\Delta\boldsymbol{\epsilon}$ cannot be constructed for such loading. In this section we demonstrate the problem using principal components of strain and stress only. A remedy is proposed in Section 9.3.4.

A description of our loading requires the transformed components of stress $t_1 = \operatorname{tr}\mathbf{T}$, $t_2 =?$, $t_3 =?$ and the transformed components of strain $e_1 =?$, $e_2 = \epsilon_2$ and $e_3 = \epsilon_3$. Transformation matrix M should therefore have the form

$$\left\{\begin{array}{c} p \\ t_2 \\ t_3 \end{array}\right\} = \overbrace{\left[\begin{array}{ccc} f & f & f \\ ? & ? & ? \\ ? & ? & ? \end{array}\right]}^{=\mathsf{M}} \left\{\begin{array}{c} T_1 \\ T_2 \\ T_3 \end{array}\right\}, \qquad \left\{\begin{array}{c} e_1 \\ e_2 \\ e_3 \end{array}\right\} = \overbrace{\left[\begin{array}{ccc} a=? & b=? & c=? \\ 0 & d=? & 0 \\ 0 & 0 & e=? \end{array}\right]}^{=\mathsf{M}^{-T}} \left\{\begin{array}{c} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \end{array}\right\} \qquad (\mathsf{M}^{-T})^{-T} = \frac{1}{a}\left[\begin{array}{ccc} 1 & 0 & 0 \\ -b/d & a/d & 0 \\ -c/e & 0 & a/e \end{array}\right] \tag{29}$$

wherein $(29)_2$ is the most general form of M^{-T}. Its inverse[11] has the form $(29)_3$ which cannot be brought to the form $(29)_1$ needed for the prescription of stress increments. In order to circumvent this and similar problems we will extend the mixed control algorithm using *restrictions*.

9.3.4 ⋆ Imposing loading via *restrictions*

At first, we consider a linear material $\Delta\mathbf{T} = \mathsf{E}\cdot\Delta\boldsymbol{\epsilon}$. Suppose that all stress and strain increments are unknown so that we have 6 constitutive equations and 12 unknowns. The additional 6 equations are called restrictions in IncrementalDriver. Restrictions allows for more flexibility than the transformation Matrix M. The additional equations are assumed to have the general linear matrix form $\mathsf{M}_t\cdot\Delta\mathbf{T} + \mathsf{M}_e\cdot\Delta\boldsymbol{\epsilon} = \mathbf{m}$ where the components of 6×6 matrices M_t and M_e are arbitrary state functions independent of increments. We have solve the system of 12 equations

$$\begin{cases} \Delta\mathbf{T} & = \mathsf{E}\cdot\Delta\boldsymbol{\epsilon} \\ \mathsf{M}_t\cdot\Delta\mathbf{T} + \mathsf{M}_e\cdot\Delta\boldsymbol{\epsilon} & = \mathbf{m} \end{cases} \tag{30}$$

with 12 unknown increments $\Delta\mathbf{T}$ and $\Delta\boldsymbol{\epsilon}$ The restrictions are linear with respect to $\Delta\mathbf{T}$ and $\Delta\boldsymbol{\epsilon}$. Only constant components of M_t, M_e and $\mathbf{m}$ have been implemented[12] in IncrementalDriver. For the isobaric biaxial test from Section 9.3.3 the restrictions would be formulated as follows

$$\overbrace{\left[\begin{array}{ccc} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]}^{=\mathsf{M}_t}\cdot\left\{\begin{array}{c} \Delta T_1 \\ \Delta T_2 \\ \Delta T_3 \end{array}\right\} + \overbrace{\left[\begin{array}{ccc} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right]}^{=\mathsf{M}_e}\cdot\left\{\begin{array}{c} \Delta\epsilon_1 \\ \Delta\epsilon_2 \\ \Delta\epsilon_3 \end{array}\right\} = \overbrace{\left\{\begin{array}{c} 0 \\ -0.01 \\ 0 \end{array}\right\}}^{=\mathbf{m}} \tag{31}$$

In the `test.inp` file we can impose this loading in `ninc = 100` increments with `maxiter = 20` and within time `deltaTime = 1` using the following description of the step `*ObeyRestrictions`. This example shows the natural manner of restricting increments

```
*ObeyRestrictions
100 20  1.0
sd1 + 1.0*sd2 + sd3 = 0  # restriction 1 = isobaric condition
ed1 = -0.01              # restriction 2
```

[11] VerbMiT = {{a,b,c},{0,d,0},{0,0,e}}; Transpose[Inverse[MiT]] //MatrixForm
[12] In general they could be functions of time, stress etc.

```
ed3 = 0.0                  # ....
ed4=0
ed5    =    0.0d0
ed6  = -0                  # restriction 6
```

Note that parsing abilities of the IncrementalDriver are very limited. Here are some formal rules:

- The input line cannot be longer than 120 characters.
- No brackets `()`, no exponents `**` or divisions `/` may appear.
- Each component `sd1,sd2,sd3,sd4,sd5,sd6` of the stress increment and/or each component `ed1,ed2,ed3,ed4,ed5,ed6` of strain increment may appear exactly once on the left-hand side of the restriction, i.e. one cannot write `sd1 + sd1` . Lower case is obligatory.
- Each component constitutes a summand and can be preceded by a single factor, e.g. `9.0d-4*sd1` but not `3*3.0d-4*sd1` and not `sd1*3.14`.
  Summands without stress or strain components are not allowed, i.e. one cannot write `sd1 + 1.0 + sd2 = 0.0`
- Only numerical coefficients are allowed, not `t*sd1`
- If a factor precedes a component, the multiplication sign `*` is obligatory, e.g. `3*ed1+ed2=0` but not `3 ed1+ed2=0`.
- The summands are separated by `+` or `-`.
- The spaces are ignored but not inside numbers, of course.
- There must be just a single constant value in each restriction. It must appear on the right-hand side of the restriction and can be preceded by `-`. No multiplication may appear on the right-hand side.

The end-of-line comments must begin with `#`. Exactly 6 lines with restrictions must be formulated without blank lines between them.

9.3.5 ⋆ Equilibrium iteration with restrictions

Most constitutive models of soil mechanics are nonlinear. Hence we must handle the discrepancy between the constitutive stress increment $\mathbf{f}(\Delta\boldsymbol{\epsilon})$ which is a nonlinear function of $\Delta\boldsymbol{\epsilon}$ and the stress increment $\Delta\mathbf{T} = \mathsf{E} \cdot \Delta\boldsymbol{\epsilon}$ obtained from the linearized constitutive model, where $\mathsf{E} = \partial\mathbf{f}/\partial\boldsymbol{\epsilon}$. This discrepancy is removed in the course of the equilibrium iteration (EI). The EI can be coupled with restrictions. The algorithm is summarized in the following flowchart:

1. Find the preliminary stiffness E_0 assuming $\Delta\boldsymbol{\epsilon} = \mathbf{0}$ and $\Delta t = 0$.
2. Make the first guess $\Delta\boldsymbol{\epsilon}_1$ by solving restrictions with linearized constitutive relation $\Delta\mathbf{T}_1 = \mathsf{E}_0\Delta\boldsymbol{\epsilon}_1$. This leads to
$$\Delta\boldsymbol{\epsilon}_1 = [\mathsf{M}_t \cdot \mathsf{E}_0 + \mathsf{M}_e]^{-1} \cdot \mathbf{m} \tag{32}$$
3. Call `umat` with $\Delta\boldsymbol{\epsilon}_1$ to get the constitutive stress $\mathbf{f}(\Delta\boldsymbol{\epsilon}_1)$ and a better estimation of the tangential stiffness E_1
4. Find the error $\mathbf{N}$ obtained substituting the exact (nonlinear) stress increment $\mathbf{f}(\Delta\boldsymbol{\epsilon}_1)$ into the restrictions
$$\mathbf{N} = \mathsf{M}_t \cdot \mathbf{f}(\Delta\boldsymbol{\epsilon}_1) + \mathsf{M}_e \cdot \Delta\boldsymbol{\epsilon}_1 - \mathbf{m} \tag{33}$$
5. Find the correction $\mathbf{c}_\epsilon$ to the strain increment $\Delta\boldsymbol{\epsilon}_1$ such that $\mathbf{N}(\Delta\boldsymbol{\epsilon}_1 + \mathbf{c}_\epsilon) = \mathbf{0}$. For this purpose we expand $\mathbf{N}(\Delta\boldsymbol{\epsilon}_1 + \mathbf{c}_\epsilon) = \mathbf{N}(\Delta\boldsymbol{\epsilon}_1) + \dfrac{\partial\mathbf{N}}{\partial\Delta\boldsymbol{\epsilon}} \cdot \mathbf{c}_\epsilon$ in the Taylor series and hence
$$\mathbf{c}_\epsilon = -\left[\frac{\partial\mathbf{N}}{\partial\Delta\boldsymbol{\epsilon}}\right]^{-1} \cdot \mathbf{N} \quad \text{with} \quad \frac{\partial\mathbf{N}}{\partial\Delta\boldsymbol{\epsilon}} = \mathsf{M}_t \cdot \mathsf{E}_1 + \mathsf{M}_e \tag{34}$$
in which the new jacobian $\dfrac{\partial\mathbf{f}}{\partial\Delta\boldsymbol{\epsilon}} = \mathsf{E}_1$ is used.
6. Finally we update the strain increment
$$\Delta\boldsymbol{\epsilon}_2 = \Delta\boldsymbol{\epsilon}_1 + \mathbf{c}_\epsilon \tag{35}$$
calculate $\mathbf{f}(\Delta\boldsymbol{\epsilon}_2)$ and repeat the iteration.

9.4 Response envelopes in stress or in strain

A perturbation of stress `*PerturbationsS` or strain `*PerturbationsE` can be applied to an arbitrary state. It must be described in the input file `test.inp` as a separate step in `test.inp` with the following syntax

```
*PerturbationsE        # here  *PerturbationsS  for stress probes is also possible
 ninc, maxiter, deltaTime
*RoscoeIsomorph        # here  *Rendulic  is also possible (otherwise warning)
deltaLoad(1)
```

Perturbations of the first two components (of strain increments or stress increments) can be performed. They are applied radially in `ninc` different directions equally distributed over the 2π angle in the space of the first two components. Usually the transformed variables will be used here, namely `*RoscoeIsomorph` or `*Rendulic`. The `*Cartesian` or `*Roscoe` are also possible although less common in perturbations. The size of the perturbation is described by R which has a special meaning of the size of perturbation. For `*RoscoeIsomorph` it means

$$\sqrt{(\breve{\Delta}\epsilon_P)^2 + (\breve{\Delta}\epsilon_Q)^2} = R \tag{36}$$

9.5 Repetition of a group of steps

The keyword `*Repetition` introduces a group of `nSteps` which will be repeated `nRepetitions` times. Each step should be defined according to its own syntax and no end of loop line is defined.

```
*Repetition
nSteps  nRepetitions
....
....here follow nSteps preceded by their own keywords
...
```

It is the responsibility of the user to describe (without errors) the exact number of steps which should be repeated. For a typical cyclic loading we will usually need a combination of a repetition of two `*linearLoad` steps or a repetition of a single `*CirculatingLoad` steps.

9.6 Predefined popular paths

Several one-line steps have been predefined in the IncrementalDriverfor the convenience of geotechnical users. These are: `*OedometricE1`, `*OedometricS1`, `*TriaxialE1`, `*TriaxialS1`, `*TriaxialUEq`, `*TriaxialUq`, `*PureRelaxation`, `*PureCreep`, `*UndrainedCreep`

These paths define the principal stresses / strains assuming x_1-axial symmetry of the applied components. The material response depends on `umat` and it need not be axisymmetric, of course. The shear components of strain are prescribed as constant

```
*OedometricE1
ninc maxiter dtime
ddstran(1)   # lateral strain is assumed constant

*OedometricS1
ninc maxiter dtime
ddstress(1)  # lateral strain is assumed constant

*TriaxialE1
ninc maxiter  dtime
ddstran(1)     # lateral stress is assumed constant

*TriaxialS1
ninc maxiter dtime
ddstress(1)    # lateral stress is assumed constant

*TriaxialUEq
ninc  maxiter dtime
ddstran(2)    #  volume = constant, Roscoe's Delta epsilon_q is applied

*TriaxialUq
ninc  maxiter  dtime
ddstress(2)    # volume = const, Roscoe's Delta q is applied

*PureRelaxation
ninc maxiter  dtime

*PureCreep
ninc  maxiter dtime
```

```
*UndrainedCreep      # Roscoe's  Delta eps_v = 0 and Delta q  = 0   other stress inc also =0
ninc   maxiter dtime

*End   # can be used to terminate the calculation
```

9.7 Enforced exit condition (new in 2016)

Each step-command may be optionally extended by a short inequality condition which will be evaluated at the end of each increment. If this condition is satisfied the remaining increments of the current step will be skipped. Here we interrupt an oedometric compression if the horizontal stress is large enough, say if $T_2 < -200$, by writing

```
*OedometricE1 ? s2 < -200.0
100   10 1
-0.0002   # lateral strain is assumed constant
```

9.8 Import a loading path from an external file (new in 2016)

You may have test results in form of a file with subsequent states listed in individual lines. For this purpose we write a step command `*ImportFile` followed by the name of the file and we choose the columns to be input. We must ascribe a column to each of six components of loading. However, if we ascribe the zero'th column then the zero *value* will be prescribed to the corresponding component of stress or strain increment.

```
*ImportFile   followMe.inp  |  ncols    #  name of file and number of reals to be input per line
ninc   maxiter   deltaTime          #   numb. of increments, max. numb. of  EI and  step time interval
*Cartesian              # obligatory
ifstress(1)   column(1)    #  (0=strain 1=stress) & column in the file. column==0 means  no increment
ifstress(2)   column(2)
...
ifstress(6)   column(6)
columnWithTime          # only if deltaTime < 0 in the second line
```

10 Non-isothermic paths (not implemented as yet)

In fully coupled thermal-stress analysis the following (essential) variables are used as an input for `umat` :

$\mathbf{T}^t$ = `STRESS(NTENS)` = pre-rotated Cauchy stress at the beginning of the increment
ϵ^t = `STRAN(NTENS)`= logarithmic strain at the beginning of the incr. without thermal expansion.
$\Delta\epsilon$ = `DSTRAN(NTENS)` Array of strain increments. If thermal expansion is prescribed in the same material definition[13], $\Delta\epsilon$ are the mechanical (i.e. total minus thermal) strain increments.
t = `TIME(1/2)` = Value of step/total time at the beginning of the current increment.
Δt = `DTIME` = Time increment.
θ^t = `TEMP`= Temperature at the start of the increment.
$\Delta\theta$ = `DTEMP` = Increment of temperature.

The following (essential) variables are defined/modified by `umat` as output:

$\mathbf{T}^{t+\Delta t}$ = `STRESS(NTENS)` = Cauchy stress at the end of the increment.
$\partial\mathbf{T}/\partial\boldsymbol{\epsilon}$ = `DDSDDE(NTENS,NTENS)` = Jacobian of the constitutive model (for implicit time integration).
$\dot{r}$ = `RPL` = rate of mechanical dissipation density, i.e. rate of heat production *per unit time and volume* at the end of the incr. *caused by mechanical work.*
$\partial\mathbf{T}/\partial\theta$ = `DDSDDT(NTENS)` = Variation of the stress increments with respect to the temperature.
$\partial\dot{r}/\partial\boldsymbol{\epsilon}$ = `DRPLDE(NTENS)` = Variation of `RPL` with respect to the strain increments.
$\partial\dot{r}/\partial\theta$ = `DRPLDT` = Variation of the rate of dissipation `RPL` due to the temperature change ($\neq$ 1/ specific heat)

The relation between corrections $c_\sqcup$ of stress, strain, heat and temperature are

$$\left\{ \begin{array}{c} c_{\mathbf{T}} \\ c_{\dot{r}} \end{array} \right\} = \left[\begin{array}{cc} \partial\mathbf{T}/\partial\boldsymbol{\epsilon} & \partial\mathbf{T}/\partial\theta \\ \partial\dot{r}/\partial\boldsymbol{\epsilon} & \partial\dot{r}/\partial\theta \end{array} \right] \cdot \left\{ \begin{array}{c} c_{\boldsymbol{\epsilon}} \\ c_{\theta} \end{array} \right\} \tag{37}$$

[13] Input lines of the section `*MATERIAL` of the Abaqus input file `*.inp`

For the non-isothermic loading, increments can be defined by a combination of 7 (and not 6) components chosen from $\Delta\epsilon_{11}, \Delta\epsilon_{22}, \Delta\epsilon_{33}, \Delta\epsilon_{12}, \Delta\epsilon_{13}, \Delta\epsilon_{23}, \Delta\theta$ and $\Delta T_{11}, \Delta T_{22}, \Delta T_{33}, \Delta T_{12}, \Delta T_{13}, \Delta T_{23}, \Delta Q$. If the increments of the first seven quantities are known then no iteration is necessary and the INCREMENTALDRIVER can follow the strain-temperature path with just a single call to `umat` per increment. Otherwise a procedure similar to equilibrium iteration is necessary. In applications to partially saturated soils one may want to interpret θ as the suction and Q as the degree of saturation. In order to simulate undrained tests on partially saturated soils we have a constant water content so assuming constant intrinsic bulk densities of water and skeleton we need to impose $S_r e = \text{const}$ This leads in the language of our control variables to the linearized form $e\ \Delta Q + S_r(1+e)\ (\Delta\epsilon_{11} + \Delta\epsilon_{22} + \Delta\epsilon_{33}) = 0$. The coefficients in the above control equations are not constant but depend on the state variables which vary within increments so strictly speaking not only a nonlinear constitutive model but also a nonlinear control equation need to be solved.

For degree of saturation S_r and suction s one needs to define the direct dependence $S_r(s)$ via the thermal properties of the material, i.e. with the user's subroutine

```
SUBROUTINE UMATHT(U,DUDT,DUDG,FLUX,DFDT,DFDG,
     1 STATEV,TEMP,DTEMP,DTEMDX,TIME,DTIME,PREDEF,DPRED,
     2 CMNAME,NTGRD,NSTATV,PROPS,NPROPS,COORDS,PNEWDT,
     3 NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
```

Here the heat energy $U =$ `U` per mass and the inverse specific heat $\partial U/\partial\theta =$ `DUDT` may be defined. Moreover `UMATH` defines the heat flux `FLUX` and its variation `DFDT` with temperature and as a function of the temperature gradient `DFDG`

A ⋆ Linking INCREMENTALDRIVER with umats written in C++

INCREMENTALDRIVER is written in Fortran and umats are also recommended to be written in Fortran. However, INCREMENTALDRIVER can also be linked with umats written in C++ (although Abaqus people do not recommend C++). This section describes how to write a C++ umat (`umat.cpp`) and how to link it with INCREMENTALDRIVER or with Abaqus. The presented method has been tested under Windows XP with Intel 9.0 Fortran compiler and with Microsoft Visual C++ .NET Version both under Microsoft Visual Studio .NET 2003 (MSDE 7.1 and Ms .NET Framework 1.1). It is also running with Abaqus 6.6.

If you like using the DOS window, in order to build INCREMENTALDRIVER with a C++ umat you may try something like

```
>  cl /I<include-directory> /c umat-full.cpp
>  ifort incrementalDriver.f usolve.f  umat-full.obj /link /out:MixedDriver.exe
```

however the MSDE (Microsoft Developer Environment) allows for nice simultaneous debugging of both Fortran and C++ in a single project. Probably the easiest way to develop and to test `umat.cpp` under INCREMENTALDRIVER is to start two projects one with fortran files only and the other with C++ files only. Let us call them

1. `LA`-Project containing `incrementalDriver.f, usolve.f` and `var_elastic_umat.f`
2. `Pure`-Project containing `pure.cpp` and `umat-full.cpp` and in my case a header file `tensor.h`

Each project should be tested individually before we start to mix them. The `var_elastic_umat.f` is a version of a Fortran umat. Fortran umats are described in detail in the Abaqus manual. The Fortran umat is called by the main program contained in the `incrementalDriver.f` file. The file `usolve.f` also belongs to the fortran project because it contains a solver (necessary for INCREMENTALDRIVER).

In the C++ project, the `pure.cpp` file contains the `main` calling unit. The C++ umat routine is contained in the file `umat-full.cpp`. The main program in `pure.cpp` is necessary for formal reasons only (there must be a `main` in the project). Apart from initialization of variables and calling umat it does practically nothing. C++ programmers not familiar with MSDE may face the problem of "procompiled headers" (this option is default in MSDE). The way around it is to right-click on the `pure` project in the Solution Explorer Window and under Properties | C/C++ | Procompiled Headers to choose not using the latter (do this for the Release and for the Debug environment). Compile the C++ project within the "Debug" environment.

For usage with `pure.cpp` and with `incrementalDriver.f` the umat code in `umat-full.cpp` should be defined like this

```
extern "C"
void  UMAT(double *stress, double *statev, double   *ddsdde,
     double *sse, double *spd, double *scd, double *rpl, double *ddsddt,
     double *drplde, double *drpldt, double *stran, double *dstran,
     double *tajm, double *dtajm, double *temp, double *dtemp, double *predef,
     double *dpred, char *cmname, int *ndi,
     int *nshr, int *ntens, int *nstatv, double *props,
     int *nprops, double *coords, double drot[][3], double *pnewdt,
     double *celent, double  dfgrd0[][3], double dfgrd1[][3], int *noel,
     int *npt,  int *layer,  int *kspt,  int *kstep,
     int *kinc){
  .........
     }
```

Of course, this should be repeated as a prototype in `pure.cpp`. The `main` in `pure.cpp` should be something like

```
int _tmain(int argc, _TCHAR* argv[])
{ int i,  kinc,  kspt, kstep, layer, ndi,  nshr,  ntens,  nstatv, nprops, noel, npt;
  double  celent, coords[3], *ddsdde, ddsddeFull[NTENS][NTENS], ddsddt[NTENS],
  dfgrd0[3][3], dfgrd1[3][3], dpred[1], drot[3][3], drplde[NTENS],
      drpldt, dstran[NTENS], stran[NTENS], temp, dtemp, pnewdt, predef[1], props[NPROPS],
      rpl, scd, spd, sse, statev[NSTATV],
      stress[NTENS], time[2], dtime;
  char* cmname="wo2 ";
  ddsdde = &ddsddeFull[0][0]; // 1-D ddsdde pointer shows to ddsddeFull[][] data
  ndi=NDI; noel=NOEL; nprops= NPROPS; npt=NPT; nshr=NSHR ; nstatv=NSTATV ;ntens=NTENS ;
  ....
  }
```

where the upper case variables denote the global constants. The above heading is followed by setting the values of the parameters and by calling of umat

```
UMAT(stress, statev, ddsdde, &sse, &spd, &scd, &rpl, ddsddt,
  drplde, &drpldt, stran, dstran,  time, &dtime, &temp, &dtemp, predef,
  dpred, cmname, &ndi,  &nshr, &ntens, &nstatv, props,
  &nprops, coords, drot, &pnewdt,  &celent, dfgrd0, dfgrd1, &noel,
  &npt, &layer, &kspt, &kstep, &kinc);
```

The code of incrementalDriver is identical for C++-umats and for Fortran-umats. The call-of-umat line has the common form

```
       call  UMAT(stress,statev,ddsdde,sse,spd,scd,
     &  rpl,ddsddt,drplde,drpldt,
     &  stran,dstran_Cart,time,dtime,temp,dtemp,predef,dpred,cmname,
     &  ndi,nshr,ntens,nstatv,props,nprops,coords,drot,pnewdt,
     &  celent,dfgrd0,dfgrd1,noel,npt,layer,kspt,kstep,kinc)
```

Suppose our both projects, the `LA`-Project and the `Pure`-Project, can be built and they both run correctly. We may now build the mixed-language exe. For this purpose we remove `var_elastic_umat.f` from the `Pure`-Project and in the properties of this "Debug" project (righ-tclick in the Solution Explorer Window) under Properties | linker | Input we put under "Additional Dependencies" the full path and name of the object file from the `pure` project. In my case it is:
`D:\i11\papers\element-tests\c\pure-C\pure\pure\Debug\umat-full.obj`.
Moreover, in the same window under "Ignore Specific Library" we put LIBC.LIB.

B ⋆ C++ umat used with Abaqus™

Abaqus takes either Fortran `var_elastic_umat.f` or an object file `umat-full.obj` from which it can internally produce a dll and link it. This necessitates:

- a small modification of the heading in the `umat-full.cpp`.
- A compilation of the code in the "Release" configuration or `cl /I<include-directory> /c umat-full.cpp` from DOS

The following (more or less) heading of the `umat-full.cpp` can be found on the Abaqus technical support pages

```
#define CHNAME(id) char* id, const unsigned int id ## _len
 extern "C"
 void  _stdcall  UMAT(double *stress, double *statev, double   *ddsdde,
      double *sse, double *spd, double *scd, double *rpl, double *ddsddt,
      double *drplde, double *drpldt, double *stran, double *dstran,
      double *tajm, double *dtajm, double *temp, double *dtemp, double *predef,
      double *dpred, CHNAME(cmname),  int *ndi,
      int *nshr, int *ntens, int *nstatv, double *props,
      int *nprops, double *coords, double drot[][3], double *pnewdt,
      double *celent, double  dfgrd0[][3], double dfgrd1[][3], int *noel,
      int *npt,  int *layer,  int *kspt,  int *kstep,
      int *kinc){
      .....
      }
```

(the double `##` is a so-called "token-pasting" operator in the preprocessor language, but probably you know that already if you are writing in C). Visit the Abaqus online support system
`http://abaqus.custhelp.com/cgi-bin/abaqus.cfg/php/enduser/home.php?p_sid=ZByLlEHi`
to learn how to write umat in C++ in operation systems different than MS Windows.

Having modified the `umat-full.cpp` we attempt to build the `pure`-Project under "Release" configuration. The linking with `pure` will fail (unless we modify suitably the calling line) but it does not bother us because only the semi product `umat-full.obj` is needed. We find this file in the Release subdirectory of the root directory of the `pure` project. This is the file which we put in the working directory of Abaqus. Now we may start our Abaqus job with the DOS command line

```
abaqus j=1 inp=inputfile.inp user=umat-full.obj
```

instead of the usual

```
abaqus j=1 inp=inputfile.inp user=var_elastic_umat.for
```

C ⋆ C++ umat used with Tochnog

Tochnog is a commercial FE-Program which allows for using umat with the syntax almost identical with the one of Abaqus. The C code of umat is described on the Tochnog www-pages or on `http://www.koders.com/` if you search for umat.c. The code looks like this

```
typedef long int integer;
typedef double doublereal;
typedef short ftnlen;

int umat_(stress, statev, ddsdde, sse, spd, scd, rpl, ddsddt,
     drplde, drpldt, stran, dstran, time, dtime, temp, dtemp, predef,
    dpred, cmname, ndi, nshr, ntens, nstatv, props, nprops, coords, drot,
    pnewdt, celent, dfgrd0, dfgrd1, noel, npt, layer, kspt, kstep, kinc,
    cmname_len)
doublereal *stress, *statev, *ddsdde, *sse, *spd, *scd, *rpl, *ddsddt, *
    drplde, *drpldt, *stran, *dstran, *time, *dtime, *temp, *dtemp, *
    predef, *dpred;
char *cmname;
integer *ndi, *nshr, *ntens, *nstatv;
doublereal *props;
integer *nprops;
doublereal *coords, *drot, *pnewdt, *celent, *dfgrd0, *dfgrd1;
integer *noel, *npt, *layer, *kspt, *kstep, *kinc;
ftnlen cmname_len;
{
......
return 0;
}
```

Mind that the length of `cmname` is passed differently to Tochnog than to Abaqus and the name of the subroutine is `int umat_` and not `extern "C" void _stdcall UMAT` . Moreover `cmname_len` is of the type `short` and not of the type `const unsigned int`.

D ⋆ incrementalDriver under Linux or Cygwin

The following description is provided by David Mašín `http://www.natur.cuni.cz/~masin/eindex.php`

To compile incrementalDriver using open-source `gcc` compiler
(`http://gcc.gnu.org\verb` for Linux OS, `http://www.cygwin.com` for Windows OS)
and link it with C++ umat proceed as follows:

1. Compile C++ umat using command
   `gcc -c *.cc`
   (tested with `umat_prague`)
2. Rename `aba_param.inc` to `ABA_PARAM.INC` (In Linux only, due to case-sensitive filenames)
3. Compile incrementalDriver files using commands

   ```
   gfortran -c usolve.f
   gfortran -c  incrementalDriver.f
   ```

   Note that `usolve.f` has been compiled first.
   The module `unsymmetric_module.mod` needed for the compilation of `incrementalDriver.f` is created during the compilation of `usolve.f`
4. Link C++ umat with incrementalDriver and create incrementalDriver executable:
   `gfortran -o incrementalDriver *.o -lstdc++ umatdir/umat_prague/*.o`
   `umatdir` should be replaced by specific directory in which `umat_prague` is located.
   Note that `gcc` option `-lstdc++` is crucial for linking, as otherwise C++ libraries needed by `umat_prague` are not loaded by `gfortran`.

D.1 ⋆ umatONE.obj for ABAQUS™ from umatONE.f

1. put all fortran files (e.g. `umat.f, sigini.f, sdvini.f` ...) into a single file `umatALL.f` (modules first)
2. create a dummy fortran program. e.g.

```
program nic
use niemunis_tools_lt
real(8) ::  T(3,3)
T = delta
write(*,*)  ( T .xx. T )
end  program nic
```

3. create a "Release" project under INTEL FORTRAN™ 8.3 MS Developer Studio for ABAQUS 6.6 or under COMPAQ VISUAL FORTRAN for Abaqus 6.5 (patches to CVF are necessary for `niemunis_tools_lt`)
4. include `nic.f` and `umatALL.f` in the project
5. if under INTEL FORTRAN™ 8.3 then set :
   `Properties| Fortran| ExternalProcedures| CallingConvention| CVF`
6. build the project
7. find `umatALL.obj` in the release subdirectory
8. use `umatALL.obj` like this: `abaqus j=1 inp=blabla.inp user=umatONE.obj`

E ⋆ General remarks on mixed Fortran/C++ codes

`call umat()` needs capital letters in the name of the function in C.

The C function's formal arguments must be declared as pointers to the appropriate data type.

In C, the first four elements of an array declared as `X[3][3]` are: `X[0][0] X[0][1] X[0][2] X[1][0]`

In Fortran, the first four elements are: `X(1,1) X(2,1) X(3,1) X(1,2)`

Language	Array Declarations	Array Reference from Fortran
Fortran:	real(8), DIMENSION :: x(i,k)	x(i,j)
C++ :	double x[k][i];	x(i-1,j-1)

In Fortran, you can specify these conventions in a mixed-language interface with the INTERFACE statement or in a data or function declaration. C/C++ and Fortran both pass arguments in order from left to right. You can specify these conventions with ATTRIBUTES options such as C or STDCALL. Individual Fortran arguments can also be designated with ATTRIBUTES option VALUE or REFERENCE.

References

[1] Hibbitt, Karlsson & Sorensen Inc. *Abaqus 5.8 User's Manual Volume III*, 1998. 1

[2] E. Hinton and D.R. Owen. An introduction to finite element computations. *Pineridge Press Ltd., Swansea, U.K.*, 1979. 6

[3] T.J.R. Hughes and J. Winget. Finite rotation effects in numerical integration of rate constitutive equations arising in large-deformation analysis. *International Journal for Numerical Methods in Engineering*, 15(11):1862–1867, 1980. 5, 5

[4] P.I. Lewin and J.B. Burland. Stress-probe experiments on saturated normally consolidated clay. *Géotechnique*, 20:38–56, 1970. 7

[5] M. M. Rashid. Incremental kinematics for finite element applications. *International Journal for Numerical Methods in Engineering*, 36:3937–3956, 1993. 5

[6] P. Royis and T Doanh. Theoretical analysis of strain response envelopes using incrementally non-linear constitutive equations. *International Journal for Numerical and Analytical Methods in Geomechanics*, 22:97 – 132, 1998. 7

Index