Contents under Creative Commons BY 4.0 license and code under MIT license. © Julian Parra 2019. This material is part of the Master of Engineering program by Julian Parra at Universidad EAFIT.

User element subroutines

Introduction

In very simple terms a finite element code, regardless of its physical context, can be viewed as an assembler of element contributions to a global system of equations complemented with a solver. If the code is intended for the solution of non-linear problems the assembly and solution stages can be combined into a Newton-Raphson algorithm. In either case, the particular physical context of the problem, is enclosed in those subroutines in charge of computing the element contributions. Following the jargon of a popular multiphysics code like `ABAQUS` these specific subroutines in `NLDYNA` are also called `user subroutines` .

There are of two types of user subroutines, namely (i) user material subroutines `UMATS` and (ii) user element subroutines `UELS` . We will discuss both here.

In the particular case of stress analysis a finite element `UEL` subroutine computes the stiffness matrix and internal (or consistent) nodal loads. The computation of the stiffness matrix reduces to the numerical determination of integrals like:

$$K = \int_V B^T \frac{\partial\sigma}{\partial\varepsilon} B \,\mathrm{d}V$$

where the term:

$$C = \frac{\partial\sigma}{\partial\varepsilon}$$

is the Material Jacobian or constitutive tensor.

If the integration to compute K is conducted via Gauss quadrature, the material jacobian must be computed at every integration point. Moreover, in complex constitutive models the material jacobian is a function of a wide variety of physical parameters therefore the values of these parameters must also be available at every integration point. The process of computing C at the integration points is performed by a material user subroutine **`UMAT`.**

In this notebook we will discuss the implementation of a user element subroutine where it will be assumed that the material model is available. The specific implementation of the `UMAT` subroutine will be discussed elsewhere.

User Element Subroutine (UEL)

A so-called `user element subroutine` must be coded into `NLDYNA` to compute the contribution of the element to the full finite element model. In the most general context a user element is intended to be used in a non-linear dynamic analysis and it is expected that the user is familiar with time stepping procedures and with the general form of Newton-Raphson algorithms.

In the current non-linear dynamics context the user subroutine must define the contribution of the element to the residual vector; the contribution of the element to the jacobian (stiffness) matrix and must also form the mass and damping matrix (when needed), and update all the solution `state variables.`

At each call to the `UEL` subroutine from `NLDYNA` the main program provides the subroutine with values of the element nodal coordinates and all solution-dependent nodal variables (like displacements in elasticity), at all degrees of freedom associated with the element as well as values at the beginning of the current increment of the `solution dependent state variables` associated with the element. Tipically these `solution dependent state variables` are used in the computation of the material response.

Subroutine interface (input and output parameters)

The following set of parameters is passed from the main program to the user subroutine:

- `iele_disp:` (ndarray) Nodal displacements at time t for all the degrees of freedom of the element.
- `coord:` (ndarray) Nodal coordinates for the nodal points defining the element.
- `par:` (ndarray) Material parameters for the material profile associated to the element.
- `svar:` (ndarray) Solution dependent state variables at the beginning of the increment.

The following set of parameters is returned by the `UEL` subroutine to the main program:

- `kG:` (ndarray) Element contribution to the Jacobian (or stiffness) matrix.
- `mG:` (ndarray) Element contribution to the mass matrix (when needed).
- `cG:` (ndarray) Element contribution to the damping matrix (when needed).
- `svar:` (ndarray) Updated vector of solution dependent state variables at the end of the increment.
- `ilf:` (ndarray) Residual vector.

Bi-lineal plane strain quad element with non-linear constitutive response

The following example describes the implementation of a user element subroutine (`UEL`) for a bi-lineal 4-noded plane strain cuadrilateral element with a non-linear constitutive response. The material model (discussed elsewhere) is the plane strain elastoplastic model with combined non-linear isotropic/kinematic hardening formulated in Simo and Hughes (1998). The integration algorithm is a return mapping scheme.

The element data is divided in **(i) solution dependent nodal data** which in this case corresponds to the displacements associated to the degrees of freedom for the element and **(ii) solution dependent state variables**. This last category includes stresses and strains at the Gauss points as well as other variables required in a specific analysis or in the computation of history dependent constitutive responses.

In this particular analysis the `solution dependent state variables` is defined as follows:

- Stress tensor: This is one of the primary variables in the analysis.
- Total strain tensor: This is one of the primary variables in the analysis.
- Elastic strain tensor: Important per se and also required in the computation of the constitutive response.
- Plastic strain tensor: Required in the computation of the constitutive response.
- Back stress tensor: Required in the computation of the constitutive response in order to consider kinematic hardening.
- Equivalent plastic strain: Required in the computation of the constitutive response in order to consider isotropic hardening.
- Equivalent Misses stress: Required in the computation of the constitutive response in order to consider isotropic hardening.

In this 2D problem each tensorial variable contributes with scalar components. Since there are 5 tensorial quantities and defined at 4 Gauss integration points that will make a total of 80 state variables for the element. Also, in the computation of the isotropic hardening behavior we require the equivalent plastic strain and the corresponding equivalent stress (Misses stress). Thus, in total there are 88 state variables per element. Therefore the vector of state variables evolving back and forth bewteen the main program and the user subroutine has a total of 88 components. This vector is termed `svars` in the subroutine.

The subrotine parameters defined next.

```
Parameters
----------
coord    : ndarray
         Nodal coordinates.
props    : ndarray
         Material properties  for the element.
svars    : ndarray
         State variables array at all integration point
s.
du       : ndarray
         Nodal (incremental) displacements vector.
```

The subroutine is conformed by an external loop which conducts Gauss point computations. Let us focus in one of these computations which occur after the Gauss point coordinates and weighting factor have been retrieved.

To start these computations the state variables associated to the current Gauss point must be retrieved from the 88-positions vector `svars` . This is achieved by the `svars` handling subroutine `svarshandl` . The subroutine returns the stress and the total strain tensors at the Gauss point, while the remaining state variables at the Gauss point are stored in a local vector named `statev` . In this problem this vector will store the elastic and plastic strain tensors, the back stress tensor and the equivalent plastic strain and Misses stress. The called to the subroutine is shown below.

`svars , statev , stress , strann = svarshandl(0 , svars , statev , stress , strann , igp)`

The second relevant aspect of the subroutine is the computation of the material jacobian or constitutive tensor. This tensor is computed by the `UMAT` subroutine like

`C , stress , statev = umat_PCLK(stress , strann , dstran , statev , props , ntens)`

Note that the subroutine receives as input the local state variables at the Gauss point `statev` in addition to the stress and strain tensors and the material properties associated to the material profile. The `UMAT` subroutine returns the constitutive tensor `C` and updated values of stress, strain and state variables.

After accumulation of the stiffness matrix (and others) at the Gauss point is completed the last step is to call once again the `svars` handling subroutine `svarshandl` but now in the reverse direction, i.e., local updated state variables are stored into the global state variables vector. This is shown in the following call:

`svars , statev , stress , strann = svarshandl(1 , svars , statev , stress , strann , igp)`

In [1]:
```python
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *
from uel_solid import *
```

In [2]:
```
def uel4nquad_PLK(coord, props , svars , du):

    rho = props[6]
    nstatev = 14
    ntens  = 4
    stress = np.zeros([ntens])
    strann = np.zeros([ntens])
    dstran = np.zeros([ntens])
    statev = np.zeros([nstatev])

    kl     = np.zeros([8, 8])
    ml     = np.zeros([8, 8])
    cl     = np.zeros([8, 8])
    rhsl   = np.zeros([8])
    XW, XP = gpoints2x2()

    ngpts = 4
    for i in range(0, ngpts):
        ri  = XP[i, 0]
        si  = XP[i, 1]
        alf = XW[i]
        igp = i*22
        svars , statev , stress , strann = svarshandl(0 , svars , statev , stress , st
nn , igp)

        ddet, B = stdm4NQ(ri, si, coord)
        dstran  = np.dot(B,du)
        strann  = strann + dstran
        C , stress , statev = umat_PCLK(stress , strann , dstran , statev , props , nt
s)
        rhsl = rhsl + np.dot(B.T,stress)*alf*ddet
        kl = kl + np.dot(np.dot(B.T,C), B)*alf*ddet
        N  = sha4(ri , si )
        ml = ml + rho*np.dot(N.T , N)*alf*ddet

        svars , statev , stress , strann = svarshandl(1 , svars , statev , stress , st
nn , igp)

    return kl, ml, cl, svars, rhsl
```

Upon execution the subroutine returns the following parameters

```
    kl       : ndarray
               Element stiffness matrix
    ml       : ndarray
               Element mass matrix
    svars    : ndarray
               Updated state variables array for the element
    rhsl     : ndarray
               Consistent internal loads vector
```

To test and execute the subroutine the following block of code emulates the required entries from the main program to the element subroutine.

In [3]:
```
nprops = 7
nsvars = 88
ntens  = 4
props   = np.zeros([nprops])
svars   = np.zeros([nsvars])
du = np.zeros([8])
coord =([0.0 , 0.0], [1.0 , 0.0], [1.0 , 1.0], [0.0 , 1.0])
props[0]= 52.0e3
props[1]= 0.33
props[2]= 60.0
props[3]= 37.0
props[4]= 383.3
props[5]= 2040.0
props[6]= 1000.0
kl, ml, cl, svars, rhsl = uel4nquad_PLK(coord, props , svars , du)
```

Results

In [4]:
```
print(kl)
print(rhsl)
```

```
[[ 32198.14241486  14374.17072092 -22423.70632464   4599.73463069
  -16099.07120743 -14374.17072092   6324.6351172   -4599.73463069]
 [ 14374.17072092  32198.14241486  -4599.73463069   6324.6351172
  -14374.17072092 -16099.07120743   4599.73463069 -22423.70632464]
 [-22423.70632464  -4599.73463069  32198.14241486 -14374.17072092
    6324.6351172    4599.73463069 -16099.07120743  14374.17072092]
 [  4599.73463069   6324.6351172  -14374.17072092  32198.14241486
   -4599.73463069 -22423.70632464  14374.17072092 -16099.07120743]
 [-16099.07120743 -14374.17072092   6324.6351172   -4599.73463069
   32198.14241486  14374.17072092 -22423.70632464   4599.73463069]
 [-14374.17072092 -16099.07120743   4599.73463069 -22423.70632464
   14374.17072092  32198.14241486  -4599.73463069   6324.6351172 ]
 [  6324.6351172    4599.73463069 -16099.07120743  14374.17072092
  -22423.70632464  -4599.73463069  32198.14241486 -14374.17072092]
 [ -4599.73463069 -22423.70632464  14374.17072092 -16099.07120743
    4599.73463069   6324.6351172  -14374.17072092  32198.14241486]]
[0. 0. 0. 0. 0. 0. 0. 0.]
```

References

Simo, Juan C., and Thomas JR Hughes. Computational inelasticity. Vol. 7. Springer Science & Business Media, 2006

In [5]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[5]:

In [ ]: