```
! Copyright (C) 2007 ... 2019 Andrzej Niemunis
!
! incrementalDriver is free software; you can redistribute it and/or modify
! it under the terms of the GNU General Public License as published by
! the Free Software Foundation; either version 2 of the License, or
! (at your option) any later version.
!
! incrementalDriver is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
! GNU General Public License for more details.


! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,USA.

! August 2007: utility routines added
! Februar 2008 : *Repetition repaired again
! March 2008 : *ObeyRestrictions added
! April 2008 : command line options: test= , param= , ini= , out= , verbose= added
! November 2008: warn if divergent equil. iter. ‖u_dstress‖ too large
! February 2009: *ObeyRestrictions works with *Repetitions
! August 2010 increases output precision + keyword(3) error interception
! December 2015: fragments of niemunis_tools_lt unsymmetric_module incorporated into a single file incrementalDriver.f
! January 2016 if nstatev = 0 no statev( ) values will be read in but we set nstatv= 1 and statev(1)= 0.0d0
! October 2016 exit from step on inequality condition, import step loading data from a file, write every n-th state only
! November 2016 alignment of stress
! Jan 2017 exit on inequality corrected twice
! June 2017 undo changes in stress and state from ZERO call of umat (just for jacobian, with zero dstran and zero dtime )
! Dec 2017 parser disregards comments beyond #
! Sept 2019 c_dstran(:) = 0 in line 590 otherwise c_dstran may be used before being initialized.
! Sept 2019 random walk

! Main program that_calls_umat ( performs calculation writing to output.txt).
      PROGRAM that_calls_umat     ! written by A.Niemunis 2007 - 2019
       implicit none
       character*80  cmname,rebarn
       integer ndi,nshr,ntens,nstatv,nprops,ncrds
       integer noel,npt,layer,kspt,lrebar,kinc,i
       real(8),parameter,dimension(3,3):: delta =
     &                      reshape((/1,0,0,0,1,0,0,0,1/),(/3,3/))

      parameter(ntens=6,ndi=3,nshr=3,ncrds=3)  ! same ntens as in SOLVER
      parameter( noel=1 ,npt=1,layer=1,kspt=1,lrebar=1)
      parameter( rebarn ='xxx')
      real*8 dtime,temp,dtemp,sse,spd,scd,rpl,drpldt,pnewdt,celent
      real*8 stress(ntens),
     &  ddsdde(ntens,ntens),ddsddt(ntens),drplde(ntens),
     &  stran(ntens),dstran(ntens),time(2),predef(1),dpred(1),
     &  coords(ncrds),drot(3,3),dfgrd0(3,3),dfgrd1(3,3)
      character(len=1) :: aChar                                      ! AN 2016
      character(len=40):: keywords(10), outputfilename,
     &        parametersfilename,
     &       initialconditionsfilename, testfilename, outputfilename1,
     &       exitCond, ImportFileName,mString, keyword2,                ! AN 2016
     &       aShortLine, leftLine, rightLine
      character(len=260) ::  inputline(6), aLine, heading
      character(len=520) :: hugeLine

      character(len=10):: timeHead(2), stranHead(6), stressHead(6)      ! AN 2016
      character(len=15), allocatable :: statevHead(:)                   ! AN 2016

      logical :: verbose
      logical :: EXITNOW, existCond,okSplit                            ! AN 2016
      real(8), dimension(6,6)  :: cMt , cMe
      real(8), dimension(6)  :: mb, mbinc


      integer :: mImport, columnsInFile(7),every,ievery                ! AN 2016
      real(8) ::  importFactor(7)
      real(8),dimension(20) :: oldState, newState,dState
      real(8), allocatable :: props(:), statev(:), r_statev(:)

      real(8),dimension(3,3):: Qb33,eps33,T33

      integer:: ifstress(ntens), maxiter, ninc,kiter, ikeyword,
     &          iRepetition, nRepetitions, kStep,iStep,nSteps,ntens_in

      real(8):: r_stress(ntens),a_dstress(ntens),u_dstress(ntens),
     &         stress_Rosc(ntens),r_stress_Rosc(ntens),
     &      ddstress(ntens), c_dstran(ntens) ,
     &      deltaLoadCirc(6),phase0(6),deltaLoad(9),
     &      dstran_Cart(6), ddsdde_bar(6,6), deltaTime
      real(8),parameter :: sq3=1.7320508075688772935d0,
     &                     sq6=2.4494897427831780982d0,
     &                     sq2=1.4142135623730950488d0,
     &                     Pi  =3.1415926535897932385d0
      real(8),parameter ::
     &                     i3=0.3333333333333333333d0,
     &                     i2=0.5d0,
     &                     isq2=1/sq2,
     &                     isq3=1.0d0/sq3,
     &                     isq6=1.0d0/sq6

      real(8), parameter,dimension(1:6,1:6)::MRoscI=reshape              ! M for isomorphic Roscoe variables P,Q,Z,....
     &  ((/-isq3,-2.0d0*isq6,0.0d0,  0.0d0, 0.0d0, 0.0d0,
     &     -isq3, isq6,       -isq2,  0.0d0, 0.0d0, 0.0d0,
     &     -isq3, isq6,        isq2,  0.0d0, 0.0d0, 0.0d0,
     &      0.0d0, 0.0d0,      0.0d0, 1.0d0, 0.0d0, 0.0d0,
     &      0.0d0, 0.0d0,      0.0d0, 0.0d0, 1.0d0, 0.0d0,
     &      0.0d0, 0.0d0,      0.0d0, 0.0d0, 0.0d0, 1.0d0
     &  /),(/6,6/))

      real(8), parameter,dimension(1:6,1:6)::MRoscImT=MRoscI             ! latest M^{-T} (is orthogonal)

       real(8), parameter,dimension(1:6,1:6)::MRendul=reshape            ! M for isomorphic Rendulic sigma_11 = -T_11, sigma_22 = -(T_22 + T_33)/√(2), Z = ...
     &  ((/ -1.0d0, 0.0d0,   0.0d0,   0.0d0,0.0d0,0.0d0,
     &      0.0d0, -isq2,  -isq2,  0.0d0,0.0d0,0.0d0,
     &      0.0d0, -isq2,   isq2,  0.0d0,0.0d0,0.0d0,
     &      0.0d0,   0.0d0, 0.0d0,  1.0d0,0.0d0,0.0d0,
     &       0.0d0,   0.0d0,  0.0d0,   0.0d0,1.0d0,0.0d0,
     &       0.0d0,   0.0d0,  0.0d0,   0.0d0,0.0d0,1.0d0
     &  /),(/6,6/))

      real(8), parameter,dimension(1:6,1:6)::MRendulmT=MRendul            ! latest M^{-T} (is orthogonal)

      real(8), parameter,dimension(1:6,1:6)::MRosc=reshape                ! M for Roscoe variables p, q, z, ....
     &  ((/-i3,-1.0d0, 0.0d0,       0.0d0,0.0d0,0.0d0,
     &     -i3, i2, -1.0d0,         0.0d0,0.0d0,0.0d0,
     &      -i3,i2, 1.0d0,          0.0d0,0.0d0,0.0d0,
     &       0.0d0, 0.0d0,0.0d0,   1.0d0,0.0d0,0.0d0,
     &       0.0d0, 0.0d0, 0.0d0, 0.0d0,1.0d0,0.0d0,
     &       0.0d0, 0.0d0, 0.0d0, 0.0d0,0.0d0,1.0d0
     &  /),(/6,6/))
      real(8), parameter,dimension(1:6,1:6)::MRoscmT=reshape              ! latest M^{-T} (is not orthogonal)
     & ((/-1.0d0, -2.0d0*i3, 0.0d0,   0.0d0, 0.0d0,0.0d0,
     &     -1.0d0,    i3,        -i2,      0.0d0,0.0d0,0.0d0,
     &     -1.0d0,    i3,          i2,      0.0d0,0.0d0,0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 1.0d0, 0.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 0.0d0, 1.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0, 1.0d0
     &  /),(/6,6/))

      real(8), parameter,dimension(1:6,1:6)::MCart=reshape                ! M for Cartesian coords T_{11}, T_{22}, T_{33}, T_{12}, .....
     & ((/ 1.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0,
     &     0.0d0, 1.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 1.0d0, 0.0d0, 0.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 1.0d0, 0.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 0.0d0, 1.0d0, 0.0d0,
     &     0.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0, 1.0d0
     &  /),(/6,6/))
      real(8), parameter,dimension(1:6,1:6)::MCartmT=MCart                ! latest M^{-T} (is orthogonal)


      real(8),dimension(1:6,1:6)::M,MmT                                   ! currrent M and M^{-T} for a given iStep
      real(8) :: aux1,aux2

      type descriptionOfStep
        integer:: ninc,maxiter, ifstress(ntens),columnsInFile(7),mImport  ! AN 2016
        real(8) :: deltaLoadCirc(ntens),phase0(ntens),deltaLoad(9),
     &             dfgrd0(3,3), dfgrd1(3,3),deltaTime, importFactor(7)
        character(40) :: keyword2, keyword3, exitCond,ImportFileName      ! AN 2016
        real(8),dimension(1:6,1:6) :: cMt, cMe
        real(8),dimension(1:6) :: mbinc
        logical::existCond                                                ! AN 2016
      end type  descriptionOfStep

      type StressAlignment
        logical:: active
        character(len=40) :: ImportFileName
        integer:: kblank,nrec,kReversal, ncol
        integer,dimension(100) :: Reversal
        integer,dimension(6):: isig
        real(8),dimension(6) :: sigFac
      end type StressAlignment

      type(StressAlignment) :: align

      type(descriptionOfStep) :: ofStep(30)              ! stores descriptions of up to 30 steps which are repeated


 ! [1] read the command-line parameters to set the file names *************
       continue

       parametersfilename = 'parameters.inp'
       initialconditionsfilename  = 'initialconditions.inp'
       testfilename = 'test.inp'
       outputfilename = '--'
       verbose = .true.
       call  get_command_line_arguments()  ! command line can override the above file names


! [2] read the material parameters ***********************************************
       open(1,err=901,file=parametersfilename,status='old')
       read(1,'(a)') cmname
       i = index(cmname, '#')
       if(i == 0) then
          cmname = trim(cmname)
       else
         cmname = cmname(:i-1)
         cmname = trim(cmname)
       endif
      read(1,*) nprops
      allocate( props(nprops) )
      do i=1,nprops
         read(1,*) props(i)
      enddo
      close(1)

![3] read initial conditions and initialize everything *************************************
      open(1,err=902,file=initialconditionsfilename,status='old')
      read(1,*) ntens_in
      stress(:) = 0.0d0
      time(:) = 0.0d0
      stran(:)=0.0d0
      dtime = 0.0d0
      do i=1,ntens_in
         read(1,*) stress(i)
      enddo
      read(1,*) nstatv
      if(nstatv >= 1) then
         allocate(statev(nstatv), r_statev(nstatv), statevHead(nstatv))  ! AN 2016
         statev(:) = 0.0d0
         do i=1,nstatv
           read(1,*,end=500) statev(i)
          enddo
      else
        allocate( statev(1) , r_statev(1), statevHead(1)  )               ! AN 2016 formal placeholder not really used
         statev(:) = 0.0d0
         nstatv = 1
      endif
```

```
  500 continue
      close(1)

! ntens_in = 6
! nstatv = 300
! allocate( statev(nstatv) , r_statev(nstatv) )
! statev = 0.0d0
! call ParaelasticInitialCondition(statev) ! Loads two states into the stack

![4] read a piece from the loading path ******************************************************
      open(1,err=903,file=testfilename,status='old')
![4.1] read the outputfilename from test.inp, create/open this file and write the tablehead, heading(if any) and the first line = initial conditions
      read(1,'(a)') aLine
        i = index(aLine,'#')
        if(i==0) then
          outputfilename1=trim(aLine)
          heading = '#'
        else
          outputfilename1=trim(aLine(:i-1))
          heading = trim( aLine(i+1:) )
        endif
      if(outputfilename == '—') outputfilename = outputfilename1
      open(2,err=904,file=outputfilename)

       do i=1,2
       write(timeHead(i),'(a,i1,a)')  'time(',i, ')'
       enddo
       do i=1,6
        write( stranHead(i), '(a,i1,a)' )   'stran(',i, ')'
        write(stressHead(i), '(a,i1,a)' )  'stress(',i, ')'
       enddo
       do i=1,nstatv
       write(statevHead(i), '(a,i3,a)' )  '␣␣statev(',i, ')'
       enddo
       write(2,'(a14,500a20)') timeHead,stranHead,stressHead,statevHead


      if(heading(1:1) /= '#') write(2,*) trim(heading)
       write(2,'(500(g17.10,3h␣␣␣␣))') time+(/dtime,dtime/),
     &                                   stran, stress, statev

![4.2] loop over keywords(1) unless keyword(1) = *Repetition it is copied to keyword(2) which is the true type of loading
      kStep = 0   ! kStep = counter over all steps whereas iStep = counter over steps within a *Repetition
      do 200 ikeyword=1,10000
        read(1,'(a)',end=999) keywords(1)
        keywords(1) = trim( keywords(1) )
        if(keywords(1) == '*Repetition') then
           read(1,*) nSteps, nRepetitions
        else
           nRepetitions=1
           nSteps=1
           keywords(2) = keywords(1)
        endif

      do 130  iRepetition  = 1,nRepetitions
      do 120 iStep = 1,nSteps
        kStep = kStep + 1

        if(iRepetition > 1) then   ! recall the loading parameters of the repeated step read in during the first iRepetition
            ninc                = ofStep(istep)%ninc
            maxiter             = ofStep(istep)%maxiter
            ifstress            = ofStep(istep)%ifstress
            deltaLoadCirc       = ofStep(istep)%deltaLoadCirc
            phase0              = ofStep(istep)%phase0
            deltaLoad           = ofStep(istep)%deltaLoad
            dfgrd0              = ofStep(istep)%dfgrd0
            dfgrd1              = ofStep(istep)%dfgrd1
            deltaTime           = ofStep(istep)%deltaTime
            keywords(2)         = ofStep(istep)%keyword2
            keywords(3)         = ofStep(istep)%keyword3
            cMe                 = ofStep(istep)%cMe
            cMt                 = ofStep(istep)%cMt
            mbinc               = ofStep(istep)%mbinc
            exitCond            = ofStep(istep)%exitCond       ! AN 2016
            existCond           = ofStep(istep)%existCond      ! AN 2016
            ImportFileName      = ofStep(istep)%ImportFileName                    ! AN 2016
            mImport             = ofStep(istep)%mImport                            ! AN 2016
            columnsInFile       = ofStep(istep)%columnsInFile                      ! AN 2016 7 integers with numbers of columns (or value = 0)
            importFactor        = ofStep(istep)%importFactor                       ! AN 2016 7 real factors to be multiplied with columns
            goto 10   ! jump over reading, because reading of steps is performed only on the first loop, when iRepetition==1
        endif

        if(keywords(1) == '*Repetition') read(1,'(a)') keywords(2)           ! = LinearLoad or CirculatingLoad ...


        call splitaLine(keywords(2),'?', keywords(2),exitCond,existCond)  ! AN 2016 look for exit condition in keywords(2)

        keywords(2)  = trim(keywords(2))

        ifstress(:)=0                                                         ! default strain control
        deltaLoadCirc(:)=0.0d0                                                ! default zero step increment
        phase0(:)=0.0d0                                                       ! default no phase shift
        deltaLoad(:) = 0.0d0
        dfgrd0 = delta
        dfgrd1 = delta


         if(keywords(2) == '*DeformationGradient') then
          ! read(1,*) ninc, maxiter, deltaTime
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)     ! AN 2016
           keywords(3) = '*Cartesian'
           do i=1,9
             read(1,*)  deltaLoad(i)                                         ! dload means total change in the whole step here
           enddo
           goto 10
        endif
        if (keywords(2) == '*CirculatingLoad') then
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)          ! AN 2016 read(1,*) ninc, maxiter, deltaTime

           read(1,*) keywords(3)                                              ! = Cartesian or Roscoe or RoscoeIsomorph or Rendulic
            keywords(3)  = trim(keywords(3))
           do i=1,6
           read(1,*) ifstress(i),deltaLoadCirc(i),phase0(i),deltaLoad(i)    ! dload means amplitude here
           enddo
           goto 10
        endif
        if(keywords(2) == '*LinearLoad') then
            call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
           read(1,'(a)') keywords(3)
           do i=1,6
             read(1,*) ifstress(i), deltaLoad(i)                               ! dload means total change in the whole step here
           enddo
           goto 10
        endif

       keyword2 = keywords(2)
       if(keyword2(1:11) == '*ImportFile') then
           keywords(2) = '*ImportFile'; keyword2 = keyword2(12:)
           call splitaLine( keyword2,'|',ImportFileName,
     &                          mString,okSplit)
        if(.not.okSplit)     stop 'missing␣|␣in␣line␣*ImportFile'
           read(mString,*) mImport
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
           read(1,'(a)') keywords(3)
           columnsInFile(:) = 0; importFactor(:) = 1
           do i=1,6
             read(1,'(a)') aShortLine
             call splitaLine(aShortLine,'*',leftLine,rightLine,okSplit )
              read(leftLine,*)   ifstress(i), columnsInFile(i)
              if(okSplit)  read(rightLine,*)   ImportFactor(i)
    ! read(1,*) ifstress(i), columnsInFile(i) , ImportFactor(i) ! dload means total change in the whole step here
           enddo
           if(deltaTime <= 0) then
              read(1,'(a)') aShortLine
              call splitaLine(aShortLine,'*',leftLine,rightLine,okSplit)
              read(leftLine,*) columnsInFile(7)
              if(okSplit)  read(rightLine,*)   ImportFactor(7)
    ! read(1,*) columnsInFile(7), ImportFac(7)
           endif !deltaTime

 !*********************************************************
            call readAlignment(align, ImportFileName )
 !*********************************************************

           goto 10
        endif

        if(keywords(2) == '*OedometricE1') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          read(1,*)    deltaLoad(1)
          goto 10
        endif
        if(keywords(2) == '*OedometricS1') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          ifstress(1) = 1
          read(1,*)    deltaLoad(1)
          goto 10
        endif
        if(keywords(2) == '*TriaxialE1') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          read(1,*) deltaLoad(1)
          ifstress(2:3) = 1
          goto 10
        endif
        if(keywords(2) == '*TriaxialS1') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          read(1,*)    deltaLoad(1)
          ifstress(1:3) = 1
          goto 10
        endif
        if(keywords(2) == '*TriaxialUEq') then
          keywords(2) = '*LinearLoad'
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          keywords(3) ='*Roscoe'
          read(1,*)    deltaLoad(2)                                  ! = deviatoric strain
          goto 10
        endif
        if(keywords(2) == '*TriaxialUq') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Roscoe'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          read(1,*)   deltaLoad(2)                                   ! = deviatoric stress
          ifstress(2) = 1
        goto 10
        endif
       if(keywords(2) == '*PureRelaxation') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          goto 10
        endif
       if(keywords(2) == '*PureCreep') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Cartesian'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)       ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          ifstress(:) =  1
          goto 10
        endif
       if(keywords(2) == '*UndrainedCreep') then
          keywords(2) = '*LinearLoad'
          keywords(3) ='*Roscoe'
          call ReadStepCommons(1, ninc, maxiter,deltaTime, every)        ! AN 2016 read(1,*) ninc, maxiter, deltaTime
          ifstress(2:6) =  1
          goto 10
       endif
       if(keywords(2) == '*ObeyRestrictions') then   ! ====================== *ObeyRestrictions ==================================
            call ReadStepCommons(1, ninc, maxiter,deltaTime, every)        ! AN 2016 read(1,*) ninc, maxiter, deltaTime
            do i=1,6
              read(1,'(a)')  inputline(i)   !=== a line of form "-sd1 + sd2 + 3.0*sd3 = -10 ! a comment " is expected
              if(index(inputline(i),'=')== 0) stop 'restr␣without␣"="␣'
            enddo
            call parser(inputline, cMt,cMe,mb )
            mbinc = mb/ninc
            keywords(3) ='*Cartesian'
            ifstress(1:6) = 1                 !=== because we solve (cMt.ddsdde + cMe).dstran = mbinc for dstran
           goto 10
        endif
        if(keywords(2) == '*PerturbationsS') then
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)         ! AN 2016 read(1,*) ninc, maxiter, deltaTime
           read(1,*) keywords(3)   ! = *Rendulic or *RoscoeIsomorph
            keywords(3) = trim( keywords(3) )
           if(keywords(3) .ne. '*Rendulic' .and.
      &       keywords(3) .ne. '*RoscoeIsomorph')
      &       write(*,*) 'warning:␣non-Isomorphic␣perturburbation'
           read(1,*)  deltaLoad(1)
           ifstress(1:6) =  1
           goto 10
        endif
        if(keywords(2) == '*PerturbationsE') then
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)      ! AN 2016 read(1,*) ninc, maxiter, deltaTime
           read(1,*) keywords(3)
           keywords(3) = trim( keywords(3) )
           if(keywords(3) .ne. '*Rendulic' .and.
      &       keywords(3) .ne. '*RoscoeIsomorph')
      &       write(*,*) 'warning:␣Anisomorphic␣perturburbation'
           read(1,*) deltaLoad(1)
           goto 10
        endif

         if(keywords(2) == '*RandomWalk') then                          ! AN 2019
           call ReadStepCommons(1, ninc, maxiter,deltaTime, every)
           read(1,*) keywords(3)
           keywords(3) = trim( keywords(3) )
            do i=1,6
            read(1,*) ifstress(i),deltaLoad(i)     ! dload means max abs value of to be multiplied by random in (-1,1)
            enddo
            goto 10
        endif


        if(keywords(2) == '*End') stop '*End␣encountered␣in␣test.inp'
        write(*,*) 'error:␣unknown␣keywords(2)=',keywords(2)
        stop 'stopped␣by␣unknown␣keyword(2)␣in␣test.inp'

   10   keywords(3) = trim(keywords(3))

         if(keywords(1) == '*Repetition' .and. iRepetition == 1) then        ! remember the description of step for the next repetition
        ofStep(istep)%ninc           =     ninc
        ofStep(istep)%maxiter        =     maxiter
        ofStep(istep)%ifstress       =     ifstress
        ofStep(istep)%deltaLoadCirc  =     deltaLoadCirc
        ofStep(istep)%phase0         =     phase0
        ofStep(istep)%deltaLoad      =     deltaLoad
        ofStep(istep)%dfgrd0         =     dfgrd0
        ofStep(istep)%dfgrd1         =     dfgrd1
        ofStep(istep)%deltaTime      =     deltaTime
        ofStep(istep)%keyword2       =     keywords(2)
        ofStep(istep)%keyword3       =     keywords(3)
        ofStep(istep)%cMe            =     cMe
        ofStep(istep)%cMt            =     cMt
        ofStep(istep)%mbinc          =     mbinc
        ofStep(istep)%exitCond       =     exitCond                    ! AN 2016
        ofStep(istep)%existCond      =     existCond                    ! AN 2016
        ofStep(istep)%ImportFileName =     ImportFileName                ! AN 2016
        ofStep(istep)%mImport         =    mImport                        ! AN 2016
        ofStep(istep)%columnsInFile   =    columnsInFile                  ! AN 2016 7 integers with numbers of columns (or value = 0)
        ofStep(istep)%importFactor    =    importFactor                   ! AN 2016
        endif

       if(any(ifstress==1)) maxiter = max(maxiter,5)                         ! at least 5 iterations
       if(all(ifstress==0) .and. keywords(2) .ne. '*ObeyRestrictions')
      &                                                      maxiter = 1   ! no iterations are necessary

 ! start the current step with zero-load call of umat() just to get the stiffness
         dstran(:)=0
         dtime=0
         dtemp=0
         kinc=0
         r_statev(:)=statev(:);   r_stress(:)=stress(:)        ! AN 21.06.2017 remember the initial state and stress
       call  UMAT(stress,statev,ddsdde,sse,spd,scd,         !=== first call umat with dstrain=0 dtime=0 just for stiffness (=jacobian ddsdde)
      &     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,0,kinc)     !=== some constitutive models require kStep=0 other do not
        statev(:)=r_statev(:);   stress(:)=r_stress(:)     ! AN 21.06.2017 recover stress and state although the ZERO call of umat should not modify them




           select case( keywords(3) )
        case('*Cartesian' ) ;        M =   MCart     ;      MmT = MCartmT
        case('*Roscoe')       ;        M = MRosc   ; MmT = MRoscmT
        case('*RoscoeIsomorph');    M = MRoscI  ;   MmT = MRoscImT
        case('*Rendulic')        ;    M = MRendul;    MmT = MRendulmT
        case default  ;    write(*,*) 'Unknown␣keyword␣=␣', keywords(3)
                            stop  '␣stopped␣by␣unknown␣keywords(3)␣in␣test.inp'
       end select



         if(keywords(2) == '*ImportFile' ) then                               ! AN 2016
           open(3,file=ImportFileName, status ='old', err=905)                ! AN 2016
           do                                                                 ! AN 2016
              read(3,'(a)',err=906) hugeLine;                                 ! AN 2016
              hugeLine= adjustL(hugeLine) ; aChar = hugeLine(1:1)             ! AN 2016
              if(index('1234567890+-.',aChar) > 0) exit ! preceding non-numeric lines in ImportFile will be ignored
           enddo
           read(hugeLine,*,err=907) oldState(1:mImport)                   ! AN 2016
         endif

       iievery=1
       do 100 kinc=1,ninc

         if(keywords(2) == '*ImportFile') then                             ! AN 2016
           read(3,*,iostat=i)  newState(1:mImport)                         ! AN 2016
           if(i > 0) then                                                  ! AN 2016
           write(*,*) 'error␣Import␣file',ImportFileName, 'line=', kinc+1  ! AN 2016
           stop                                                            ! AN 2016
           endif                                                           ! AN 2016
           if(i < 0) then                                                  ! AN 2016
               close(3)                                                    ! AN 2016
               write(*,*) 'finished␣reading␣file', ImportFileName          ! AN 2016
               exit                                                        ! AN 2016
           endif                                                           ! AN 2016
           dState = newState(:) -  oldState(:)                             ! AN 2016
           do i=1,6                                                        ! AN 2016
           if  (columnsInFile(i) == 0) cycle                               ! AN 2016
           if (ifstress(i)==1  )  ddstress(i)= dState(columnsInFile(i))*
      &                                                ImportFactor(i)     ! AN 2016
           if (ifstress(i)==0)     dstran(i)  = dState(columnsInFile(i))*
      &                                               ImportFactor(i)      ! AN 2016
           enddo                                                             ! AN 2016
            if(columnsInFile(7)/= 0) deltaTime= dState(columnsInFile(7))*
      &                                                  ImportFactor(i)   ! AN 2016
            dtime = deltaTime                                             ! AN 2016
            oldState(:) = newState(:)                                    ! AN 2016
         endif                                                             ! AN 2016

         if(keywords(2) /= '*ImportFile') then                            ! AN 2016
         call get_increment(keywords, time, deltaTime, ifstress, ninc,   ! get inc. in terms of Rosc. variables
      &                           deltaLoadCirc,phase0,deltaLoad,
      &                           dtime, ddstress,  dstran, Qb33,
      &                           dfgrd0, dfgrd1,drot )    ! to be called in each increment
         endif




         a_dstress(:)= 0.0d0     ! approximated Roscoe's dstress
         r_statev(:)=statev(:)   ! remember the initial state and stress till the iteration is completed
         r_stress(:)= stress     ! remembered Cartesian stress

        do 95 kiter=1, maxiter   !-----Equilibrium Iteration-----
        c_dstran(:) = 0
        if(keywords(2)== '*ObeyRestrictions'  ) then  ! ======================= ObeyRestrictions =========
           ddsdde_bar = matmul(cMt,ddsdde) + cMe
           u_dstress = - matmul(cMt,a_dstress)-matmul(cMe,dstran)+ mbinc
           call  USOLVER(ddsdde_bar,c_dstran,u_dstress,ifstress,ntens)
           dstran = dstran + c_dstran

           call  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)

           if(kiter.lt.maxiter) then                                      ! continue iteration
              statev(:)=r_statev(:)                                       ! 1) undo the update of state (done by umat)
              a_dstress  = stress  - r_stress                             ! 2) compute the new approximation of dstress
              stress(:)=r_stress(:)                                       ! 3) undo the update of (stress done by umat)
           else
              stran(:)=stran(:)+dstran(:)                                 ! accept the updated state and stress (Cartesian)
           endif
        endif  ! ==== obey-restrictions

        if(keywords(2) /= '*ObeyRestrictions'  ) then   ! ======================= disObeyRestrictions ==========
          u_dstress = 0.0d0
          where (ifstress == 1)  u_dstress =ddstress -a_dstress           ! undesired Roscoe stress
          ddsdde_bar = matmul(matmul(M,ddsdde),transpose(M))              ! Roscoe-Roscoe stiffness

          call  USOLVER(ddsdde_bar,c_dstran,u_dstress,ifstress,ntens)     ! get Rosc. correction c_dstran() caused by undesired Rosc. dstress
          where (ifstress == 1) dstran = dstran + c_dstran                ! corrected Rosc. dstran where stress-controlled
          dstran_Cart = matmul( transpose(M),dstran )                     ! transform Rosc. to Cartesian dstran
          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)

         if (kiter.lt.maxiter) then                                       ! continue iteration
            statev(:)=r_statev(:)                                         ! 1) forget the changes of state done in umat
            stress_Rosc = matmul(M,stress)                                ! output from umat transform to Roscoe ?
            r_stress_Rosc = matmul(M,r_stress)
            where (ifstress ==1) a_dstress = stress_Rosc - r_stress_Rosc  ! 2) compute the new approximation of stress
            stress(:)=r_stress(:)                                         ! 3) forget the changes of stress done in umat
         else
           stran(:)=stran(:)+dstran_Cart(:)                               ! accept the updated state and stress (Cartesian)
         endif
       endif  ! ==== disObey-restrictions

 94    continue
       if((kiter==maxiter) .and. mod(kinc,10)==0 .and. verbose ) then     ! write to screen
       write(*,'(12H␣ikeyword␣=␣,i3,␣8H␣kstep␣=␣,i3,7H␣kinc␣=␣,i5,
 ␣␣␣␣␣&␣␣␣␣␣␣␣␣␣␣9H␣kiter␣=␣,i2)')  ikeyword, kStep, kinc, kiter
       endif

  95   continue !-----------end of Equilibrium Iteration

       aux1 =  dot_product(a_dstress,a_dstress)
       aux2 =  dot_product(u_dstress,u_dstress)
       if((aux1>1.d-10 .and. aux2/aux1 > 1.0d-2) .or.
      &    (aux1<1.d-10 .and. aux2 > 1.0d-12)  ) then
         write(*,*)
      &  'I␣cannot␣apply␣the␣prescribed␣stress␣components,'//
      & '||u_dstress||␣too␣large.'                       ! check the Rosc.stress error ¡ toler
       endif

       if(keywords(2) =='*DeformationGradient' ) then                     ! rigid rotation of stress
         T33 = map2T(stress,6)
         T33 = matmul( matmul(Qb33,T33),transpose(Qb33))
         stress=map2stress(T33,6)                                    ! rigid rotation of strain
         eps33 = map2D(stran,6)
         eps33 = matmul( matmul(Qb33,eps33),transpose(Qb33))
         stran=map2stran(eps33,6)
       endif

       where(abs(time) < 1.0d-99)    time = 0.0d0    ! prevents fortran error write 1.3E-391
       where(abs(stran) < 1.0d-99)   stran = 0.0d0
       where(abs(stress) < 1.0d-99)  stress = 0.0d0
       where(abs(statev) < 1.0d-99)  statev = 0.0d0
       if(iEvery==1) then
       write(2,'(500(g17.10,3h␣␣␣␣))') time+(/dtime,dtime/),
      &                                  stran, stress, statev
       endif
       if(keywords(2) =='*PerturbationsS' .or.
      &   keywords(2) =='*PerturbationsE' ) then ! having plotted everything undo the increment
         stran(:)=stran(:) - dstran_Cart(:)
         statev(:)=r_statev(:)
         stress(:)=r_stress(:)
       endif




       time(1)=time(1)+dtime        ! step time at the beginning of the increment
       time(2)=time(2)+dtime        ! total time at the beginning of the increment

       if( existCond ) then                                          ! AN 2016 only if a condition exists
           if( EXITNOW(exitCond, stress,stran,statev,nstatv) ) exit  ! AN 2016 depending on exitCond go to next step
       endif                                                         ! AN 2016
       iEvery = iEvery+1; if(iEvery > every) iEvery = 1

 !**************************************************
        if(keywords(2) == '*ImportFile' ) then
        call  tryAlignStress(align, kinc, newState, mImport,stress,ntens)
        endif
  !*********************************************

   100 continue   ! next kinc
   120 continue   ! next iStep
   130 continue   ! next iRepetition
   200 continue   ! next keyword




  998  stop 'End␣or␣record␣encountered␣in␣test.inp'
  999  close(1)
       close(2)
       stop 'I␣have␣reached␣end␣of␣the␣file␣test.inp'
  901  stop 'I␣cannot␣open␣the␣file␣parameters.inp'
  902  stop 'I␣cannot␣open␣the␣file␣initialconditions.inp'
  903  stop 'I␣cannot␣open␣the␣file␣test.inp'
  904  stop 'I␣cannot␣open␣the␣outputfile'
  905  stop 'I␣cannot␣open␣the␣ImportFile'
  906  stop 'Error␣reading␣ImportFile␣in␣the␣first␣non-numeric␣records␣'
  907  stop 'Error␣reading␣ImportFile␣in␣the␣first␣numeric␣record␣'

       contains !=================================================
       ! contained in program_that_calls_umat that reads the command line
       subroutine get_command_line_arguments()
       implicit none               ! ===file names in the command line override defaults
       integer :: iarg,narg, is, iargc
       integer, parameter :: argLength=40
       character(argLength) :: anArgument, argType, argValue
       narg = iargc()
       do iarg = 1,narg
         call getarg(iarg,anArgument)
         is = index(anArgument,'=')
         if(is == 0) stop 'error:␣a␣command␣line␣argument␣without␣"="␣'
         argType = anArgument(:is-1)
         argValue =  anArgument(is+1:)
         select case (argType)
         case ('param')
         parametersfilename = argValue
         case ('ini')
         initialconditionsfilename = argValue
         case ('test')
         testfilename = argValue
         case ('out')
         outputfilename = argValue
         case ('verbose')
         if (argValue == 'true') verbose = .true.
         if (argValue == 'false') verbose = .false.
         end select
       enddo
       end  subroutine get_command_line_arguments

       ! contained in program_that_calls_umat writes a 6x6 matrix for debugging with Mma
       subroutine write66(a)
       implicit none
       real(8),dimension(6,6) :: a,aT
        aT = Transpose(a)
        open(12,file='nic.m',access='append')
        write(12,'(6ha66={␣,(␣2h{␣␣,5(f15.4,2h,␣␣),f15.4,␣3h},␣))' ) aT
        close(12)
       end subroutine write66

       ! contained in program_that_calls_umat writes a 6x1 matrix for debugging with Mma
       subroutine write6(a)
       implicit none
       real(8), dimension(6) :: a
        open(12,file='nic.m',access='append')
        write(12,'(␣5hx6={␣,␣5(f15.4,2h,␣␣),f15.4,␣3h}␣␣)' ) a
        close(12)
       end subroutine write6


       ! contained in program_that_calls_umat converts D(3,3) to stran(6)
       function map2stran(a,ntens)
         implicit none                 !===converts D(3,3) to stran(6) with $\gamma_{12} = 2\epsilon_{12}$ etc.
         real(8), intent(in), dimension(1:3,1:3) :: a
         integer, intent(in) :: ntens
         real(8),  dimension(1:ntens) :: map2stran
         real(8), dimension(1:6) :: b
         b =[a(1,1),a(2,2),a(3,3),2*a(1,2),2*a(1,3),2*a(2,3)]
         map2stran(1:ntens)=b(1:ntens)
       end function map2stran

         ! contained in program_that_calls_umat converts strain rate from vector dstran(1:ntens) to D(3,3)
       function map2D(a,ntens)
         implicit none
         real(8),  dimension(1:3,1:3) :: map2D
         integer, intent(in) :: ntens
         real(8), intent(in), dimension(:) :: a
         real(8),dimension(1:6) :: b = 0
         b(1:ntens) = a(1:ntens)
         map2D = reshape( [b(1), b(4)/2, b(5)/2,
      &          b(4)/2,b(2),b(6)/2, b(5)/2,b(6)/2, b(3)],[3,3] )
       end function map2D

       ! contained in program_that_calls_umat converts tensor T(3,3) to matrix stress(ntens)
       function map2stress(a,ntens)
         implicit none
         real(8), intent(in), dimension(1:3,1:3) :: a
         integer, intent(in) :: ntens
         real(8),  dimension(1:ntens) :: map2stress
         real(8), dimension(1:6) :: b
         b = [a(1,1),a(2,2),a(3,3),a(1,2),a(1,3),a(2,3)]
         map2stress = b(1:ntens)
       end function map2stress

       ! contained in program_that_calls_umat converts matrix stress(1:ntens) to tensor T(3,3)
       function map2T(a,ntens)
         implicit none
         real(8),  dimension(1:3,1:3) :: map2T
         integer, intent(in) :: ntens
         real(8), intent(in), dimension(:) :: a
         real(8), dimension(1:6) :: b= 0
         b(1:ntens) = a(1:ntens)
         map2T = reshape( [b(1),b(4),b(5),
      &        b(4),b(2),b(6),  b(5),b(6),b(3) ],[3,3] )
       end function map2T

       ! contained in program_that_calls_umat reads a file with instructions for stress alignment
       subroutine  readAlignment(align, ImportFileName )
        implicit none
        character(len=40) ImportFileName, trunc, extension
        character(len=80) ReversalFileName
        logical ::  okSplit
        type(StressAlignment) :: align
        call splitaLine( ImportFileName ,'.',trunc, extension, okSplit )
        if(.not. okSplit) stop'error  readAlignment FileName without . '
        reversalFileName = Trim(trunc) // 'rev'
        open(22, file=reversalFileName,status ='old', err=555 )
         align%active=.True.
         align%reversal(:) = 0
         read(22,*,err=556)  align%kblank,align%nrec,align%kReversal,
      &                      align%ncol
         read(22,*,err=557)  align%reversal(1:align%kReversal)
         read(22,*,err=558)  align%isig(1:6)
         read(22,*,err=559)  align%sigFac(1:6)
        return
 555    align%active=.False.
        return
 556    stop 'error   readAlignment  cannot read kblank... '
 557    stop 'error   readAlignment  cannot read reversal() '
 558    stop 'error   readAlignment  cannot read sigCol() '
 559    stop 'error   readAlignment  cannot read factor() '
       end subroutine  readAlignment

 ! contained in program_that_calls_umat tries to align stress to values from aState(1:mImport)
        subroutine  tryAlignStress
      &             (align, kinc, aState, mImport,stress,ntens)
        implicit none
        integer:: mImport,kinc,ntens,ie
        real(8) :: aState(mImport)
        real(8) :: stress(ntens)
        type(StressAlignment) :: align

        if(.not. align%active) return
        if(.not. any(align%Reversal == kinc)) return

        ! only stress components for which isig(ie) /= 0 will be aligned
        forall(ie=1:ntens, align%isig(ie) /= 0)
      &      stress(ie)= aState( align%isig(ie))*align%sigFac(ie)
        return
       end subroutine  tryAlignStress

       end program that_calls_umat



 ! ===========================================================================
 ! basing on an input command with parameters converts deltaLoad or deltaLoadCirc
 ! to the canonical three lists: dstress(), dstrain(), ifstress()
 ! get_increment is called in each increment (and not once per step )
       subroutine get_increment(keywords, time, deltaTime,ifstress,ninc,
      &                          deltaLoadCirc,phase0,deltaLoad,
      &                          dtime, ddstress,  dstran , Qb33,
      &                          dfgrd0, dfgrd1,drot )
       implicit none
       character(40):: keywords(10)
       integer, intent(in)  :: ifstress(6),ninc
       real(8), intent(in) :: time(2), deltaTime,
      &                        deltaLoadCirc(6),phase0(6),
      &                        deltaLoad(9)
       real(8), intent(out) ::  dtime, ddstress(6), dstran(6), Qb33(3,3)
       real(8), intent(in out) ::  dfgrd0(3,3), dfgrd1(3,3), drot(3,3)


       real(8), parameter :: Pi = 3.1415926535897932385d0
       real(8),parameter,dimension(3,3):: delta =
      &                          reshape((/1,0,0,0,1,0,0,0,1/),(/3,3/))
       real(8),dimension(3,3):: Fb,Fbb, dFb,aux33,dLb,depsb,dOmegab
       real(8):: wd(6),   ! angular velocity (in future individual for each component)
      &          w0(6),   ! initial phase shift for a component
      &          t        ! step time
       real(8) :: arandom
       integer(4) :: i
       logical :: ok

       dtime =  deltaTime/ ninc
       dstran= 0
       ddstress=0
       Qb33 = delta
       drot = delta
       dfgrd0=delta
       dfgrd1=delta

       !--------------------------------
       if(keywords(2) == '*LinearLoad') then                              ! proportional loading
        do i=1,6
          if (ifstress(i)==1)   ddstress(i) = deltaLoad(i)/ ninc
          if (ifstress(i)==0)    dstran(i) = deltaLoad(i)/ ninc           ! log strain -¿ corresp. displac. inc. not constant
        enddo
        ! here dfgrd0 and dfgrd1 can be defined from stran assuming polar decomposition F=V.R with R=1 and V = exp(stran)
        ! for dfgrd0 use stran
        ! for dfgrd1 use stran-dstran
       endif
       !-----------------------------
       if(keywords(2) == '*DeformationGradient') then                     ! full deformation gradient.
       Fb = reshape((/deltaLoad(1), deltaLoad(5), deltaLoad(7),           ! finite rotations calculated after Hughes+Winget 1980
      &              deltaLoad(4), deltaLoad(2), deltaLoad(9),
      &              deltaLoad(6), deltaLoad(8), deltaLoad(3)
      &  /) ,  (/3,3/))
       Fbb = delta + (Fb-delta)*(time(1)/deltaTime)
       dfgrd0  = Fbb
       dFb = (Fb-delta)/ninc
       aux33 =  Fbb + dFb/2.0d0
       dfgrd1   = Fbb  + dFb

   ! call matrix('inverse', aux33, 3, ok )
       aux33 = inv33(aux33)
       dLb =  matmul(dFb,aux33)
       depsb = 0.5d0*(dLb + transpose(dLb))
       dstran=(/depsb(1,1), depsb(2,2),depsb(3,3),
      &          2.0d0*depsb(1,2),2.0d0*depsb(1,3),2.0d0*depsb(2,3)/)
       dOmegab =    0.5d0*(dLb - transpose(dLb))
       aux33 =  delta - 0.5d0*dOmegab
  ! call matrix('inverse', aux33, 3, ok )
       aux33 = inv33(aux33)
       Qb33 = matmul(aux33, (delta+0.5d0*dOmegab))
       drot=Qb33
       endif
       !--------------------------------
       if(keywords(2) == '*CirculatingLoad' )then                         ! harmonic oscillation
       wd(:) = 2*Pi/deltaTime
       w0 = phase0
       t= time(1)  + dtime/2    ! step time in the middle of the increment
       do i=1,6
       if(ifstress(i)==1)
      &  ddstress(i)=dtime*deltaLoadCirc(i)*wd(i)*Cos(wd(i)*t+w0(i))+
      &             deltaLoad(i)/ ninc
       if(ifstress(i)==0)dstran(i)=
      &              dtime*deltaLoadCirc(i)*wd(i)*Cos(wd(i)*t+w0(i))+
      &              deltaLoad(i)/ ninc
       enddo
        ! here dfgrd0 and dfgrd1 can be defined from stran assuming polar decomposition F=V.R with R=1 and V = exp(stran)
        ! for dfgrd0 use stran
        ! for dfgrd1 use stran-dstran
       endif

       !--------------------------------
       if(keywords(2) == '*PerturbationsS' )then
       ddstress(1)= deltaLoad(1)*cos( time(1)*2*Pi/deltaTime )
       ddstress(2)= deltaLoad(1)*sin( time(1)*2*Pi/deltaTime  )
        ! here dfgrd0 and dfgrd1 can be defined from stran assuming polar decomposition F=V.R with R=1 and V = exp(stran)
        ! for dfgrd0 use stran
        ! for dfgrd1 use stran-dstran
       endif

        !--------------------------------
       if(keywords(2) == '*PerturbationsE' )then
       dstran(1)= deltaLoad(1)*cos( time(1)*2*Pi/deltaTime )
       dstran(2)= deltaLoad(1)*sin( time(1)*2*Pi/deltaTime  )
        ! here dfgrd0 and dfgrd1 can be defined from stran assuming polar decomposition F=V.R with R=1 and V = exp(stran)
        ! for dfgrd0 use stran
        ! for dfgrd1 use stran-dstran
       endif

       if(keywords(2) == '*RandomWalk' )then
       call random_seed
         do i =1,6
         call random_number(arandom)
         if( ifstress(i)== 1) ddstress(i)= 2*(arandom-0.5d0)*deltaLoad(i)
         if( ifstress(i)== 0) dstran(i)= 2*(arandom-0.5d0)*deltaLoad(i)
         enddo
       endif

       return

       contains !=====================================================

 ! contained in get_increment inverts a 3x3 matrix
       function inv33( a )  !==================contained in get_increment
       implicit none
       real(8), dimension(3,3), intent(in) :: a
       real(8), dimension(3,3) :: b
       real(8), dimension(3,3) :: inv33
       real(8) :: det
       det = - a(1,3)*a(2,2)*a(3,1) + a(1,2)*a(2,3)*a(3,1) +
      &          a(1,3)*a(2,1)*a(3,2) - a(1,1)*a(2,3)*a(3,2) -
      &          a(1,2)*a(2,1)*a(3,3) + a(1,1)*a(2,2)*a(3,3)
       b= reshape(
      &    [-a(2,3)*a(3,2)+ a(2,2)*a(3,3), a(1,3)*a(3,2)-a(1,2)*a(3,3),
      &   -a(1,3)*a(2,2) + a(1,2)*a(2,3),  a(2,3)*a(3,1)- a(2,1)*a(3,3),
      &   -a(1,3)*a(3,1) + a(1,1)*a(3,3),  a(1,3)*a(2,1)- a(1,1)*a(2,3),
      &   -a(2,2)*a(3,1)  + a(2,1)*a(3,2), a(1,2)*a(3,1)- a(1,1)*a(3,2),
      &  -a(1,2)*a(2,1) + a(1,1)*a(2,2)]     ,[3,3]     )
       inv33 = transpose(b)/det
       end function inv33

       end subroutine get_increment



 ! Imitation of utility routine provided by abaqus for people writing umats
 ! rotates a tensor input as vector : if LSTR == 1 → stress or LSTR == 0 → strain
       SUBROUTINE ROTSIG(S,R,SPRIME,LSTR,NDI,NSHR)
       implicit none
       integer, intent(in) ::  LSTR,NDI,NSHR
       integer :: ntens
       real(8), dimension(3,3),intent(in) ::  R
       real(8), dimension(1:NDI+NSHR), intent(in) :: S
       real(8), dimension(1:NDI+NSHR) , intent(out):: SPRIME
       real(8)::  a(6), b(3,3)
       ntens = ndi+nshr
       a(:) = 0
       a(1:ntens) = S(:)
       if(LSTR==1) b = reshape( [a(1),a(4),a(5),a(4),a(2),a(6),
      &                        a(5),a(6),a(3)], [3,3] )
       if(LSTR==0) b = reshape([a(1),a(4)/2,a(5)/2,a(4)/2,a(2),a(6)/2,
      &                 a(5)/2, a(6)/2, a(3) ],[3,3] )
       b = matmul( matmul(R,b),transpose(R))
       if(LSTR==1) a = [b(1,1),b(2,2),b(3,3),b(1,2),b(1,3),b(2,3)]
       if(LSTR==0) a = [b(1,1),b(2,2),b(3,3),2*b(1,2),2*b(1,3),2*b(2,3)]
       SPRIME = a(1:ntens)
       return
       END  SUBROUTINE ROTSIG


 ! Imitation of utility routine provided by abaqus for people writing umats
 ! returns two stress invariants
       subroutine SINV(STRESS,SINV1,SINV2,NDI,NSHR)
       implicit none
       real(8),intent(in) :: STRESS(NDI+NSHR)
       real(8),intent(out) ::  SINV1,SINV2
       integer, intent(in) ::  NDI,NSHR
       real(8) :: devia(NDI+NSHR)
       real(8), parameter :: sq2 = 1.4142135623730950488d0
       if(NDI /= 3) stop 'stopped␣because␣ndi/=3␣in␣sinv'
       sinv1 = (stress(1) + stress(2) + stress(3) )/3.0d0
       devia(1:3) = stress(1:3) - sinv1
       devia(3+1:3+nshr) = stress(3+1:3+nshr) * sq2
       sinv2 = sqrt(1.5d0 *  dot_product(devia, devia)  )
       end subroutine SINV

 ! Imitation of utility routine provided by abaqus for people writing umats
 ! returns principal values if LSTR == 1 -¿ for stress or LSTR == 2 -¿ for strain
       subroutine SPRINC(S,PS,LSTR,NDI,NSHR)
       integer, intent(in) :: LSTR,NDI,NSHR
       real(8),intent(in) :: S(NDI+NSHR)
       real(8),intent(out) :: PS(NDI+NSHR)
       real(8):: A(3,3),AN(3,3)
       real(8) :: r(6)
       if(NDI /= 3) stop 'stopped␣because␣ndi/=3␣in␣sprinc'
       r(1:3) = s(1:3)
       if(LSTR == 1 .and. nshr > 0) r(4:3+nshr) = s(4:3+nshr)
       if(LSTR == 2 .and. nshr > 0) r(4:3+nshr) = s(4:3+nshr)/2
       A= reshape([r(1),r(4),r(5),r(4),r(2),r(6),r(5),r(6),r(3)],[3,3])
       call spectral_decomposition_of_symmetric(A, PS, AN, 3)
       return
       end subroutine SPRINC

 ! Imitation of utility routine provided by abaqus for people writing umats
 ! returns principal directions LSTR == 1 -¿ stress or LSTR == 2 -¿ strain
       subroutine SPRIND(S,PS,AN,LSTR,NDI,NSHR)
       implicit none
       real(8),intent(in) :: S(NDI+NSHR)
       real(8),intent(out) :: PS(3),AN(3,3)
       integer, intent(in) :: LSTR,NDI,NSHR
       real(8):: A(3,3)
       real(8) :: r(6)
       if(NDI /= 3) stop 'stopped␣because␣ndi/=3␣in␣sprind'
       r(1:3) = s(1:3)
       if(LSTR == 1 .and. nshr > 0) r(4:3+nshr) = s(4:3+nshr)
       if(LSTR == 2 .and. nshr > 0) r(4:3+nshr) = s(4:3+nshr)/2
       A= reshape([r(1),r(4),r(5),r(4),r(2),r(6),r(5),r(6),r(3)],[3,3])
       call spectral_decomposition_of_symmetric(A, PS, AN, 3)
       return
       end subroutine SPRIND

 ! Imitation of quit utility routine provided by abaqus for people writing umats
       subroutine XIT
       stop 'stopped␣because␣umat␣called␣XIT'
       end subroutine XIT

 ! used by utility routine SPRINC or SPRIND
        SUBROUTINE  spectral_decomposition_of_symmetric(A, Lam, G, n)
        implicit none
        integer, intent(in) :: n                                       ! size of the matrix
        real(8), INTENT(in)   :: A(n,n)                                ! symmetric input matrix n x n (not destroyed in this routine)
        real(8), INTENT(out)   :: Lam(n)                               ! eigenvalues
        real(8), INTENT(out)   :: G(n,n)                               ! corresponding eigenvectors in columns of G
        integer ::  iter,i, p,q
        real(8) ::   cosine, sine
        real(8), dimension(:), allocatable :: pcol ,qcol
        real(8), dimension(:,:), allocatable :: x

        allocate(pcol(n) ,qcol(n), x(n,n) )
        x = A
        G=0.0d0
        do i=1,n
        G(i,i) = 1.0d0
        enddo
       do  iter = 1,30
         call  get_jacobian_rot(x, p ,q, cosine, sine, n)                  ! find how to apply optimal similarity mapping
         call  app_jacobian_similarity(x, p,q, cosine, sine, n)            ! perform mapping

         pcol = G(:,p)                                                     ! collect rotations to global similarity matrix
         qcol = G(:,q)
         G(:,p) =   pcol*cosine - qcol*sine
         G(:,q) =   pcol* sine + qcol *cosine

         ! here write a problem-oriented accuracy test max_off_diagonal ¡ something
         ! but 30 iterations are usually ok for 3x3 stress or 6x6 stiffness matrix
       enddo

       do i=1,n
        Lam(i) = x(i,i)                                                    ! eigenvalues
       enddo
       deallocate( pcol ,qcol, x )
       return
       end

 ! used by utility routine SPRINC or SPRIND
       SUBROUTINE  app_jacobian_similarity(A, p,q, c, s, n)                ! jacobian similarity tranformation of a square symmetric matrix A
       implicit none                                                       ! ( A := G^T.A.G with Givens rotation G_pq = {{c, s}, {−s, c}} )
       INTEGER, INTENT(IN)          :: p,q                                 ! G is an identity n x n matrix overridden with values {{c,s},{-s,c}} )
       real(8), INTENT(IN)          :: c ,s                                ! in cells {{pp, pq}, {qp, qq}} algorithm according to Kielbasinski p.385
       integer , INTENT(IN)         :: n
       real(8), dimension(n,n),intent(inout) :: A
       real(8), dimension(n)  :: prow ,qrow
       real(8) :: App, Apq, Aqq

       if(p == q)  stop 'error:␣jacobian_similarity␣␣p␣==␣q'
       if(p<1 .or. p>n) stop 'error:␣jacobian_similarity␣p␣out␣of␣range'
       if(q<1 .or. q>n) stop 'error:␣jacobian_similarity␣q␣out␣of␣range'

       prow(1:n) = c*A(1:n,p) - s*A(1:n,q)
       qrow(1:n) = s*A(1:n,p) + c*A(1:n,q)
       App = c*c*A(p,p) -2*c*s*A(p,q) + s*s*A(q,q)
       Aqq =  s*s*A(p,p) +2*c*s*A(p,q) + c*c*A(q,q)
       Apq = c*s*(A(p,p) - A(q,q)) + (c*c - s*s)* A(p,q)
       A(p,1:n) =   prow(1:n)
       A(1:n,p) =   prow(1:n)
       A(q,1:n) =   qrow(1:n)
       A(1:n,q) =   qrow(1:n)
       A(p,p) =   App
       A(q,q) =   Aqq
       A(p,q) =   Apq
       A(q,p) =   Apq

       END SUBROUTINE  app_jacobian_similarity


 ! used by utility routine SPRINC or SPRIND for iterative diagonalization
       SUBROUTINE  get_jacobian_rot(A, p,q, c, s, n)           ! returns jacobian similarity tranformation param.
       implicit none                                           ! for iterative diagonalization of a square symm. A
       integer , INTENT(IN)                 :: n               ! algorithm according to Kielbasinski 385-386
       real(8), dimension(n,n),intent(in) :: A
       INTEGER, INTENT(OUT)                 :: p,q
       real(8), INTENT(OUT)                 :: c ,s
       real(8) :: App, Apq, Aqq, d, t, maxoff
       integer ::   i,j

       p = 0
       q = 0
       maxoff  = tiny(maxoff)
       do i=1,n-1
       do j=i+1,n
        if( abs(A(i,j)) > maxoff ) then
          maxoff = abs(A(i,j))
          p=i
          q=j
        endif
       enddo
       enddo
       if (p > 0) then
         App = A(p,p)
         Apq = A(p,q)
         Aqq = A(q,q)
         d = (Aqq - App)/ (2.0d0*Apq)
         t = 1.0d0/ sign(abs(d) + sqrt(1.0d0 + d*d) , d )
         c = 1.0d0/sqrt(1.0d0 + t*t)
         s = t*c
       else                                                                 ! no rotation
         p=1
         q=2
         c=1
         s=0
       endif
       end subroutine get_jacobian_rot

         subroutine ReadStepCommons(from, ninc, maxiter,deltaTime, every)
         integer, intent(in) :: from
         real(8), intent(out) :: deltaTime
         integer, intent(out) :: ninc,maxiter,every
         logical ::  okSplit
         character(Len=40)   aShortLine, leftLine, rightLine
          read(from,'(a)') aShortLine
          call splitaLine(aShortLine,':',leftLine,rightLine,okSplit )
          read(leftLine,*)  ninc, maxiter, deltaTime
           every = 1
           if(okSplit)  read(rightLine,*)  every
           if (every > ninc) every=ninc
           if (every < 1 ) every= 1
         end subroutine ReadStepCommons

 !--------------------------------------------------------------------
 ! SplitaLine gets aLinie and returns two portions left of the separator sep and right of the separator if sep is found
 ! then ok is set to .true.
       subroutine splitaLine(aLine,  sep, left,  right , ok )
       implicit none
       character(len=40), intent(in) ::  aLine
       character(len=40), intent(out):: left,  right
       character(len=40) :: tmp
       character(Len=1), intent(in):: sep

       integer:: iSep
       logical:: ok
       ok=.False.
       isep = index(aLine,sep);
       if(isep==0) then
         ok=.False.
         left = trim(adjustl( aLine))
         right = '␣␣'
       endif
       if(isep > 0) then
          ok=.True.
          tmp  =  aLine(:isep-1)
          right = aLine(isep+1:)
          left = tmp
        endif
       end subroutine splitaLine

 !------------------------------------------------------
 ! reads a condition (= string cond) and returns true if stress stran and statev satisfy this condition
 ! it is used after each increment of a step. If cond == true then the remaining increments of a step are skipped
        function EXITNOW(cond, stress,stran,statev,nstatv) !-AN 2016-------------¿
        implicit none
        integer, parameter:: ntens=6, mSummands=5
        integer, intent(in):: nstatv
        real(8), intent(in) :: stress(ntens), stran(ntens),statev(nstatv)
        character(len=40), intent(in) :: cond
        logical:: EXITNOW
        integer:: i,igt, ilt,iis,imin,iplus,iminus,Nsummands,itimes
        character(len=40) :: inp, rhs, summand(mSummands), aux
        real(8):: factor(mSummands),fac,x,y
        real(8), parameter :: sq3 = 1.7320508075689d0,
      &                         sq23 = 0.81649658092773d0


        exitnow = .False.
        igt = index(cond,'>'); ilt = index(cond,'<');iis = max(igt,ilt)        ! look for a ¡ ¿ sign
        if (iis == 0) goto 555                                                ! correct condition must contain ¡ or ¿
        inp = adjustl(cond(:iis)); rhs =  trim(adjustl( cond (iis+1:)))


       factor(1) = 1;
       if(inp(1:1) == '-') then          ! do not treat the first minus as a separator
        factor(1) = -1;  inp=inp(2:)    ! remove the first character = '-' from inp
       endif
        do i=1,mSummands  ! loop over all possible summands
                   iplus = index(inp,'+');   if(iplus==0) iplus=200         ! position of an operator in the string set to 200 if this operator is absent
                   iminus = index(inp,'-');   if(iminus==0) iminus=200
                   igt= index(inp,'>');   if(igt==0) igt=200               ! actually inp cannot contain ¿ or ¡
                   ilt= index(inp,'<');   if(ilt==0) ilt=200

                   imin = min(iplus,iminus,igt,ilt)  ! choose the first separator
                   if(imin==200)   exit                     ! no more summands encountered
                   if(imin==iplus) then                     ! separator= '+' everything left from + save as summand and positive sign for the next summand
                     summand(i) = inp(:imin-1) ; factor(i+1) = 1
                     inp = inp(imin+1:)
                   endif
                    if(imin==iminus) then  ! separator= '+' everything left from + save as summand
                     summand(i) = inp(:imin-1);   factor(i+1) = -1
                     inp = inp(imin+1:)
                    endif
                    if(imin==ilt  .or. imin == igt) then
                      summand(i) = inp(:imin-1);   exit
                    endif
        enddo
        Nsummands = i    ! last factor(i)*summand(i) was encountered before exit
        x = 0;
          do i=1,Nsummands     ! for each summand on the LHS
            aux =    adjustl( summand(i) )
            itimes = index(aux,'*')
            if(itimes /= 0) then  ! '*' exists: split the summand into factor and component
            read(aux(:itimes-1),*) fac     ! numeric factor of the summand
            factor(i) = factor(i)*fac     ! the signed numeric factor of the summand
            aux=trim(adjustl(aux(itimes+1:)))
            endif
            if(itimes == 0)  aux=trim(aux)   ! no '*' aux == component
            select case(aux)
             case ('s1');x = x + factor(i)*stress(1)
             case ('s2');x = x + factor(i)*stress(2)
             case ('s3');x = x + factor(i)*stress(3)
             case ('s12');x = x + factor(i)*stress(4)     ! AN 2020
             case ('s13');x = x + factor(i)*stress(5)     ! AN 2020
             case ('s23');x = x + factor(i)*stress(6)     ! AN 2020


             case ('v1');x = x + factor(i)*statev(1)
             case ('v2');x = x + factor(i)*statev(2)
             case ('v3');x = x + factor(i)*statev(3)
             case ('v4');x = x + factor(i)*statev(4)
             case ('v5');x = x + factor(i)*statev(5)
             case ('v6');x = x + factor(i)*statev(6)
             case ('v7');x = x + factor(i)*statev(7)
             case ('v8');x = x + factor(i)*statev(8)
             case ('v9');x = x + factor(i)*statev(9)

             case ('p');x=x-factor(i)*(stress(1)+stress(2)+stress(3))/3
             case ('q');x = x - factor(i)*(stress(1) - stress(3) )
             case ('P');x=x-factor(i)*(stress(1)+stress(2)+stress(3))/sq3
             case ('Q');x = x - factor(i)*(stress(1) - stress(3) )

             case ('e1');x = x + factor(i)*stran(1)   ! AN 2017
             case ('e2');x = x + factor(i)*stran(2)
             case ('e3');x = x + factor(i)*stran(3)
             case ('g12');x = x + factor(i)*stran(4)     ! AN 2020
             case ('g13');x = x + factor(i)*stran(5)     ! AN 2020
             case ('g23');x = x + factor(i)*stran(6)     ! AN 2020

             case ('ev');x = x - factor(i)* (stran(1)+stran(2)+stran(3))
             case ('eq'); x = x - 2* (stran(1)- stran(3))/3                        ! AN 2017
             case ('eP');x =x- factor(i)*(stran(1)+stran(2)+stran(3))/sq3
```

```
               case ('eQ'); x = x - factor(i)* sq23* (stran(1)- stran(3))
               case DEFAULT; goto 555
             end select
          enddo

        read(rhs,*) y
        igt = index(cond,'>'); ilt = index(cond,'<')
        if(igt /= 0)    exitnow  = ( x > y )
        if(ilt /= 0)    exitnow  = ( x < y )
        return
555     write(*,*) 'inp␣syntax␣error:␣',cond,'␣exit␣condition␣ignored'
        EXITNOW = .False.
        end function  EXITNOW      !¡–AN 2016——————————

!——————————————————————————————————————————————
! used to read test.inp when the option *ObeyRestrictions is used
        subroutine  PARSER(inputline, Mt,Me,mb)
        implicit none
        character(260), intent(in) ::  inputline(6)
        real(8), dimension(6,6), intent(out) :: Mt , Me
        real(8), dimension(6),intent(out) :: mb

        character(len=260) ::  inp, aux,aux3
        character(40) ::  summand(13)
        integer :: iis,i,iplus,iminus,iequal,imin,iex,itimes,Irestr,ihash,
       &              Nsummands
        real(8) :: factor(13),fac


         Mt = 0; Me= 0; mb= 0

              Do Irestr = 1,6  ! Irestr loop over restriction lines

                    inp = trim(adjustl(inputline(Irestr)))
                    ihash = index(inp,'#')
                    if(ihash /= 0) inp = inp(:ihash-1)
                    iis = index(inp,'=')
                    if(iis==0) stop 'parser␣error:␣no␣=␣in␣restriction'

                    factor(1) =1;
                    if(inp(1:1) == '-') then ! do not treat the first minus as a separator
                      factor(1) = -1
                      inp=inp(2:)                  ! remove the first character = '-' from inp
                    endif
                 do i=1,13  ! loop over possible summands
                     iplus = index(inp,'+');   if(iplus==0) iplus=200
                     iminus = index(inp,'-');   if(iminus==0) iminus=200
                     iequal = index(inp,'=');   if(iequal==0) iequal=200
                     imin = min(iplus,iminus,iequal) ! choose the first separator
                     if(imin==200)   stop 'parser␣err:␣no␣+,-,=␣in␣restric'
                     if(imin==iplus) then ! separator= '+' everything left from + save as summand
                       summand(i) = inp(:imin-1) ; factor(i+1) = 1
                       inp = inp(imin+1:)
                     endif
                     if(imin==iminus) then ! separator= '+' everything left from + save as summand
                       summand(i) = inp(:imin-1);   factor(i+1) = -1
                       inp = inp(imin+1:)
                     endif
                     if(imin==iequal) then ! separator= '=' everything left from + save as summand
                       summand(i) = inp(:imin-1);
                       inp = inp(imin+1:)   ! rhs possibly with sign
                       iminus = index(inp,'-');   if(iminus==0) iminus=200
                       iex = index(inp,'!');    if(iex==0) iex=len(inp)+1   ! right limit = comment or EOL
                       if(iminus == 200) then       ! '=' is not followed by '-'
                         factor(i+1) = 1
                         summand(i+1) = inp(:iex-1)
                       else                             ! double separator: '=' followed by '-'
                         factor(i+1) = -1
                         summand(i+1) = inp(iminus+1:iex-1)
                       endif
                       exit    ! reading a single summand after '=' ends reading of the line
                     endif
                 enddo ! i-loop
          Nsummands=i+1   ! summand()=LHS, summand(Nsummands)=RHS, signs in factor()

             Do i=1,Nsummands-1   ! for summands on the LHS
              aux =   adjustl( summand(i) )
              itimes = index(aux,'*')
              if(itimes /= 0) then                  ! if exists '*' then split the summand into factor and component
                  read(aux(:itimes-1),*) fac      ! numeric factor of the summand ———————— TODO it need not be a number it can be a stress s1,s2,s3,s4,s5,s6
                  factor(i) = factor(i)*fac       ! the signed numeric factor of the summand
                  aux=adjustl(aux(itimes+1:))
              endif
              aux3 = aux(1:3)
              select case(aux3)
               case ('sd1') ;   Mt(Irestr,1) = factor(i)
               case ('sd2') ;   Mt(Irestr,2) = factor(i)
               case ('sd3') ;   Mt(Irestr,3) = factor(i)
               case ('sd4') ;   Mt(Irestr,4) = factor(i)
               case ('sd5') ;   Mt(Irestr,5) = factor(i)
               case ('sd6') ;   Mt(Irestr,6) = factor(i)
               case ('ed1') ;   Me(Irestr,1) = factor(i)
               case ('ed2') ;   Me(Irestr,2) = factor(i)
               case ('ed3') ;   Me(Irestr,3) = factor(i)
               case ('ed4') ;   Me(Irestr,4) = factor(i)
               case ('ed5') ;   Me(Irestr,5) = factor(i)
               case ('ed6') ;   Me(Irestr,6) = factor(i)
              end select
          enddo
          read(summand(Nsummands) ,*) mb(Irestr)      ! RHS numeric without sign
          mb(Irestr) =  mb(Irestr)*factor(Nsummands)   ! RHS numeric with sign
        enddo ! Irestr
        end subroutine PARSER

! solver for unsymmetric matrix and unknowns on both sides of equation
        subroutine USOLVER(KK,u,rhs,is,ntens) ! 23.7.2008 new usolver with improvement after numerical recipes
! KK - stiffness is not spoiled within the subroutine
! u - strain rhs - stress
! is(i)= 1 means rhs(i) is prescribed,
! is(i)= 0 means u(i) is prescribed

        implicit none
```

```fortran
      integer,  intent(in):: ntens
      integer,  dimension(1:ntens),  intent(in):: is
      real(8),  dimension(1:ntens,1:ntens),  intent(in):: KK
      real(8),  dimension(1:ntens),  intent(inout)::   u,rhs
      real(8),  dimension(1:ntens):: rhs1
      real(8),  allocatable :: rhsPrim(:),  KKprim(:,:),  uprim(:)
      integer  ::   i,j,ii,nis
      integer,allocatable  ::  is1(:)

      nis = sum(is)                                                      ! number of prescribed stress components

      if  (all(  is(1:ntens)== 0)  ) then
      rhs =  matmul(KK,u)
      return
      endif

      if  (all(is(1:ntens) == 1)) then                                   ! a special case with full stress control
       u =xLittleUnsymmetricSolver(KK,rhs)
      return
      endif

      rhs1 = rhs   ! modify the rhs to rhs1
      do i=1,ntens
      if  (is(i) == 0)  rhs1 = rhs1 - u(i)*KK(:,i)                       ! modify rhs wherever strain control
      enddo

      allocate(KKprim(nis,nis),  rhsprim(nis),  uprim(nis),  is1(nis))   ! re-dimension stiffness and rhs

      ii=0
      do i=1,ntens
        if(is(i)==1) then
          ii = ii+1
          is1(ii) = i                                                     ! list with positions of is(i) == 1
        endif
      enddo


      do i=1,nis
      rhsPrim(i) = rhs1(  is1(i)  )
      do j=1,nis
      KKprim(i,j) =  KK(is1(i),is1(j))
      enddo
      enddo

      if  (nis ==1) uprim = rhsprim / KKprim(1,1)
      if  (nis > 1) uprim =xLittleUnsymmetricSolver(KKprim,rhsprim)
      do i=1,nis
           u(is1(i)) = uprim(i)
      enddo
      do i=1,ntens
        if  (  is(i) == 0  ) rhs(i) = dot_product( KK(i,:),  u)          ! calculate rhs where u prescribed
      enddo
      deallocate(KKprim,rhsprim,uprim,is1)


      CONTAINS  !==================================================

! contained in USOLVER LU-decomposition from NR
      SUBROUTINE ludcmp(a,indx,d)
      IMPLICIT NONE
      REAL(8), DIMENSION(:,:), INTENT(INOUT)  :: a
      INTEGER, DIMENSION(:), INTENT(OUT)  :: indx
      REAL(8), INTENT(OUT)  :: d
      REAL(8), DIMENSION(size(a,1))  :: vv  ,aux
      integer, dimension(1)  :: imaxlocs
      REAL(8), PARAMETER :: TINY=1.0d-20
      INTEGER  :: j,n,imax
      n = size(a,1)
      d=1.0
      vv=maxval(abs(a),dim=2)
      if  (any(vv == 0.0))  stop  'singular matrix in ludcmp'
      vv=1.0d0/vv
      do j=1,n
      imaxlocs=maxloc(   vv(j:n)*abs( a(j:n,j) ) )
      imax=(j-1)+imaxlocs(1)
      if  (j /= imax) then
           aux = a(j,:)          ! call swap(a(imax,:),a(j,:))
           a(j,:) = a(imax,:)
           a(imax,:) = aux
       d=-d
       vv(imax)=vv(j)
      end if
      indx(j)=imax
      if  (a(j,j) == 0.0)  a(j,j)=TINY
      a(j+1:n,j)=a(j+1:n,j)/a(j,j)
      a(j+1:n,j+1:n)=a(j+1:n,j+1:n)- spread(a(j+1:n,j),2,n-j )*
     &                   spread(a(j,j+1:n),1, n-j)    ! outerprod
      end do
      END SUBROUTINE ludcmp

! contained in USOLVER LU-back substitution from NR
      SUBROUTINE lubksb(a,indx,b)
      IMPLICIT NONE
      REAL(8), DIMENSION(:,:), INTENT(IN)  :: a
      INTEGER, DIMENSION(:), INTENT(IN)  :: indx
      REAL(8), DIMENSION(:), INTENT(INOUT)  :: b
      INTEGER :: i,n,ii,ll
      REAL(8)  :: summ
      n=size(a,1)
       ii=0
      do i=1,n
        ll=indx(i)
       summ=b(ll)
       b(ll)=b(i)
        if  (ii /= 0) then
         summ=summ-dot_product(a(i,ii:i-1),b(ii:i-1))
        else if (summ /= 0.0) then
          ii=i
       end if
       b(i)=summ
      end do
      do i=n,1,-1
```

```
          b(i) = (b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/a(i,i)
        end do
        END SUBROUTINE lubksb

! contained in USOLVER improvement of the accuracy
        SUBROUTINE mprove(a,alud,indx,b,x)
        IMPLICIT NONE
        REAL(8), DIMENSION(:,:), INTENT(IN) :: a,alud
        INTEGER, DIMENSION(:), INTENT(IN) :: indx
        REAL(8), DIMENSION(:), INTENT(IN) :: b
        REAL(8), DIMENSION(:), INTENT(INOUT) :: x
        REAL(8), DIMENSION(size(a,1)) :: r
        r=matmul(a,x)-b
        call lubksb(alud,indx,r)
        x=x-r
        END SUBROUTINE mprove

! solver contained in USOLVER for problems with unknowns on the left-hand side
        function xLittleUnsymmetricSolver(a,b)
        IMPLICIT NONE                  !==== solves a.x = b & doesn't spoil a or b
        REAL(8), DIMENSION(:), intent(inout) :: b
        REAL(8), DIMENSION(:,:), intent(in) ::  a
        REAL(8), DIMENSION(size(b,1)) :: x
        REAL(8), DIMENSION(size(b,1),size(b,1)) ::  aa
        INTEGER, DIMENSION(1:size(b,1)) :: indx
        real(8), DIMENSION(1:size(b,1)):: xLittleUnsymmetricSolver
        REAL(8) :: d
        x(:)=b(:)
        aa(:,:)=a(:,:)
        call ludcmp(aa,indx,d)
        call lubksb(aa,indx,x)
        call mprove(a,aa,indx,b,x)
        xLittleUnsymmetricSolver= x(:)
        end function xLittleUnsymmetricSolver


        end subroutine USOLVER



        subroutine stopp(i, whyStopText)                       ! AN 2016
        USE ISO_FORTRAN_ENV   !, ONLY : ERROR_UNIT ! AN 2016
        implicit none                                          ! AN 2016
        integer, intent(in) :: i                               ! AN 2016
        character(*)       :: whyStopText                      ! AN 2016
        stop   'whyStopText'                                   ! AN 2016
         WRITE(ERROR_UNIT,*)    whyStopText                    ! AN 2016
        CALL EXIT(5)                                           ! AN 2016
        end subroutine stopp                                   ! AN 2016
```