-------- Original Message --------
Subject: inaccuracies in attenuation calculations (at least in the 2D code)
Date: Wed, 28 Sep 2011 12:51:31 +0200
From: Dimitri Komatitsch
Organization: University of Toulouse, France
To: Jeroen Tromp

Hi all,

Roland Martin has noticed the following problem in the 2D code, in the section in which we march the attenuation memory variables (see the two red boxes in the PDF file attached): the two gradients, which are supposed to be computed at time t_n and t_{n+1} respectively, are in fact exactly the same in the current version of the code because "displ_elastic" is not updated in that routine.

(for the displacement vector in the explicit Newmark scheme the displacement predictor is equal to the displacement updated at time t_{n+1} because the corrector is zero, see equation 9.1.9 in the other PDF file attached).

This could explain the small inaccuracies that we observe for very long runs with attenuation: the gradient of displacement is not very accurately computed (and is not consistent with the Runge-Kutta RK4 scheme used for the memory variables).

We should fix that at some point (unfortunately I have no time for now). We should also check if this is true in the 3D codes as well (I guess so, but I have no time to check).

Thanks,
Dimitri.

--
Dimitri Komatitsch
Professor, University of Toulouse and Institut universitaire de France,
CNRS UMR 5563 GET, France http://www.univ-pau.fr/~dkomati1

```
!========================================================================
!
!                   S P E C F E M 2 D  Version 6 . 2
!                   --------------------------------
!
! Copyright Universite de Pau, CNRS and INRIA, France,
! and Princeton University / California Institute of Technology, USA.
! Contributors: Dimitri Komatitsch, dimitri DOT komatitsch aT univ-pau DOT fr
!               Nicolas Le Goff, nicolas DOT legoff aT univ-pau DOT fr
!               Roland Martin, roland DOT martin aT univ-pau DOT fr
!               Christina Morency, cmorency aT princeton DOT edu
!
! This software is a computer program whose purpose is to solve
! the two-dimensional viscoelastic anisotropic or poroelastic wave equation
! using a spectral-element method (SEM).
!
! This software is governed by the CeCILL license under French law and
! abiding by the rules of distribution of free software. You can use,
! modify and/or redistribute the software under the terms of the CeCILL
! license as circulated by CEA, CNRS and INRIA at the following URL
! "http://www.cecill.info".
!
! As a counterpart to the access to the source code and rights to copy,
! modify and redistribute granted by the license, users are provided only
! with a limited warranty and the software's author, the holder of the
! economic rights, and the successive licensors have only limited
! liability.
!
! In this respect, the user's attention is drawn to the risks associated
! with loading, using, modifying and/or developing or reproducing the
! software by the user in light of its specific status of free software,
! that may mean that it is complicated to manipulate, and that also
! therefore means that it is reserved for developers and experienced
! professionals having in-depth computer knowledge. Users are therefore
! encouraged to load and test the software's suitability as regards their
! requirements in conditions enabling the security of their systems and/or
! data to be ensured and, more generally, to use and operate it in the
! same conditions as regards security.
!
! The full text of the license is available in file "LICENSE".
!
!========================================================================

subroutine compute_forces_viscoelastic(p_sv,nglob,nspec,myrank,nelemabs,numat, &
     ispec_selected_source,ispec_selected_rec,is_proc_source,which_proc_receiver, &
     source_type,it,NSTEP,anyabs,assign_external_model, &
     initialfield,TURN_ATTENUATION_ON,angleforce,deltatcube, &
     deltatfourth,twelvedeltat,fourdeltatsquare,ibool,kmato,numabs,elastic,codeabs, &
     accel_elastic,veloc_elastic,displ_elastic,b_accel_elastic,b_displ_elastic, &
     density,poroelastcoef,xix,xiz,gammax,gammaz, &
     jacobian,vpext,vsext,rhoext,c11ext,c13ext,c15ext,c33ext,c35ext,c55ext,anisotropic,anisotropy, &
     source_time_function,sourcearray,adj_sourcearrays,e1,e11, &
     e13,dux_dxl_n,duz_dzl_n,duz_dxl_n,dux_dzl_n, &
     dux_dxl_np1,duz_dzl_np1,duz_dxl_np1,dux_dzl_np1,hprime_xx,hprimewgll_xx, &
     hprime_zz,hprimewgll_zz,wxgll,wzgll,inv_tau_sigma_nu1,phi_nu1,inv_tau_sigma_nu2,phi_nu2,Mu_nu1,Mu_nu2,N_SLS, &
     deltat,coord,add_Bielak_conditions, &
```



```
     x0_source, z0_source, A_plane, B_plane, C_plane, angleforce_refl, c_inc, c_refl, time_offset,f0, &
     v0x_left,v0z_left,v0x_right,v0z_right,v0x_bot,v0z_bot,t0x_left,t0z_left,t0x_right,t0z_right,t0x_bot,t0z_bot,&
     nleft,nright,nbot,over_critical_angle,NSOURCES,nrec,SIMULATION_TYPE,SAVE_FORWARD,b_absorb_elastic_left,&
     b_absorb_elastic_right,b_absorb_elastic_bottom,b_absorb_elastic_top,nspec_left,nspec_right,&
     nspec_bottom,nspec_top,ib_left,ib_right,ib_bottom,ib_top,mu_k,kappa_k)

  ! compute forces for the elastic elements

  implicit none

  include "constants.h"

  logical :: p_sv
  integer :: NSOURCES, i_source
  integer :: nglob,nspec,myrank,nelemabs,numat,it,NSTEP
  integer, dimension(NSOURCES) :: ispec_selected_source,is_proc_source,source_type

  integer :: nrec,SIMULATION_TYPE
  integer, dimension(nrec) :: ispec_selected_rec,which_proc_receiver
  integer :: nspec_left,nspec_right,nspec_bottom,nspec_top
  integer, dimension(nelemabs) :: ib_left
  integer, dimension(nelemabs) :: ib_right
  integer, dimension(nelemabs) :: ib_bottom
  integer, dimension(nelemabs) :: ib_top

  logical :: anyabs,assign_external_model,initialfield,TURN_ATTENUATION_ON,add_Bielak_conditions

  logical :: SAVE_FORWARD

  double precision :: deltatcube,deltatfourth,twelvedeltat,fourdeltatsquare
  double precision, dimension(NSOURCES) :: angleforce

  integer, dimension(NGLLX,NGLLZ,nspec) :: ibool
  integer, dimension(nspec) :: kmato
  integer, dimension(nelemabs) :: numabs

  logical, dimension(nspec) :: elastic,anisotropic
  logical, dimension(4,nelemabs)  :: codeabs

  real(kind=CUSTOM_REAL), dimension(3,nglob) :: accel_elastic,veloc_elastic,displ_elastic
  double precision, dimension(2,numat) :: density
  double precision, dimension(4,3,numat) :: poroelastcoef
  double precision, dimension(6,numat) :: anisotropy
  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLZ,nspec) :: xix,xiz,gammax,gammaz,jacobian
  double precision, dimension(NGLLX,NGLLZ,nspec) :: vpext,vsext,rhoext
  double precision, dimension(NGLLX,NGLLZ,nspec) ::  c11ext,c15ext,c13ext,c33ext,c35ext,c55ext

  real(kind=CUSTOM_REAL), dimension(NSOURCES,NSTEP) :: source_time_function
  real(kind=CUSTOM_REAL), dimension(NSOURCES,NDIM,NGLLX,NGLLZ) :: sourcearray

  real(kind=CUSTOM_REAL), dimension(3,nglob) :: b_accel_elastic,b_displ_elastic
  real(kind=CUSTOM_REAL), dimension(nrec,NSTEP,3,NGLLX,NGLLZ) :: adj_sourcearrays
```

```
  real(kind=CUSTOM_REAL), dimension(nglob) :: mu_k,kappa_k
  real(kind=CUSTOM_REAL), dimension(3,NGLLZ,nspec_left,NSTEP) :: b_absorb_elasti
c_left
  real(kind=CUSTOM_REAL), dimension(3,NGLLZ,nspec_right,NSTEP) :: b_absorb_elast
ic_right
  real(kind=CUSTOM_REAL), dimension(3,NGLLX,nspec_top,NSTEP) :: b_absorb_elastic
_top
  real(kind=CUSTOM_REAL), dimension(3,NGLLX,nspec_bottom,NSTEP) :: b_absorb_elas
tic_bottom

  integer :: N_SLS
  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLZ,nspec,N_SLS) :: e1,e11,e13
  double precision, dimension(NGLLX,NGLLZ,nspec,N_SLS) :: inv_tau_sigma_nu1,phi_
nu1,inv_tau_sigma_nu2,phi_nu2
  double precision, dimension(NGLLX,NGLLZ,nspec) :: Mu_nu1,Mu_nu2
  real(kind=CUSTOM_REAL) :: e1_sum,e11_sum,e13_sum
  integer :: i_sls

  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLZ,nspec) :: &
       dux_dxl_n,duz_dzl_n,duz_dxl_n,dux_dzl_n,dux_dxl_np1,duz_dzl_np1,duz_dxl_n
p1,dux_dzl_np1

  ! derivatives of Lagrange polynomials
  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLX) :: hprime_xx,hprimewgll_xx
  real(kind=CUSTOM_REAL), dimension(NGLLZ,NGLLZ) :: hprime_zz,hprimewgll_zz

  ! Gauss-Lobatto-Legendre weights
  real(kind=CUSTOM_REAL), dimension(NGLLX) :: wxgll
  real(kind=CUSTOM_REAL), dimension(NGLLZ) :: wzgll


  !---
  !--- local variables
  !---

  integer :: ispec,i,j,k,iglob,ispecabs,ibegin,iend,irec,irec_local

  ! spatial derivatives
  real(kind=CUSTOM_REAL) :: dux_dxi,dux_dgamma,duy_dxi,duy_dgamma,duz_dxi,duz_dg
amma
  real(kind=CUSTOM_REAL) :: dux_dxl,duy_dxl,duz_dxl,dux_dzl,duy_dzl,duz_dzl
  real(kind=CUSTOM_REAL) :: b_dux_dxi,b_dux_dgamma,b_duy_dxi,b_duy_dgamma,b_duz_
dxi,b_duz_dgamma
  real(kind=CUSTOM_REAL) :: b_dux_dxl,b_duy_dxl,b_duz_dxl,b_dux_dzl,b_duy_dzl,b_
duz_dzl
  real(kind=CUSTOM_REAL) :: dsxx,dsxz,dszz
  real(kind=CUSTOM_REAL) :: b_dsxx,b_dsxz,b_dszz
  real(kind=CUSTOM_REAL) :: sigma_xx,sigma_xy,sigma_xz,sigma_zy,sigma_zz
  real(kind=CUSTOM_REAL) :: b_sigma_xx,b_sigma_xy,b_sigma_xz,b_sigma_zy,b_sigma_
zz
  real(kind=CUSTOM_REAL) :: nx,nz,vx,vy,vz,vn,rho_vp,rho_vs,tx,ty,tz,weight,xxi,
zxi,xgamma,zgamma,jacobian1D

  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLZ) :: tempx1,tempx2,tempy1,tempy2,
tempz1,tempz2
  real(kind=CUSTOM_REAL), dimension(NGLLX,NGLLZ) :: b_tempx1,b_tempx2,b_tempy1,b
_tempy2,b_tempz1,b_tempz2

  ! Jacobian matrix and determinant
  real(kind=CUSTOM_REAL) :: xixl,xizl,gammaxl,gammazl,jacobianl

  ! material properties of the elastic medium
```



```
  real(kind=CUSTOM_REAL) :: mul_relaxed,lambdal_relaxed,lambdalplus2mul_relaxed,
kappal,cpl,csl,rhol, &
       lambdal_unrelaxed,mul_unrelaxed,lambdalplus2mul_unrelaxed

  ! for attenuation
  real(kind=CUSTOM_REAL) :: Un,Unp1,tauinv,Sn,Snp1,theta_n,theta_np1,tauinvsquar
e,tauinvcube,tauinvUn

  ! for anisotropy
  double precision ::  c11,c15,c13,c33,c35,c55

  ! for analytical initial plane wave for Bielak's conditions
  double precision :: veloc_horiz,veloc_vert,dxUx,dzUx,dxUz,dzUz,traction_x_t0,t
raction_z_t0,deltat
  double precision, dimension(NDIM,nglob), intent(in) :: coord
  double precision x0_source, z0_source, angleforce_refl, c_inc, c_refl, time_of
fset, f0
  double precision, dimension(NDIM) :: A_plane, B_plane, C_plane
  !over critical angle
  logical :: over_critical_angle
  integer :: nleft, nright, nbot
  double precision, dimension(nleft) :: v0x_left,v0z_left,t0x_left,t0z_left
  double precision, dimension(nright) :: v0x_right,v0z_right,t0x_right,t0z_right
  double precision, dimension(nbot) :: v0x_bot,v0z_bot,t0x_bot,t0z_bot
  integer count_left,count_right,count_bottom

  integer :: ifirstelem,ilastelem

  ! compute Grad(displ_elastic) at time step n for attenuation
  if(TURN_ATTENUATION_ON) then
     call compute_gradient_attenuation(displ_elastic,dux_dxl_n,duz_dxl_n, &
          dux_dzl_n,duz_dzl_n,xix,xiz,gammax,gammaz,ibool,elastic,hprime_xx,hpri
me_zz,nspec,nglob)
  endif

  ifirstelem = 1
  ilastelem = nspec

  ! loop over spectral elements
  do ispec = ifirstelem,ilastelem

     tempx1(:,:) = ZERO
     tempy1(:,:) = ZERO
     tempz1(:,:) = ZERO
     tempx2(:,:) = ZERO
     tempy2(:,:) = ZERO
     tempz2(:,:) = ZERO
     if(SIMULATION_TYPE ==2)then
        b_tempx1(:,:) = ZERO
        b_tempy1(:,:) = ZERO
        b_tempz1(:,:) = ZERO
        b_tempx2(:,:) = ZERO
        b_tempy2(:,:) = ZERO
        b_tempz2(:,:) = ZERO
     endif

     !---
     !--- elastic spectral element
     !---
     if(elastic(ispec)) then

        ! get relaxed elastic parameters of current spectral element
```

```
        lambdal_relaxed = poroelastcoef(1,1,kmato(ispec))
        mul_relaxed = poroelastcoef(2,1,kmato(ispec))
        lambdalplus2mul_relaxed = poroelastcoef(3,1,kmato(ispec))

        ! first double loop over GLL points to compute and store gradients
        do j = 1,NGLLZ
           do i = 1,NGLLX

              !--- if external medium, get elastic parameters of current grid po
int
              if(assign_external_model) then
                 cpl = vpext(i,j,ispec)
                 csl = vsext(i,j,ispec)
                 rhol = rhoext(i,j,ispec)
                 mul_relaxed = rhol*csl*csl
                 lambdal_relaxed = rhol*cpl*cpl - TWO*mul_relaxed
                 lambdalplus2mul_relaxed = lambdal_relaxed + TWO*mul_relaxed
              endif

              ! derivative along x and along z
              dux_dxi = ZERO
              duy_dxi = ZERO
              duz_dxi = ZERO

              dux_dgamma = ZERO
              duy_dgamma = ZERO
              duz_dgamma = ZERO

              if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward wave
field
                 b_dux_dxi = ZERO
                 b_duy_dxi = ZERO
                 b_duz_dxi = ZERO

                 b_dux_dgamma = ZERO
                 b_duy_dgamma = ZERO
                 b_duz_dgamma = ZERO
              endif

              ! first double loop over GLL points to compute and store gradients
              ! we can merge the two loops because NGLLX == NGLLZ
              do k = 1,NGLLX
                 dux_dxi = dux_dxi + displ_elastic(1,ibool(k,j,ispec))*hprime_xx
(i,k)
                 duy_dxi = duy_dxi + displ_elastic(2,ibool(k,j,ispec))*hprime_xx
(i,k)
                 duz_dxi = duz_dxi + displ_elastic(3,ibool(k,j,ispec))*hprime_xx
(i,k)
                 dux_dgamma = dux_dgamma + displ_elastic(1,ibool(i,k,ispec))*hpr
ime_zz(j,k)
                 duy_dgamma = duy_dgamma + displ_elastic(2,ibool(i,k,ispec))*hpr
ime_zz(j,k)
                 duz_dgamma = duz_dgamma + displ_elastic(3,ibool(i,k,ispec))*hpr
ime_zz(j,k)

                 if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward w
avefield
                    b_dux_dxi = b_dux_dxi + b_displ_elastic(1,ibool(k,j,ispec))*
hprime_xx(i,k)
                    b_duy_dxi = b_duy_dxi + b_displ_elastic(2,ibool(k,j,ispec))*
hprime_xx(i,k)
                    b_duz_dxi = b_duz_dxi + b_displ_elastic(3,ibool(k,j,ispec))*
```



```
hprime_xx(i,k)
                    b_dux_dgamma = b_dux_dgamma + b_displ_elastic(1,ibool(i,k,is
pec))*hprime_zz(j,k)
                    b_duy_dgamma = b_duy_dgamma + b_displ_elastic(2,ibool(i,k,is
pec))*hprime_zz(j,k)
                    b_duz_dgamma = b_duz_dgamma + b_displ_elastic(3,ibool(i,k,is
pec))*hprime_zz(j,k)
                 endif
              enddo

              xixl = xix(i,j,ispec)
              xizl = xiz(i,j,ispec)
              gammaxl = gammax(i,j,ispec)
              gammazl = gammaz(i,j,ispec)

              ! derivatives of displacement
              dux_dxl = dux_dxi*xixl + dux_dgamma*gammaxl
              dux_dzl = dux_dxi*xizl + dux_dgamma*gammazl

              duy_dxl = duy_dxi*xixl + duy_dgamma*gammaxl
              duy_dzl = duy_dxi*xizl + duy_dgamma*gammazl

              duz_dxl = duz_dxi*xixl + duz_dgamma*gammaxl
              duz_dzl = duz_dxi*xizl + duz_dgamma*gammazl

              if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward wave
field
                 b_dux_dxl = b_dux_dxi*xixl + b_dux_dgamma*gammaxl
                 b_dux_dzl = b_dux_dxi*xizl + b_dux_dgamma*gammazl

                 b_duy_dxl = b_duy_dxi*xixl + b_duy_dgamma*gammaxl
                 b_duy_dzl = b_duy_dxi*xizl + b_duy_dgamma*gammazl

                 b_duz_dxl = b_duz_dxi*xixl + b_duz_dgamma*gammaxl
                 b_duz_dzl = b_duz_dxi*xizl + b_duz_dgamma*gammazl
              endif

              ! compute stress tensor (include attenuation or anisotropy if need
ed)

              if(TURN_ATTENUATION_ON) then

                 ! attenuation is implemented following the memory variable form
ulation of
                 ! J. M. Carcione, Seismic modeling in viscoelastic media, Geoph
ysics,
                 ! vol. 58(1), p. 110-120 (1993). More details can be found in
                 ! J. M. Carcione, D. Kosloff and R. Kosloff, Wave propagation s
imulation in a linear
                 ! viscoelastic medium, Geophysical Journal International, vol.
95, p. 597-611 (1988).

                 ! compute unrelaxed elastic coefficients from formulas in Carci
one 1993 page 111
                 lambdal_unrelaxed = (lambdal_relaxed + mul_relaxed) * Mu_nu1(i,
j,ispec) - mul_relaxed * Mu_nu2(i,j,ispec)
                 mul_unrelaxed = mul_relaxed * Mu_nu2(i,j,ispec)
                 lambdalplus2mul_unrelaxed = lambdal_unrelaxed + TWO*mul_unrelax
ed

                 ! compute the stress using the unrelaxed Lame parameters (Carci
one 1993, page 111)
```

```
                  sigma_xx = lambdalplus2mul_unrelaxed*dux_dxl + lambdal_unrelaxe
d*duz_dzl
                  sigma_xz = mul_unrelaxed*(duz_dxl + dux_dzl)
                  sigma_zz = lambdalplus2mul_unrelaxed*duz_dzl + lambdal_unrelaxe
d*dux_dxl

                  ! add the memory variables using the relaxed parameters (Carcio
ne 1993, page 111)
                  ! beware: there is a bug in Carcione's equation (2c) for sigma_
zz, we fixed it in the code below
                  e1_sum = 0._CUSTOM_REAL
                  e11_sum = 0._CUSTOM_REAL
                  e13_sum = 0._CUSTOM_REAL

                  do i_sls = 1,N_SLS
                     e1_sum = e1_sum + e1(i,j,ispec,i_sls)
                     e11_sum = e11_sum + e11(i,j,ispec,i_sls)
                     e13_sum = e13_sum + e13(i,j,ispec,i_sls)
                  enddo

                  sigma_xx = sigma_xx + (lambdal_relaxed + mul_relaxed) * e1_sum
+ TWO * mul_relaxed * e11_sum
                  sigma_xz = sigma_xz + mul_relaxed * e13_sum
                  sigma_zz = sigma_zz + (lambdal_relaxed + mul_relaxed) * e1_sum
- TWO * mul_relaxed * e11_sum

               else

                  ! no attenuation
                  sigma_xx = lambdalplus2mul_relaxed*dux_dxl + lambdal_relaxed*du
z_dzl
                  sigma_xy = mul_relaxed*duy_dxl
                  sigma_xz = mul_relaxed*(duz_dxl + dux_dzl)
                  sigma_zy = mul_relaxed*duy_dzl
                  sigma_zz = lambdalplus2mul_relaxed*duz_dzl + lambdal_relaxed*du
x_dxl

                  if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward w
avefield
                     b_sigma_xx = lambdalplus2mul_relaxed*b_dux_dxl + lambdal_rel
axed*b_duz_dzl
                     b_sigma_xy = mul_relaxed*b_duy_dxl
                     b_sigma_xz = mul_relaxed*(b_duz_dxl + b_dux_dzl)
                     b_sigma_zy = mul_relaxed*b_duy_dzl
                     b_sigma_zz = lambdalplus2mul_relaxed*b_duz_dzl + lambdal_rel
axed*b_dux_dxl
                  endif

               endif

               ! full anisotropy
               if(anisotropic(ispec)) then
                  if(assign_external_model) then
                     c11 = c11ext(i,j,ispec)
                     c13 = c13ext(i,j,ispec)
                     c15 = c15ext(i,j,ispec)
                     c33 = c33ext(i,j,ispec)
                     c35 = c35ext(i,j,ispec)
                     c55 = c55ext(i,j,ispec)
                  else
                     c11 = anisotropy(1,kmato(ispec))
                     c13 = anisotropy(2,kmato(ispec))
```



```
                     c15 = anisotropy(3,kmato(ispec))
                     c33 = anisotropy(4,kmato(ispec))
                     c35 = anisotropy(5,kmato(ispec))
                     c55 = anisotropy(6,kmato(ispec))
                  end if

                  ! implement anisotropy in 2D
                  sigma_xx = c11*dux_dxl + c15*(duz_dxl + dux_dzl) + c13*duz_dzl
                  sigma_zz = c13*dux_dxl + c35*(duz_dxl + dux_dzl) + c33*duz_dzl
                  sigma_xz = c15*dux_dxl + c55*(duz_dxl + dux_dzl) + c35*duz_dzl

               endif

               ! Pre-kernels calculation
               if(SIMULATION_TYPE == 2) then
                  iglob = ibool(i,j,ispec)
                  if(p_sv)then !P-SV waves
                     dsxx =  dux_dxl
                     dsxz = HALF * (duz_dxl + dux_dzl)
                     dszz =  duz_dzl

                     b_dsxx =  b_dux_dxl
                     b_dsxz = HALF * (b_duz_dxl + b_dux_dzl)
                     b_dszz =  b_duz_dzl

                     kappa_k(iglob) = (dux_dxl + duz_dzl) *  (b_dux_dxl + b_duz_d
zl)
                     mu_k(iglob) = dsxx * b_dsxx + dszz * b_dszz + &
                          2._CUSTOM_REAL * dsxz * b_dsxz - 1._CUSTOM_REAL/3._CUST
OM_REAL * kappa_k(iglob)
                  else !SH (membrane) waves
                     mu_k(iglob) = duy_dxl * b_duy_dxl + duy_dzl * b_duy_dzl
                  endif
               endif

               jacobianl = jacobian(i,j,ispec)

               ! weak formulation term based on stress tensor (non-symmetric form
)
               ! also add GLL integration weights
               tempx1(i,j) = wzgll(j)*jacobianl*(sigma_xx*xixl+sigma_xz*xizl)
               tempy1(i,j) = wzgll(j)*jacobianl*(sigma_xy*xixl+sigma_zy*xizl)
               tempz1(i,j) = wzgll(j)*jacobianl*(sigma_xz*xixl+sigma_zz*xizl)

               tempx2(i,j) = wxgll(i)*jacobianl*(sigma_xx*gammaxl+sigma_xz*gammaz
l)
               tempy2(i,j) = wxgll(i)*jacobianl*(sigma_xy*gammaxl+sigma_zy*gammaz
l)
               tempz2(i,j) = wxgll(i)*jacobianl*(sigma_xz*gammaxl+sigma_zz*gammaz
l)

               if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward wave
field
                  b_tempx1(i,j) = wzgll(j)*jacobianl*(b_sigma_xx*xixl+b_sigma_xz*
xizl)
                  b_tempy1(i,j) = wzgll(j)*jacobianl*(b_sigma_xy*xixl+b_sigma_zy*
xizl)
                  b_tempz1(i,j) = wzgll(j)*jacobianl*(b_sigma_xz*xixl+b_sigma_zz*
xizl)

                  b_tempx2(i,j) = wxgll(i)*jacobianl*(b_sigma_xx*gammaxl+b_sigma_
xz*gammazl)
```

```
                  b_tempy2(i,j) = wxgll(i)*jacobianl*(b_sigma_xy*gammaxl+b_sigma_
zy*gammazl)
                  b_tempz2(i,j) = wxgll(i)*jacobianl*(b_sigma_xz*gammaxl+b_sigma_
zz*gammazl)
               endif

            enddo
         enddo

         !
         ! second double-loop over GLL to compute all the terms
         !
         do j = 1,NGLLZ
            do i = 1,NGLLX

               iglob = ibool(i,j,ispec)

               ! along x direction and z direction
               ! and assemble the contributions
               ! we can merge the two loops because NGLLX == NGLLZ
               do k = 1,NGLLX
                  accel_elastic(1,iglob) = accel_elastic(1,iglob) - (tempx1(k,j)*
hprimewgll_xx(k,i) + tempx2(i,k)*hprimewgll_zz(k,j))
                  accel_elastic(2,iglob) = accel_elastic(2,iglob) - (tempy1(k,j)*
hprimewgll_xx(k,i) + tempy2(i,k)*hprimewgll_zz(k,j))
                  accel_elastic(3,iglob) = accel_elastic(3,iglob) - (tempz1(k,j)*
hprimewgll_xx(k,i) + tempz2(i,k)*hprimewgll_zz(k,j))

                  if(SIMULATION_TYPE == 2) then ! Adjoint calculation, backward w
avefield
                     b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) - &
                          (b_tempx1(k,j)*hprimewgll_xx(k,i) + b_tempx2(i,k)*hprim
ewgll_zz(k,j))
                     b_accel_elastic(2,iglob) = b_accel_elastic(2,iglob) - &
                          (b_tempy1(k,j)*hprimewgll_xx(k,i) + b_tempy2(i,k)*hprim
ewgll_zz(k,j))
                     b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) - &
                          (b_tempz1(k,j)*hprimewgll_xx(k,i) + b_tempz2(i,k)*hprim
ewgll_zz(k,j))
                  endif
               enddo

            enddo ! second loop over the GLL points
         enddo

      endif ! end of test if elastic element

   enddo ! end of loop over all spectral elements

   !
   !--- absorbing boundaries
   !
   if(anyabs) then

      count_left=1
      count_right=1
      count_bottom=1

      do ispecabs = 1,nelemabs

         ispec = numabs(ispecabs)
```



```
         ! get elastic parameters of current spectral element
         lambdal_relaxed = poroelastcoef(1,1,kmato(ispec))
         mul_relaxed = poroelastcoef(2,1,kmato(ispec))
         rhol  = density(1,kmato(ispec))
         kappal  = lambdal_relaxed + TWO*mul_relaxed/3._CUSTOM_REAL
         cpl = sqrt((kappal + 4._CUSTOM_REAL*mul_relaxed/3._CUSTOM_REAL)/rhol)
         csl = sqrt(mul_relaxed/rhol)

         !--- left absorbing boundary
         if(codeabs(ILEFT,ispecabs)) then

            i = 1

            do j = 1,NGLLZ

               iglob = ibool(i,j,ispec)

               ! for analytical initial plane wave for Bielak's conditions
               ! left or right edge, horizontal normal vector
               if(add_Bielak_conditions .and. initialfield) then
                  if (.not.over_critical_angle) then
                     call compute_Bielak_conditions(coord,iglob,nglob,it,deltat,dxUx,
dxUz,dzUx,dzUz,veloc_horiz,veloc_vert, &
                          x0_source, z0_source, A_plane, B_plane, C_plane, anglef
orce(1), angleforce_refl, &
                          c_inc, c_refl, time_offset,f0)
                     traction_x_t0 = (lambdal_relaxed+2*mul_relaxed)*dxUx + lambd
al_relaxed*dzUz
                     traction_z_t0 = mul_relaxed*(dxUz + dzUx)
                  else
                     veloc_horiz=v0x_left(count_left)
                     veloc_vert=v0z_left(count_left)
                     traction_x_t0=t0x_left(count_left)
                     traction_z_t0=t0z_left(count_left)
                     count_left=count_left+1
                  end if
               else
                  veloc_horiz = 0
                  veloc_vert = 0
                  traction_x_t0 = 0
                  traction_z_t0 = 0
               endif

               ! external velocity model
               if(assign_external_model) then
                  cpl = vpext(i,j,ispec)
                  csl = vsext(i,j,ispec)
                  rhol = rhoext(i,j,ispec)
               endif

               rho_vp = rhol*cpl
               rho_vs = rhol*csl

               xgamma = - xiz(i,j,ispec) * jacobian(i,j,ispec)
               zgamma = + xix(i,j,ispec) * jacobian(i,j,ispec)
               jacobian1D = sqrt(xgamma**2 + zgamma**2)
               nx = - zgamma / jacobian1D
               nz = + xgamma / jacobian1D

               weight = jacobian1D * wzgll(j)

               ! Clayton-Engquist condition if elastic
```

```
              if(elastic(ispec)) then
                 vx = veloc_elastic(1,iglob) - veloc_horiz
                 vy = veloc_elastic(2,iglob)
                 vz = veloc_elastic(3,iglob) - veloc_vert

                 vn = nx*vx+nz*vz

                 tx = rho_vp*vn*nx+rho_vs*(vx-vn*nx)
                 ty = rho_vs*vy
                 tz = rho_vp*vn*nz+rho_vs*(vz-vn*nz)

                 accel_elastic(1,iglob) = accel_elastic(1,iglob) - (tx + tractio
n_x_t0)*weight
                 accel_elastic(2,iglob) = accel_elastic(2,iglob) - ty*weight
                 accel_elastic(3,iglob) = accel_elastic(3,iglob) - (tz + tractio
n_z_t0)*weight

                 if(SAVE_FORWARD .and. SIMULATION_TYPE ==1) then
                    if(p_sv)then !P-SV waves
                       b_absorb_elastic_left(1,j,ib_left(ispecabs),it) = tx*weig
ht
                       b_absorb_elastic_left(3,j,ib_left(ispecabs),it) = tz*weig
ht
                    else !SH (membrane) waves
                       b_absorb_elastic_left(2,j,ib_left(ispecabs),it) = ty*weig
ht
                    endif
                 elseif(SIMULATION_TYPE == 2) then
                    if(p_sv)then !P-SV waves
                       b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) - &
                            b_absorb_elastic_left(1,j,ib_left(ispecabs),NSTEP-it
+1)
                       b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) - &
                            b_absorb_elastic_left(3,j,ib_left(ispecabs),NSTEP-it
+1)
                    else !SH (membrane) waves
                       b_accel_elastic(2,iglob) = b_accel_elastic(2,iglob) - &
                            b_absorb_elastic_left(2,j,ib_left(ispecabs),NSTEP-it
+1)
                    endif
                 endif

              endif

           enddo

        endif  !  end of left absorbing boundary

        !--- right absorbing boundary
        if(codeabs(IRIGHT,ispecabs)) then

           i = NGLLX

           do j = 1,NGLLZ

              iglob = ibool(i,j,ispec)

              ! for analytical initial plane wave for Bielak's conditions
              ! left or right edge, horizontal normal vector
              if(add_Bielak_conditions .and. initialfield) then
                 if (.not.over_critical_angle) then
                    call compute_Bielak_conditions(coord,iglob,nglob,it,deltat,dxUx,
```



```
dxUz,dzUx,dzUz,veloc_horiz,veloc_vert, &
                         x0_source, z0_source, A_plane, B_plane, C_plane, anglef
orce(1), angleforce_refl, &
                         c_inc, c_refl, time_offset,f0)
                    traction_x_t0 = (lambdal_relaxed+2*mul_relaxed)*dxUx + lambd
al_relaxed*dzUz
                    traction_z_t0 = mul_relaxed*(dxUz + dzUx)
                 else
                    veloc_horiz=v0x_right(count_right)
                    veloc_vert=v0z_right(count_right)
                    traction_x_t0=t0x_right(count_right)
                    traction_z_t0=t0z_right(count_right)
                    count_right=count_right+1
                 end if
              else
                 veloc_horiz = 0
                 veloc_vert = 0
                 traction_x_t0 = 0
                 traction_z_t0 = 0
              endif

              ! external velocity model
              if(assign_external_model) then
                 cpl = vpext(i,j,ispec)
                 csl = vsext(i,j,ispec)
                 rhol = rhoext(i,j,ispec)
              endif

              rho_vp = rhol*cpl
              rho_vs = rhol*csl

              xgamma = - xiz(i,j,ispec) * jacobian(i,j,ispec)
              zgamma = + xix(i,j,ispec) * jacobian(i,j,ispec)
              jacobian1D = sqrt(xgamma**2 + zgamma**2)
              nx = + zgamma / jacobian1D
              nz = - xgamma / jacobian1D

              weight = jacobian1D * wzgll(j)

              ! Clayton-Engquist condition if elastic
              if(elastic(ispec)) then
                 vx = veloc_elastic(1,iglob) - veloc_horiz
                 vy = veloc_elastic(2,iglob)
                 vz = veloc_elastic(3,iglob) - veloc_vert

                 vn = nx*vx+nz*vz

                 tx = rho_vp*vn*nx+rho_vs*(vx-vn*nx)
                 ty = rho_vs*vy
                 tz = rho_vp*vn*nz+rho_vs*(vz-vn*nz)

                 accel_elastic(1,iglob) = accel_elastic(1,iglob) - (tx - tractio
n_x_t0)*weight
                 accel_elastic(2,iglob) = accel_elastic(2,iglob) - ty*weight
                 accel_elastic(3,iglob) = accel_elastic(3,iglob) - (tz - tractio
n_z_t0)*weight

                 if(SAVE_FORWARD .and. SIMULATION_TYPE ==1) then
                    if(p_sv)then !P-SV waves
                       b_absorb_elastic_right(1,j,ib_right(ispecabs),it) = tx*we
ight
                       b_absorb_elastic_right(3,j,ib_right(ispecabs),it) = tz*we
```

```
ight
                    else! SH (membrane) waves
                       b_absorb_elastic_right(2,j,ib_right(ispecabs),it) = ty*we
ight
                    endif
                 elseif(SIMULATION_TYPE == 2) then
                    if(p_sv)then !P-SV waves
                       b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) - &
                            b_absorb_elastic_right(1,j,ib_right(ispecabs),NSTEP-
it+1)
                       b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) - &
                            b_absorb_elastic_right(3,j,ib_right(ispecabs),NSTEP-
it+1)
                    else! SH (membrane) waves
                       b_accel_elastic(2,iglob) = b_accel_elastic(2,iglob) - &
                            b_absorb_elastic_right(2,j,ib_right(ispecabs),NSTEP-
it+1)
                    endif
                 endif

              endif

           enddo

        endif  !  end of right absorbing boundary

        !--- bottom absorbing boundary
        if(codeabs(IBOTTOM,ispecabs)) then

           j = 1

           ! exclude corners to make sure there is no contradiction on the norma
l
           ibegin = 1
           iend = NGLLX
           if(codeabs(ILEFT,ispecabs)) ibegin = 2
           if(codeabs(IRIGHT,ispecabs)) iend = NGLLX-1

           do i = ibegin,iend

              iglob = ibool(i,j,ispec)

              ! for analytical initial plane wave for Bielak's conditions
              ! top or bottom edge, vertical normal vector
              if(add_Bielak_conditions .and. initialfield) then
                 if (.not.over_critical_angle) then
                    call compute_Bielak_conditions(coord,iglob,nglob,it,deltat,dxUx,
dxUz,dzUx,dzUz,veloc_horiz,veloc_vert, &
                         x0_source, z0_source, A_plane, B_plane, C_plane, anglef
orce(1), angleforce_refl, &
                         c_inc, c_refl, time_offset,f0)
                    traction_x_t0 = mul_relaxed*(dxUz + dzUx)
                    traction_z_t0 = lambdal_relaxed*dxUx + (lambdal_relaxed+2*mu
l_relaxed)*dzUz
                 else
                    veloc_horiz=v0x_bot(count_bottom)
                    veloc_vert=v0z_bot(count_bottom)
                    traction_x_t0=t0x_bot(count_bottom)
                    traction_z_t0=t0z_bot(count_bottom)
                    count_bottom=count_bottom+1
                 end if
              else
```



```
                 veloc_horiz = 0
                 veloc_vert = 0
                 traction_x_t0 = 0
                 traction_z_t0 = 0
              endif

              ! external velocity model
              if(assign_external_model) then
                 cpl = vpext(i,j,ispec)
                 csl = vsext(i,j,ispec)
                 rhol = rhoext(i,j,ispec)
              endif

              rho_vp = rhol*cpl
              rho_vs = rhol*csl

              xxi = + gammaz(i,j,ispec) * jacobian(i,j,ispec)
              zxi = - gammax(i,j,ispec) * jacobian(i,j,ispec)
              jacobian1D = sqrt(xxi**2 + zxi**2)
              nx = + zxi / jacobian1D
              nz = - xxi / jacobian1D

              weight = jacobian1D * wxgll(i)

              ! Clayton-Engquist condition if elastic
              if(elastic(ispec)) then
                 vx = veloc_elastic(1,iglob) - veloc_horiz
                 vy = veloc_elastic(2,iglob)
                 vz = veloc_elastic(3,iglob) - veloc_vert

                 vn = nx*vx+nz*vz

                 tx = rho_vp*vn*nx+rho_vs*(vx-vn*nx)
                 ty = rho_vs*vy
                 tz = rho_vp*vn*nz+rho_vs*(vz-vn*nz)

                 accel_elastic(1,iglob) = accel_elastic(1,iglob) - (tx + tractio
n_x_t0)*weight
                 accel_elastic(2,iglob) = accel_elastic(2,iglob) - ty*weight
                 accel_elastic(3,iglob) = accel_elastic(3,iglob) - (tz + tractio
n_z_t0)*weight

                 if(SAVE_FORWARD .and. SIMULATION_TYPE ==1) then
                    if(p_sv)then !P-SV waves
                       b_absorb_elastic_bottom(1,i,ib_bottom(ispecabs),it) = tx*
weight
                       b_absorb_elastic_bottom(3,i,ib_bottom(ispecabs),it) = tz*
weight
                    else!SH (membrane) waves
                       b_absorb_elastic_bottom(2,i,ib_bottom(ispecabs),it) = ty*
weight
                    endif
                 elseif(SIMULATION_TYPE == 2) then
                    if(p_sv)then !P-SV waves
                       b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) - &
                            b_absorb_elastic_bottom(1,i,ib_bottom(ispecabs),NSTE
P-it+1)
                       b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) - &
                            b_absorb_elastic_bottom(3,i,ib_bottom(ispecabs),NSTE
P-it+1)
                    else!SH (membrane) waves
                       b_accel_elastic(2,iglob) = b_accel_elastic(2,iglob) - &
```

```
                              b_absorb_elastic_bottom(2,i,ib_bottom(ispecabs),NSTE
P-it+1)
                     endif
                  endif

               endif

            enddo

         endif  !  end of bottom absorbing boundary

         !--- top absorbing boundary
         if(codeabs(ITOP,ispecabs)) then

            j = NGLLZ

            ! exclude corners to make sure there is no contradiction on the norma
l
            ibegin = 1
            iend = NGLLX
            if(codeabs(ILEFT,ispecabs)) ibegin = 2
            if(codeabs(IRIGHT,ispecabs)) iend = NGLLX-1

            do i = ibegin,iend

               iglob = ibool(i,j,ispec)

               ! for analytical initial plane wave for Bielak's conditions
               ! top or bottom edge, vertical normal vector
               if(add_Bielak_conditions .and. initialfield) then
                  call compute_Bielak_conditions(coord,iglob,nglob,it,deltat,dxUx,dxU
z,dzUx,dzUz,veloc_horiz,veloc_vert, &
                       x0_source, z0_source, A_plane, B_plane, C_plane, angleforc
e(1), angleforce_refl, &
                       c_inc, c_refl, time_offset,f0)
                  traction_x_t0 = mul_relaxed*(dxUz + dzUx)
                  traction_z_t0 = lambdal_relaxed*dxUx + (lambdal_relaxed+2*mul_r
elaxed)*dzUz
               else
                  veloc_horiz = 0
                  veloc_vert = 0
                  traction_x_t0 = 0
                  traction_z_t0 = 0
               endif

               ! external velocity model
               if(assign_external_model) then
                  cpl = vpext(i,j,ispec)
                  csl = vsext(i,j,ispec)
                  rhol = rhoext(i,j,ispec)
               endif

               rho_vp = rhol*cpl
               rho_vs = rhol*csl

               xxi = + gammaz(i,j,ispec) * jacobian(i,j,ispec)
               zxi = - gammax(i,j,ispec) * jacobian(i,j,ispec)
               jacobian1D = sqrt(xxi**2 + zxi**2)
               nx = - zxi / jacobian1D
               nz = + xxi / jacobian1D

               weight = jacobian1D * wxgll(i)
```



```
               ! Clayton-Engquist condition if elastic
               if(elastic(ispec)) then
                  vx = veloc_elastic(1,iglob) - veloc_horiz
                  vy = veloc_elastic(2,iglob)
                  vz = veloc_elastic(3,iglob) - veloc_vert

                  vn = nx*vx+nz*vz

                  tx = rho_vp*vn*nx+rho_vs*(vx-vn*nx)
                  ty = rho_vs*vy
                  tz = rho_vp*vn*nz+rho_vs*(vz-vn*nz)

                  accel_elastic(1,iglob) = accel_elastic(1,iglob) - (tx - tractio
n_x_t0)*weight
                  accel_elastic(2,iglob) = accel_elastic(2,iglob) - ty*weight
                  accel_elastic(3,iglob) = accel_elastic(3,iglob) - (tz - tractio
n_z_t0)*weight

                  if(SAVE_FORWARD .and. SIMULATION_TYPE ==1) then
                     if(p_sv)then !P-SV waves
                        b_absorb_elastic_top(1,i,ib_top(ispecabs),it) = tx*weight
                        b_absorb_elastic_top(3,i,ib_top(ispecabs),it) = tz*weight
                     else!SH (membrane) waves
                        b_absorb_elastic_top(2,i,ib_top(ispecabs),it) = ty*weight
                     endif
                  elseif(SIMULATION_TYPE == 2) then
                     if(p_sv)then !P-SV waves
                        b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) - b_a
bsorb_elastic_top(1,i,ib_top(ispecabs),NSTEP-it+1)
                        b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) - b_a
bsorb_elastic_top(3,i,ib_top(ispecabs),NSTEP-it+1)
                     else!SH (membrane) waves
                        b_accel_elastic(2,iglob) = b_accel_elastic(2,iglob) - b_a
bsorb_elastic_top(2,i,ib_top(ispecabs),NSTEP-it+1)
                     endif
                  endif

               endif

            enddo

         endif  !  end of top absorbing boundary

      enddo

   endif  ! end of absorbing boundaries

   ! --- add the source if it is a moment tensor
   if(.not. initialfield) then

      do i_source=1,NSOURCES
         ! if this processor carries the source and the source element is elastic
         if (is_proc_source(i_source) == 1 .and. elastic(ispec_selected_source(i_
source))) then

            ! moment tensor
            if(source_type(i_source) == 2) then

               if(.not.p_sv)  call exit_MPI('cannot have moment tensor source in SH (membrane)
waves calculation')
```

```
              if(SIMULATION_TYPE == 1) then  ! forward wavefield
                 ! add source array
                 do j=1,NGLLZ
                    do i=1,NGLLX
                       iglob = ibool(i,j,ispec_selected_source(i_source))
                       accel_elastic(1,iglob) = accel_elastic(1,iglob) + &
                            sourcearray(i_source,1,i,j)*source_time_function(i_s
ource,it)
                       accel_elastic(3,iglob) = accel_elastic(3,iglob) + &
                            sourcearray(i_source,2,i,j)*source_time_function(i_s
ource,it)
                    enddo
                 enddo
              else                    ! backward wavefield
                 do j=1,NGLLZ
                    do i=1,NGLLX
                       iglob = ibool(i,j,ispec_selected_source(i_source))
                       b_accel_elastic(1,iglob) = b_accel_elastic(1,iglob) + &
                            sourcearray(i_source,1,i,j)*source_time_function(i_s
ource,NSTEP-it+1)
                       b_accel_elastic(3,iglob) = b_accel_elastic(3,iglob) + &
                            sourcearray(i_source,2,i,j)*source_time_function(i_s
ource,NSTEP-it+1)
                    enddo
                 enddo
              endif  !endif SIMULATION_TYPE == 1

           endif !if(source_type(i_source) == 2)

        endif ! if this processor carries the source and the source element is e
lastic
     enddo ! do i_source=1,NSOURCES

     if(SIMULATION_TYPE == 2) then   ! adjoint wavefield

        irec_local = 0
        do irec = 1,nrec
           !   add the source (only if this proc carries the source)
           if(myrank == which_proc_receiver(irec)) then

              irec_local = irec_local + 1
              if(elastic(ispec_selected_rec(irec))) then
                 ! add source array
                 do j=1,NGLLZ
                    do i=1,NGLLX
                       iglob = ibool(i,j,ispec_selected_rec(irec))
                       if(p_sv)then !P-SH waves
                          accel_elastic(1,iglob) = accel_elastic(1,iglob) + adj_
sourcearrays(irec_local,NSTEP-it+1,1,i,j)
                          accel_elastic(3,iglob) = accel_elastic(3,iglob) + adj_
sourcearrays(irec_local,NSTEP-it+1,3,i,j)
                       else !SH (membrane) waves
                          accel_elastic(2,iglob) = accel_elastic(2,iglob) + adj_
sourcearrays(irec_local,NSTEP-it+1,2,i,j)
                       endif
                    enddo
                 enddo
              endif ! if element is elastic

           endif ! if this processor carries the adjoint source and the source e
lement is elastic
        enddo ! irec = 1,nrec
```



```
     endif ! if SIMULATION_TYPE == 2 adjoint wavefield

  endif ! if not using an initial field

  ! implement attenuation
  if(TURN_ATTENUATION_ON) then

     ! compute Grad(displ_elastic) at time step n+1 for attenuation
     call compute_gradient_attenuation(displ_elastic,dux_dxl_np1,duz_dxl_np1, &
          dux_dzl_np1,duz_dzl_np1,xix,xiz,gammax,gammaz,ibool,elastic,hprime_xx,
hprime_zz,nspec,nglob)

     ! update memory variables with fourth-order Runge-Kutta time scheme for att
enuation
     ! loop over spectral elements
     do ispec = 1,nspec

        do j=1,NGLLZ
           do i=1,NGLLX

              theta_n   = dux_dxl_n(i,j,ispec) + duz_dzl_n(i,j,ispec)
              theta_np1 = dux_dxl_np1(i,j,ispec) + duz_dzl_np1(i,j,ispec)

              ! loop on all the standard linear solids
              do i_sls = 1,N_SLS

                 ! evolution e1
                 Un = e1(i,j,ispec,i_sls)
                 tauinv = - inv_tau_sigma_nu1(i,j,ispec,i_sls)
                 tauinvsquare = tauinv * tauinv
                 tauinvcube = tauinvsquare * tauinv
                 tauinvUn = tauinv * Un
                 Sn   = theta_n * phi_nu1(i,j,ispec,i_sls)
                 Snp1 = theta_np1 * phi_nu1(i,j,ispec,i_sls)
                 Unp1 = Un + (deltatfourth*tauinvcube*(Sn + tauinvUn) + &
                      twelvedeltat*(Sn + Snp1 + 2*tauinvUn) + &
                      fourdeltatsquare*tauinv*(2*Sn + Snp1 + 3*tauinvUn) + &
                      deltatcube*tauinvsquare*(3*Sn + Snp1 + 4*tauinvUn))* ONE_O
VER_24
                 e1(i,j,ispec,i_sls) = Unp1

                 ! evolution e11
                 Un = e11(i,j,ispec,i_sls)
                 tauinv = - inv_tau_sigma_nu2(i,j,ispec,i_sls)
                 tauinvsquare = tauinv * tauinv
                 tauinvcube = tauinvsquare * tauinv
                 tauinvUn = tauinv * Un
                 Sn   = (dux_dxl_n(i,j,ispec) - theta_n/TWO) * phi_nu2(i,j,ispec
,i_sls)
                 Snp1 = (dux_dxl_np1(i,j,ispec) - theta_np1/TWO) * phi_nu2(i,j,i
spec,i_sls)
                 Unp1 = Un + (deltatfourth*tauinvcube*(Sn + tauinvUn) + &
                      twelvedeltat*(Sn + Snp1 + 2*tauinvUn) + &
                      fourdeltatsquare*tauinv*(2*Sn + Snp1 + 3*tauinvUn) + &
                      deltatcube*tauinvsquare*(3*Sn + Snp1 + 4*tauinvUn))* ONE_O
VER_24
                 e11(i,j,ispec,i_sls) = Unp1

                 ! evolution e13
                 Un = e13(i,j,ispec,i_sls)
                 tauinv = - inv_tau_sigma_nu2(i,j,ispec,i_sls)
```

```
                  tauinvsquare = tauinv * tauinv
                  tauinvcube = tauinvsquare * tauinv
                  tauinvUn = tauinv * Un
                  Sn   = (dux_dzl_n(i,j,ispec) + duz_dxl_n(i,j,ispec)) * phi_nu2(
i,j,ispec,i_sls)
                  Snp1 = (dux_dzl_np1(i,j,ispec) + duz_dxl_np1(i,j,ispec)) * phi_
nu2(i,j,ispec,i_sls)
                  Unp1 = Un + (deltatfourth*tauinvcube*(Sn + tauinvUn) + &
                       twelvedeltat*(Sn + Snp1 + 2*tauinvUn) + &
                       fourdeltatsquare*tauinv*(2*Sn + Snp1 + 3*tauinvUn) + &
                       deltatcube*tauinvsquare*(3*Sn + Snp1 + 4*tauinvUn))* ONE_O
VER_24
                  e13(i,j,ispec,i_sls) = Unp1

               enddo

            enddo
         enddo
      enddo

  endif ! end of test on attenuation

end subroutine compute_forces_viscoelastic
```

9

Algorithms for Hyperbolic and Parabolic-Hyperbolic Problems

9.1 ONE-STEP ALGORITHMS FOR THE SEMIDISCRETE EQUATION OF MOTION

9.1.1 The Newmark Method

Recall from Chapter 7 that the semidiscrete equation of motion is written as

$$\boldsymbol{M}\ddot{\boldsymbol{d}} + \boldsymbol{C}\dot{\boldsymbol{d}} + \boldsymbol{K}\boldsymbol{d} = \boldsymbol{F} \tag{9.1.1}$$

where $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{C}$ is the viscous damping matrix, $\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{F}$ is the vector of applied forces, and $\boldsymbol{d}$, $\dot{\boldsymbol{d}}$, and $\ddot{\boldsymbol{d}}$ are the displacement, velocity and acceleration vectors, respectively. We take $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{K}$ to be symmetric; $\boldsymbol{M}$ is positive-definite, and $\boldsymbol{C}$ and $\boldsymbol{K}$ are positive-semidefinite.

The initial-value problem for (9.1.1) consists of finding a displacement, $\boldsymbol{d} = \boldsymbol{d}(t)$, satisfying (9.1.1) and the given initial data:

$$\boldsymbol{d}(0) = \boldsymbol{d}_0 \tag{9.1.2}$$

$$\dot{\boldsymbol{d}}(0) = \boldsymbol{v}_0 \tag{9.1.3}$$

Perhaps the most widely used family of direct methods for solving (9.1.1) to (9.1.3) is the *Newmark family* [1], which consists of the following equations:

$$\boldsymbol{M}\boldsymbol{a}_{n+1} + \boldsymbol{C}\boldsymbol{v}_{n+1} + \boldsymbol{K}\boldsymbol{d}_{n+1} = \boldsymbol{F}_{n+1} \tag{9.1.4}$$

$$\boldsymbol{d}_{n+1} = \boldsymbol{d}_n + \Delta t \boldsymbol{v}_n + \frac{\Delta t^2}{2}[(1 - 2\beta)\boldsymbol{a}_n + 2\beta \boldsymbol{a}_{n+1}] \tag{9.1.5}$$

$$v_{n+1} = v_n + \Delta t[(1 - \gamma)a_n + \gamma a_{n+1}] \tag{9.1.6}$$

where d_n, v_n, and a_n are the approximations of $d(t_n)$, $\dot{d}(t_n)$, and $\ddot{d}(t_n)$, respectively. Equation (9.1.4) is simply the equation of motion in terms of the approximate solution, and (9.1.5) and (9.1.6) are finite difference formulas describing the evolution of the approximate solution. The parameters β and γ determine the stability and accuracy characteristics of the algorithm under consideration. Equations (9.1.4) to (9.1.6) may be thought of as three equations for determining the three unknowns d_{n+1}, v_{n+1}, and a_{n+1}, it being assumed that d_n, v_n, and a_n are known from the previous step's calculations. The Newmark family contains as special cases many well-known and widely used methods.

Implementation: a-form

In our case beta = 0 (purely explicit scheme), gamma = 1/2, and matrix C is zero in the elastic case.

There are several possible implementations. We will sketch one, but we leave further details until Sec. 9.4, which deals with operator and mesh partitions. The results in Sec. 9.4 include the Newmark method as a special case. Define ***predictors***:

$$\tilde{d}_{n+1} = d_n + \Delta t v_n + \frac{\Delta t^2}{2}(1 - \cancel{\beta})a_n \tag{9.1.7}$$

$$\tilde{v}_{n+1} = v_n + (1 - \gamma)\Delta t a_n \tag{9.1.8}$$

Equations (9.1.5) and (9.1.6) may then be written as

$$d_{n+1} = \tilde{d}_{n+1} + \cancel{\beta} a_{n+1} \tag{9.1.9}$$

$$v_{n+1} = \tilde{v}_{n+1} + \gamma \Delta t a_{n+1} \tag{9.1.10}$$

The recursion relation determines a_{n+1}:

$$(M + \cancel{\gamma C} + \cancel{\beta K})a_{n+1} = F_{n+1} - \cancel{C v_{n+1}} - K\tilde{d}_{n+1} \tag{9.1.12}$$

Equations (9.1.9) and (9.1.10) may then be used to calculate d_{n+1} and v_{n+1}, respectively.

This form of implementation is convenient for generalization to algorithms that employ "mesh partitions" (see Sec. 9.4) but is not the most efficient implementation.

$$v_{n+1} = v_n + \Delta t[(1 - \gamma)a_n + \gamma a_{n+1}] \tag{9.1.6}$$

where d_n, v_n, and a_n are the approximations of $d(t_n)$, $\dot{d}(t_n)$, and $\ddot{d}(t_n)$, respectively. Equation (9.1.4) is simply the equation of motion in terms of the approximate solution, and (9.1.5) and (9.1.6) are finite difference formulas describing the evolution of the approximate solution. The parameters β and γ determine the stability and accuracy characteristics of the algorithm under consideration. Equations (9.1.4) to (9.1.6) may be thought of as three equations for determining the three unknowns d_{n+1}, v_{n+1}, and a_{n+1}, it being assumed that d_n, v_n, and a_n are known from the previous step's calculations. The Newmark family contains as special cases many well-known and widely used methods.

This is the original version of the previous page.

Implementation: a-form

There are several possible implementations. We will sketch one, but we leave further details until Sec. 9.4, which deals with operator and mesh partitions. The results in Sec. 9.4 include the Newmark method as a special case. Define *predictors*:

$$\tilde{d}_{n+1} = d_n + \Delta t v_n + \frac{\Delta t^2}{2}(1 - 2\beta)a_n \tag{9.1.7}$$

$$\tilde{v}_{n+1} = v_n + (1 - \gamma)\Delta t a_n \tag{9.1.8}$$

Equations (9.1.5) and (9.1.6) may then be written as

$$d_{n+1} = \tilde{d}_{n+1} + \beta\Delta t^2 a_{n+1} \tag{9.1.9}$$

$$v_{n+1} = \tilde{v}_{n+1} + \gamma\Delta t a_{n+1} \tag{9.1.10}$$

To start the process, a_0 may be calculated from

$$Ma_0 = F - Cv_0 - Kd_0 \tag{9.1.11}$$

or specified directly. The recursion relation determines a_{n+1}:

$$(M + \gamma\Delta t C + \beta\Delta t^2 K)a_{n+1} = F_{n+1} - C\tilde{v}_{n+1} - K\tilde{d}_{n+1} \tag{9.1.12}$$

Equations (9.1.9) and (9.1.10) may then be used to calculate d_{n+1} and v_{n+1}, respectively.

This form of implementation is convenient for generalization to algorithms that employ "mesh partitions" (see Sec. 9.4) but is not the most efficient implementation.