Sheet metal forming - Asaro and Bassani-Wu models - Fortran

Hui ZHOU
CityU - Hong Kong
2022-09

We firstly recall the notations from continuum mechanics. For convenience, the Einstein's summation rule is used. σ_{ij} denotes the Cauchy stress and ε_{ij} denotes the infinitesimal strain. The elastic strain and plastic strain are denoted by $\varepsilon_{ij}^{\mathrm{e}}$ and $\varepsilon_{ij}^{\mathrm{p}}$ respectively.

The crystal plasticity model is embedded in the following equations,

1) $\dot{\varepsilon}_{ij}^{\mathrm{e}} = S_{ijkl}\dot{\sigma}_{kl}$.

2) $\dot{\varepsilon}_{ij}^{\mathrm{p}} = \sum_{\beta=1}^{N} \dot{\gamma}^{\beta}\mu_{ij}^{\beta}$.

3) $\dot{\gamma} = \sum_{\beta=1}^{N} |\dot{\gamma}^{\beta}|$.

4) $\dot{\gamma}^{\alpha} = H_{\alpha\beta}^{-1}(\gamma)\dot{\tau}^{\beta}$.

In which,

S_{ijkl} is the compliance matrix (6×6).

μ_{ij}^{β} is the symmetric part of Schmid factor, i.e., $\tau^{\beta} = \sigma_{ij}\mu_{ij}^{\beta}$.

$\dot{\gamma}^{\beta}$ is the shear rate of the βth slip system.

γ is the total shear strain.

$H_{\alpha\beta}(\gamma)$ is the hardening rate matrix where the slip interaction of α and β is involved.

We note that the elastic strain and plastic strain are split in this formulation. The plastic deformation is obtained hereafter with the aid of hardening rate.

The fortran programming language is used to implement the constitutive equations. In general, the main file is written in the following,

```fortran
program main
    ! use Bassani-Wu model
    use bassani1
    implicit none

    real, dimension(:), allocatable :: strains, stress

    ! real parameters
    real, dimension(:), allocatable :: rparams

    ! integer parameters
    integer, dimension(:), allocatable :: iparams

    real :: t, dt
    integer :: i, j

    allocate(strains(7))
    allocate(rparams(6), iparams(12))
    allocate(stress(6))

    ! initialize the model
    call init(1.0, 0.1, 0.5, 1.0, 1.0, 2.0, 0.01, 1.4)
```

```fortran
      ! initial conditions
      strains = 0.0
      rparams = (/0.0, 0.0, 0.0, 1.0, 0.0, 0.0/)
      iparams = (/1, 1, -1,
     1           -1, -1,-1,
     2           -1, -1,-1,
     3            1, -1,-1/)
      stress = 0.0

      open(1, FILE="test_bassani1.dat")
      dt = 1e-2
      t = 0.0

      write(1, "(F10.4, F10.4)") strains(4), stress(4)

      ! time integration
      do i = 1, 1000
          write(6, "(a5, i3)") "step ", i
          call run_forward(7, strains, 0.0, dt,
     1                 rparams, iparams)
          t = t+dt
          stress = stress + rparams*dt
          write(6, "(9F10.4)") strains
          write(1, "(F10.4, F10.4)") strains(4), stress(4)
      end do

      close(1)

end program
```

This main file opens a data file called test_bassani1.dat and outputs the stress component σ_{23} and strain component ε^{p}_{23} into the data file. It is noted that the fortran 77 compiler is strict for line continuation and number of characters per line.

We would like to compile this file using the following command,

```
gfortran Bassani1.f test_bassani1.f -lblas -llapack -o test_bassani1
```

In the command, the blas and lapack packages are required. Bassani1.f is the module containing the Bassani-Wu model and test_bassani1.f is the main file.

If running smoothly, the executable file generates the data file test_bassani1.dat. Otherwise, the user need to check and install the blas and lapack packages.

The data in the test_bassani1.dat can be visualized in the following python codes.

```python
# -*- coding: utf-8 -*-
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
mpl.use('agg')
```

```
filename = "test_bassani1.dat"
data = np.loadtxt(filename)

x = [abs(t[0]) for t in data]
y = [t[1] for t in data]

fig = plt.figure()
plt.plot(x, y)
plt.grid(True)
plt.xlabel(r"$\gamma$")
plt.ylabel(r"$\tau$")
plt.title("plot y(x)")
plt.tight_layout()
plt.savefig(filename+".ps", format="ps")
```

This file saved as plot_test_bassani1.py plots the data using the shell command

```
python plot_test_bassani1.py
```

Finally, we can obtain the following figure (Fig. 1).

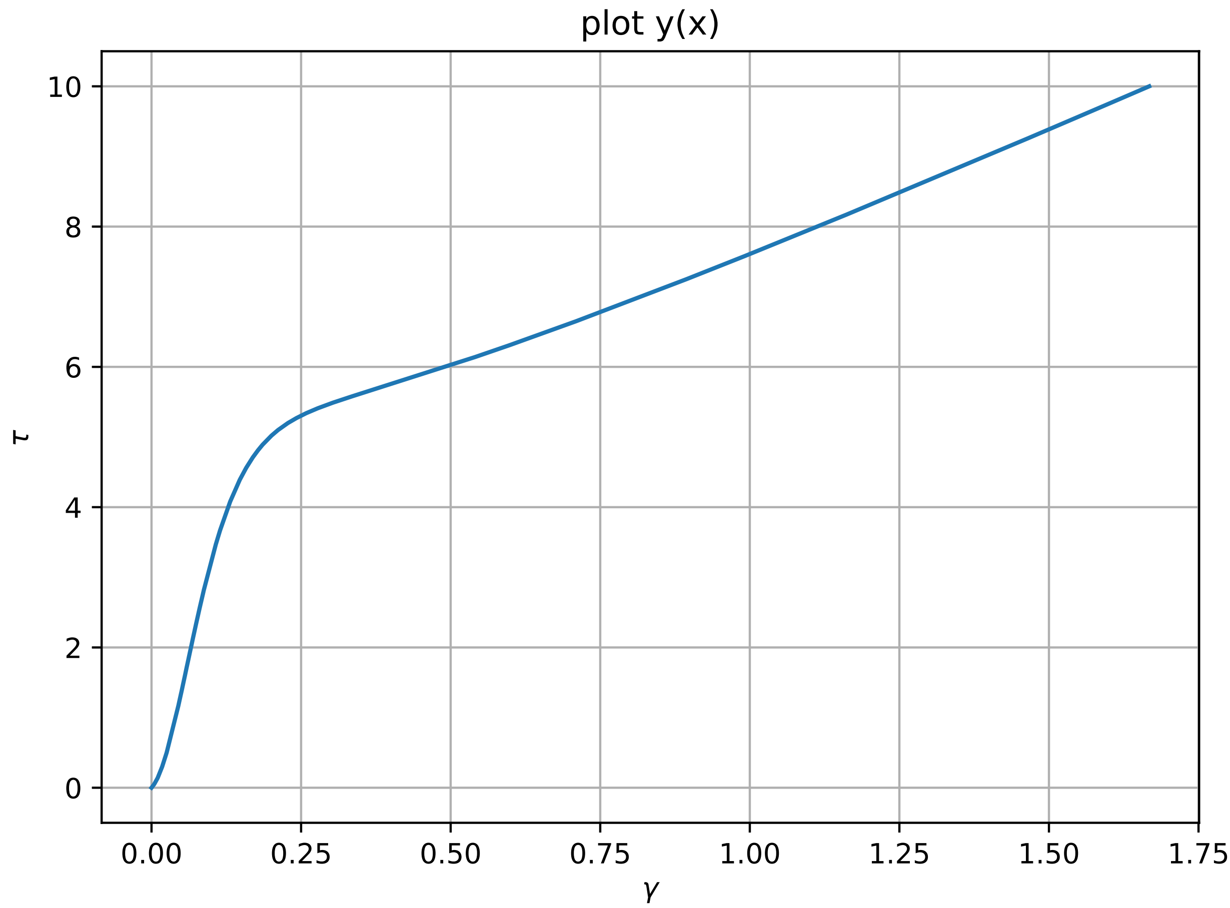

Figure 1: plot test_bassani1.dat

We would like to elaborate the procedures in the main file step by step.

The index of stress and strain adopts the Voigt's notation, therefore, the variable stress is an array contains (/σ_{11}, σ_{22}, σ_{33}, σ_{23}, σ_{13}, σ_{12}/). The variable strains is an array contains (/ε_{11}, ε_{22}, ε_{33}, ε_{23}, ε_{13}, ε_{12}, γ/). The total strain γ is the additional term.

The variable rparams is an array contains the stress rates, (/$\dot{\sigma}_{11}$, $\dot{\sigma}_{22}$, $\dot{\sigma}_{33}$, $\dot{\sigma}_{23}$, $\dot{\sigma}_{13}$, $\dot{\sigma}_{12}$/). In the main file, the fourth term $\dot{\sigma}_{23}$ is assigned value 1.0.

The variable iparams is an integer array marks the active slip systems (1: active, -1: non-active), here, iparams = (/1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1/) activates three slip system of the FCC 12 slip systems.

Table 1: FCC 12 slip systems

primary system (11-1)	conjugate system (1-11)	cross-glide system (1-1-1)	critical system (111)
1: [101]	4: [-1-10]	7: [-10-1]	10: [-101]
2: [0-1-1]	5: [011]	8: [110]	11: [01-1]
3: [-110]	6: [10-1]	9: [0-11]	12: [1-10]

From Table 1, the active slip systems are 1: (11-1)[101], 2: (11-1)[0-1-1] and 10: (111)[-101].

The subroutine run_forward is an RK-4 integration step,

```
call run_forward(7, strains, 0.0, dt, rparams, iparams)
```

The first item 7 is the size of variable strains. The third item 0.0 labels the initial condition: time t=0. The variable strains at t=0 is set to zero. The time step dt is specified to be 0.01. In the time loop, the variable stress and time t are updated and the variable strains are updated internally in the subroutine. The Asaro model has the stress limit which in turns out to be the upper bound of time steps. The subroutine run_forward outputs the state of strains variable for each time step. In the console, the user need to notice if the matrix singularity meets. That means the time steps need to be tuned.

The subroutine init is the parameter setting,

```
call init(1.0, 0.1, 0.5, 1.0, 1.0, 2.0, 0.01, 1.4)
```

corresponding to the Table 2.

Table 2: Parameters of the Bassabi-Wu model

elastic constant	c_{11}	1.0
	c_{12}	0.1
	c_{44}	0.5
initial hardening rate	h_0	1.0
initial yield shear	τ_0	1.0
saturation shear	τ_{s}	2.0
saturation hardening rate	h_{s}	0.01
latent hardening coefficient	q_{l}	1.4

The elastic strains are not counted here. In case, the module provides the subroutine for the elastic moduli C_{ijkl} (6×6). The Asaro model excludes the linear hardening term h_{s}. Its init subroutine,

```
call init(1.0, 0.1, 0.5, 1.0, 1.0, 2.0, 1.4)
```

corresponding to the Table 3.

To use the Asaro model, we only need to replace the Bassani model in the following changes,

Table 3: Parameters of the Asaro model

elastic constant	c_{11}	1.0
	c_{12}	0.1
	c_{44}	0.5
initial hardening rate	h_0	1.0
initial yield shear	τ_0	1.0
saturation shear	τ_s	2.0
latent hardening coefficient	q_l	1.4

```
program main
    ! use Asaro model
    use asaro2
    implicit none

    real, dimension(:), allocatable :: strains, stress

    ! real parameters
    real, dimension(:), allocatable :: rparams

    ! integer parameters
    integer, dimension(:), allocatable :: iparams

    real :: t, dt
    integer :: i, j

    allocate(strains(7))
    allocate(rparams(6), iparams(12))
    allocate(stress(6))

    ! model parameter changed
    call init(1.0, 0.1, 0.5, 1.0, 1.0, 2.0, 1.4)

    ! initial conditions
    strains = 0.0
    rparams = (/0.0, 0.0, 0.0, 1.0, 0.0, 0.0/)
    iparams = (/1, 1, -1,
     1          -1, -1,-1,
     2          -1, -1,-1,
     3           1, -1,-1/)
    stress = 0.0

    open(1, FILE="test_asaro2.dat")
    dt = 1e-2
    t = 0.0

    write(1, "(F10.4, F10.4)") strains(4), stress(4)

    ! time steps changed
    do i = 1, 75
```

```
        write(6, "(a5, i3)") "step ", i
        call run_forward(7, strains, 0.0, dt,
     1             rparams, iparams)
        t = t+dt
        stress = stress + rparams*dt
        write(6, "(9F10.4)") strains
        write(1, "(F10.4, F10.4)") strains(4), stress(4)
    end do

    close(1)

end program
```

The plotting of the data file test_asaro2.dat is shown in Fig. 2.

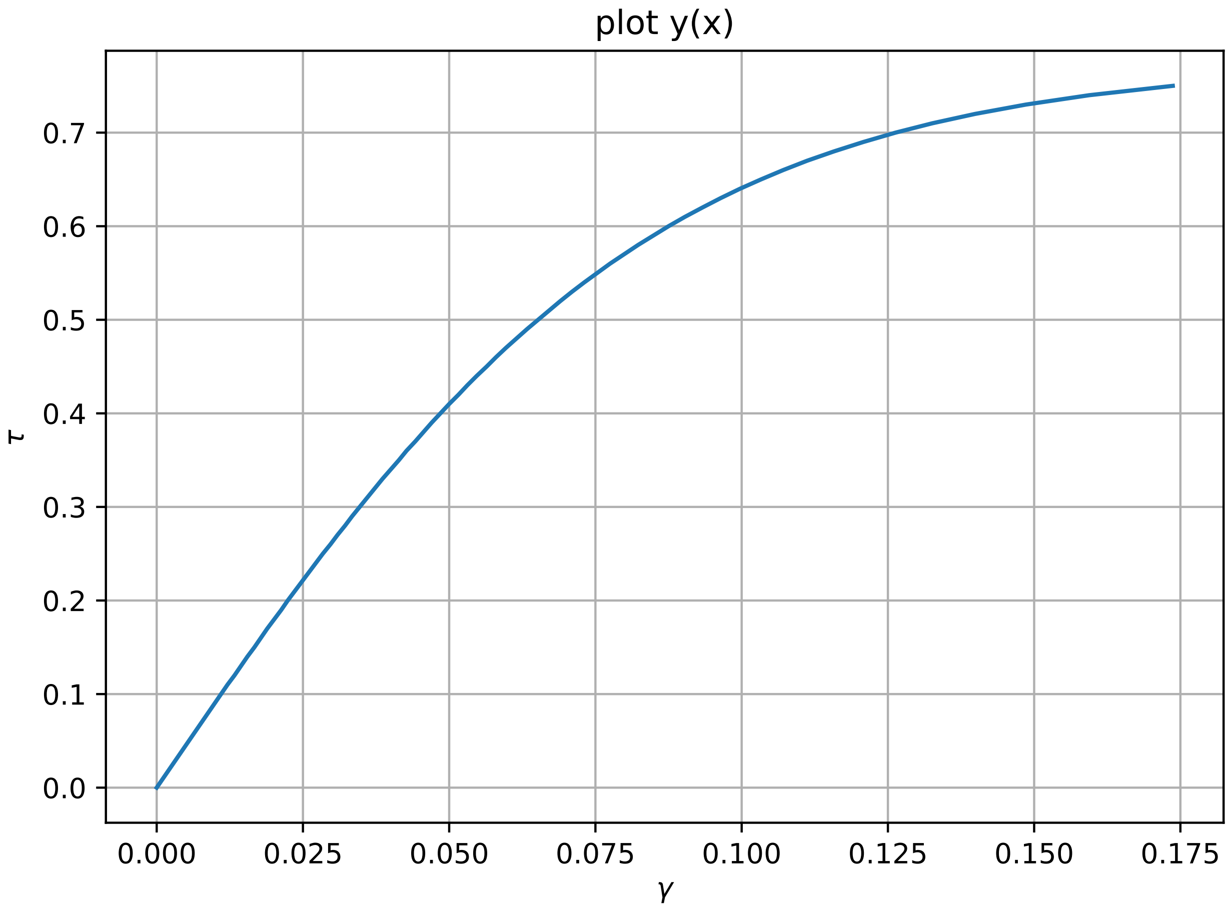

Figure 2: plot test_asaro2.dat

In conclusion, we build the ode formulation of two crystal plasticity models: Asaro and Bassani-Wu. The fortran modules are flexible, compact and minimal dependence. We provide the modules Bassani1.f and Asaro2.f in the appendix.

Bassani1.f

```
      module bassani1
      implicit none
```

```
    integer, dimension(:,:), allocatable :: fcc_slipnor
    integer, dimension(:,:), allocatable :: fcc_primary
    integer, dimension(:,:), allocatable :: fcc_conjugate
    integer, dimension(:,:), allocatable :: fcc_crossglide
    integer, dimension(:,:), allocatable :: fcc_critical

    real, dimension(:,:), allocatable :: fcc_C

    real :: fcc_c11, fcc_c12, fcc_c44
    real :: bassani_h0, bassani_tau0, bassani_taus, bassani_hs
    real :: bassani_q

    integer :: fcc_a1, fcc_a2, fcc_a3, fcc_a4, fcc_a5

    public :: fcc_slipnor
    public :: fcc_primary, fcc_conjugate
    public :: fcc_crossglide, fcc_critical

    private :: fcc_c11, fcc_c12, fcc_c44
    private :: bassani_h0, bassani_tau0, bassani_taus, bassani_hs
    private :: bassani_q
    private :: fcc_a1, fcc_a2, fcc_a3, fcc_a4, fcc_a5
contains

    subroutine init(c11, c12, c44, h0, tau0, taus, hs, q)
    real, intent(in) :: c11, c12, c44, h0, tau0, taus, hs
    real, intent(in) :: q
        allocate(fcc_slipnor(3, 4))
        allocate(fcc_primary(3, 3))
        allocate(fcc_conjugate(3, 3))
        allocate(fcc_crossglide(3, 3))
        allocate(fcc_critical(3, 3))

        !allocate(fcc_C(6,6))

        fcc_slipnor = reshape((/
1                  1, 1, -1,
2                  1, -1, 1,
3                  1, -1, -1,
4                  1,  1,  1
5                 /), shape(fcc_slipnor))

        fcc_primary = reshape((/
1                   1, 0, 1,
2                  0, -1, -1,
3                  -1, 1, 0
4                  /), shape(fcc_primary))

        fcc_conjugate = reshape((/
1                  -1, -1, 0,
```

```
                  0, 1, 1,
                  1, 0, -1
                  /), shape(fcc_conjugate))

        fcc_crossglide = reshape((/
                  -1, 0, -1,
                  1, 1, 0,
                  0, -1, 1
                  /), shape(fcc_crossglide))

        fcc_critical = reshape((/
                  -1, 0, 1,
                  0, 1, -1,
                  1, -1, 0
                  /), shape(fcc_critical))


        fcc_c11 = c11
        fcc_c12 = c12
        fcc_c44 = c44

        bassani_h0 = h0
        bassani_tau0 = tau0
        bassani_taus = taus
        bassani_hs = hs

        bassani_q = q
        !fcc_C = 0

        fcc_a1 = 8
        fcc_a2 = 8
        fcc_a3 = 8
        fcc_a4 = 15
        fcc_a5 = 20
    end subroutine

    subroutine elastic_moduli(C, c11, c12, c44)
    real, intent(in) :: c11, c12, c44
    real, dimension(:,:), intent(out) :: C

    integer :: i,j

    C(1,1) = c11
    C(2,2) = c11
    C(3,3) = c11

    C(4,4) = c44
    C(5,5) = c44
    C(6,6) = c44
```

```fortran
C(1,2) = c12
C(1,3) = c12
C(2,3) = c12

! isotropic
! c44 = (c11-c12)/2.0

do i = 1, 6
  do j = 1, i
    C(i, j) = C(j, i)
  end do
end do

end subroutine

! axb = c
subroutine cross_product(c, b, a)
integer, dimension(:), intent(in) :: b, a
integer, dimension(:), intent(out) :: c

c(1) = a(2)*b(3) - a(3)*b(2)
c(2) = a(3)*b(1) - a(1)*b(3)
c(3) = a(1)*b(2) - a(2)*b(1)

end subroutine

subroutine slip_system(slipsys, slipdir, slipnor)
integer, dimension(:), intent(in) :: slipdir, slipnor
real, dimension(:,:), intent(out) :: slipsys

integer, dimension(3) :: sliplin
real :: er, es, et
er = 0
es = 0
et = 0

call cross_product(sliplin, slipdir, slipnor)

er = sqrt(real(dot_product(slipnor, slipnor)))
es = sqrt(real(dot_product(slipdir, slipdir)))
et = sqrt(real(dot_product(sliplin, sliplin)))

slipsys(:,1) = slipnor/er
slipsys(:,2) = slipdir/es
slipsys(:,3) = sliplin/et

end subroutine

! (x, y, z) -> (r, s, t)
subroutine rotation_r(R, ex, ey, ez, er, es, et)
```

```
real, dimension(:), intent(in) :: ex, ey, ez
real, dimension(:), intent(in) :: er, es, et
real, dimension(:,:), intent(out) :: R

R(1,1) = dot_product(ex, er)
R(2,1) = dot_product(ex, es)
R(3,1) = dot_product(ex, et)

R(1,2) = dot_product(ey, er)
R(2,2) = dot_product(ey, es)
R(3,2) = dot_product(ey, et)

R(1,3) = dot_product(ez, er)
R(2,3) = dot_product(ez, es)
R(3,3) = dot_product(ez, et)

end subroutine

subroutine rotation_q(Q, ex, ey, ez, er, es, et)
real, dimension(:), intent(in) :: ex, ey, ez
real, dimension(:), intent(in) :: er, es, et
real, dimension(:,:), intent(out) :: Q

real :: l1, l2, l3
real :: m1, m2, m3
real :: n1, n2, n3
real, dimension(:,:), allocatable :: R
l1 = 0
l2 = 0
l3 = 0
m1 = 0
m2 = 0
m3 = 0
n1 = 0
n2 = 0
n3 = 0

allocate(R(3,3))
R = 0

l1 = dot_product(ex, er)
l2 = dot_product(ex, es)
l3 = dot_product(ex, et)

m1 = dot_product(ey, er)
m2 = dot_product(ey, es)
m3 = dot_product(ey, et)

n1 = dot_product(ez, er)
n2 = dot_product(ez, es)
```

```
n3 = dot_product(ez, et)

R(1,1) = l1
R(2,1) = l2
R(3,1) = l3
R(1,2) = m1
R(2,2) = m2
R(3,2) = m3
R(1,3) = n1
R(2,3) = n2
R(3,3) = n3

Q(1:3, 1:3) = R*R
! voigt
Q(4, 1) = l2*l3
Q(5, 1) = l1*l3
Q(6, 1) = l1*l2

Q(4, 2) = m2*m3
Q(5, 2) = m1*m3
Q(6, 2) = m1*m2

Q(4, 3) = n2*n3
Q(5, 3) = n1*n3
Q(6, 3) = n1*n2

Q(1, 4) = 2.0*m1*n1
Q(2, 4) = 2.0*m2*n2
Q(3, 4) = 2.0*m3*n3

Q(1, 5) = 2.0*l1*n1
Q(2, 5) = 2.0*l2*n2
Q(3, 5) = 2.0*l3*n3

Q(1, 6) = 2.0*l1*m1
Q(2, 6) = 2.0*l2*m2
Q(3, 6) = 2.0*l3*m3

Q(4, 4) = m2*n3 + n2*m3
Q(5, 4) = m1*n3 + n1*m3
Q(6, 4) = m1*n2 + n1*m2

Q(4, 5) = l2*n3 + n2*l3
Q(5, 5) = l1*n3 + n1*l3
Q(6, 5) = l1*n2 + n1*l2

Q(4, 6) = l2*m3 + m2*l3
Q(5, 6) = l1*m3 + m1*l3
Q(6, 6) = l1*m2 + m1*l2
```

```fortran
      deallocate(R)
      end subroutine

      subroutine slip_system_r(R, slipdir, slipnor)
      integer, dimension(:), intent(in) :: slipdir, slipnor
      real, dimension(:,:), intent(out) :: R

      real, dimension(:), allocatable :: ex, ey, ez
      real, dimension(:,:), allocatable :: slipsys
      allocate(ex(3), ey(3), ez(3))
      allocate(slipsys(3,3))

      ex = (/1, 0, 0/)
      ey = (/0, 1, 0/)
      ez = (/0, 0, 1/)
      slipsys = 0

      call slip_system(slipsys, slipdir, slipnor)

      call rotation_r(R, ex, ey, ez,
     1 slipsys(:,1), slipsys(:,2), slipsys(:,3))


      deallocate(slipsys)
      deallocate(ex, ey, ez)
      end subroutine

      subroutine slip_system_q(Q, slipdir, slipnor)
      integer, dimension(:), intent(in) :: slipdir, slipnor
      real, dimension(:,:), intent(out) :: Q

      real, dimension(:), allocatable :: ex, ey, ez
      real, dimension(:,:), allocatable :: slipsys
      allocate(ex(3), ey(3), ez(3))
      allocate(slipsys(3,3))

      ex = (/1, 0, 0/)
      ey = (/0, 1, 0/)
      ez = (/0, 0, 1/)
      slipsys = 0

      call slip_system(slipsys, slipdir, slipnor)

      call rotation_q(Q, ex, ey, ez,
     1 slipsys(:,1), slipsys(:,2), slipsys(:,3))


      deallocate(slipsys)
      deallocate(ex, ey, ez)
      end subroutine
```

```
subroutine slipsys_Schmid(schmid, slipdir, slipnor)
integer, dimension(:), intent(in) :: slipdir, slipnor
real, dimension(:,:), intent(out) :: schmid

real, dimension(:,:), allocatable :: slipsys
real, dimension(:), allocatable :: en, em
integer :: i, j
allocate(slipsys(3,3))
allocate(en(3), em(3))
slipsys = 0
en = 0
em = 0
call slip_system(slipsys, slipdir, slipnor)
en = slipsys(:,1)
em = slipsys(:,2)
do i = 1, 3
  do j = 1, 3
    schmid(i,j) = em(i)*en(j)
  end do
end do

deallocate(slipsys)
deallocate(en, em)
end subroutine

subroutine slipsys_symSchmid(symSchmid, slipdir, slipnor)
integer, dimension(:), intent(in) :: slipdir, slipnor
real, dimension(:,:), intent(out) :: symSchmid

real, dimension(:,:), allocatable :: schmid
allocate(schmid(3,3))
schmid = 0
call slipsys_Schmid(schmid, slipdir, slipnor)
symSchmid = 0.5*(schmid + transpose(schmid))
deallocate(schmid)
end subroutine

function schmid_shear(stress, slipdir, slipnor)
integer, dimension(:), intent(in) :: slipdir, slipnor
real, dimension(:,:), intent(in) :: stress
real :: schmid_shear

real :: shr1, shr2, shr3
real, dimension(:,:), allocatable :: symSchmid
allocate(symSchmid(3,3))
symSchmid = 0
call slipsys_symSchmid(symSchmid, slipdir, slipnor)
shr1 = 0
shr2 = 0
```

```fortran
shr3 = 0
shr1= dot_product(stress(:,1), symSchmid(:,1))
shr2= dot_product(stress(:,2), symSchmid(:,2))
shr3= dot_product(stress(:,3), symSchmid(:,3))

schmid_shear = shr1 + shr2 + shr3
deallocate(symSchmid)
end function

function hard(p, pgamma, h0, tau0, taus, hs, iflags)
real, intent(in) :: p
real, intent(in) :: h0, tau0, taus, hs
real, dimension(:), intent(in) :: pgamma
integer :: n
integer, dimension(:), intent(in) :: iflags
real :: hard

integer, dimension(:,:), allocatable :: hcoeff, icoeff
real :: p0
real :: F, G

integer, dimension(:), allocatable :: nslipsys
integer :: i, j

n = 0
do i = 1, 12
  if (iflags(i) >= 0) then
      n = n + 1
  end if
end do
!print *, n
allocate(nslipsys(n))

n = 0
do i = 1, 12
if (iflags(i) >= 0) then
      n = n + 1
      ! index from 1
      nslipsys(n) = i
  end if
end do


allocate(hcoeff(12,12))
allocate(icoeff(n, n))
hcoeff = 0
F = 0
G = 0
p0 = 0
icoeff = 0
```

```fortran
      if (taus < tau0) then
          print *, ' tau0 < taus '
          call abort
      end if
      if (h0 < hs) then
          print *, ' hs < h0 '
          call abort
      end if
      if (n > 12) then
          print *, " maximum slip systems: 12"
          call abort
      end if

      !asaro: hard = h0*(1.0/cosh(h0*p/(taus-tau0)))**2
      p0 = (taus-tau0)/h0
      F = (h0-hs)*(1.0/cosh((h0-hs)*p/(taus-tau0)))**2
&       + hs

      call hardcoeff_matrix(hcoeff,
1       fcc_a1, fcc_a2, fcc_a3, fcc_a4, fcc_a5)

      do i = 1, n
      do j = 1, n
        icoeff(i,j) = hcoeff(nslipsys(i), nslipsys(j))
      end do
      end do

      G = 1+sum(matmul(icoeff, tanh(pgamma/p0)))

      hard = F*G
      return
      end function


      subroutine hard_moduli(H, N, p, pgamma,
1       h0, tau0, taus, hs, q, iflags)
      real, intent(in) :: p
      real, dimension(:), intent(in) :: pgamma
      real, intent(in) :: h0, tau0, taus, hs, q
      integer, intent(in) :: N
      integer, dimension(:), intent(in) :: iflags

      real, dimension(:,:), intent(out) :: H

      integer :: i, j

      do i = 1, N
        do j = 1, N
          if (i .eq. j) then
```

```
             H(i, j) = hard(p, pgamma, h0, tau0, taus, hs,
1                           iflags)
         else
             H(i, j) = q*hard(p, pgamma, h0, tau0, taus, hs,
1                           iflags)
         end if
       end do
     end do

     end subroutine

     function hard_coeff(nslipdir1, nslipnor1,
1      nslipdir2, nslipnor2, a1, a2, a3, a4, a5)
     integer :: nslipdir1, nslipnor1
     integer :: nslipdir2, nslipnor2
     ! character :: a1, a2, a3, a4, a5
     integer :: a1, a2, a3, a4, a5
     integer :: a0
     ! character :: hard_coeff
     integer :: hard_coeff

     integer, dimension(:), allocatable :: slipdir1, slipdir2
     allocate(slipdir1(3), slipdir2(3))
     slipdir1 = 0
     slipdir2 = 0

     call slipsys_dir(slipdir1, nslipdir1, nslipnor1)
     call slipsys_dir(slipdir2, nslipdir2, nslipnor2)

     a0 = 0
     hard_coeff = 0

     a0 = dot_product(slipdir1, slipdir2)

     if (nslipnor1 .ne. nslipnor2) then
         if (a0 .eq. -2) then
             ! 'N : no junction'
             hard_coeff = a1
             return
         end if
         if (a0 .eq. 0) then
             ! 'H: Hirth lock'
             hard_coeff = a2
             return
         end if
         if (a0 .eq. 1) then
             ! 'G: glissile junction'
             hard_coeff = a4
             return
         end if
```

```
    if (a0 .eq. -1) then
        ! 'S: sessile junction'
        hard_coeff = a5
        return
    end if
end if


if (nslipnor1 .eq. nslipnor2) then
    if (a0 .eq. -1) then
      ! 'C: coplanar junction'
      hard_coeff = a3
      return
    end if
end if

deallocate(slipdir1, slipdir2)
return
end function

subroutine hardcoeff_matrix(hcoeff, a1, a2, a3, a4, a5)
!character, dimension(:,:), intent(out) :: hcoeff
integer, dimension(:,:), intent(out) :: hcoeff
!character, intent(in) :: a1, a2, a3, a4, a5
integer, intent(in) :: a1, a2, a3, a4, a5

integer :: i, j, k, l
integer :: m, n
!character, dimension(:,:,:,:), allocatable :: hcoeff1
integer, dimension(:,:,:,:), allocatable :: hcoeff1
allocate(hcoeff1(4, 3, 4, 3))
hcoeff1 = 1

do i = 1, 4
  do j = 1, 3
    do k = 1, 4
      do l = 1, 3
        hcoeff1(i, j, k, l) = hard_coeff(j-1, i-1, l-1, k-1,
&                                 a1, a2, a3, a4, a5)
        ! write(6, "(a3)") hcoeff1(i, j, k, l)
        m = 3*(i-1) + (j-1)
        n = 3*(k-1) + (l-1)
        hcoeff(m+1,n+1) = hcoeff1(i, j, k, l)
      end do
    end do
    ! print *
  end do
end do

deallocate(hcoeff1)
```

```fortran
end subroutine

subroutine slipsys_index(nslipdir, nslipnor, n)
integer, intent(in) :: n
integer, intent(out) :: nslipdir, nslipnor
nslipdir = 0
nslipnor = 0

if ( (n .lt. 0) .or. (n .ge. 12)) then
    print *, " slipsys index fault "
    call abort
end if

nslipnor = n/3
nslipdir = mod(n, 3)

end subroutine

subroutine slipsys_dir(slipdir, nslipdir, nslipnor)
integer, intent(in) :: nslipdir, nslipnor
integer, dimension(:), intent(out) :: slipdir
integer :: ndir
slipdir = 0
ndir = 0
if ( (nslipnor .lt. 0) .or. (nslipnor .ge. 4)) then
    print *, " slipnor index fault "
    call abort
end if
ndir = nslipdir + 1
if ( (nslipdir .lt. 0) .or. (nslipdir .ge. 3)) then
    print *, " slipdir index fault "
    call abort
end if

if (nslipnor .eq. 0) then
    slipdir = fcc_primary(:, ndir)
else if (nslipnor .eq. 1) then
    slipdir = fcc_conjugate(:, ndir)
else if (nslipnor .eq. 2) then
    slipdir = fcc_crossglide(:, ndir)
else if (nslipnor .eq. 3) then
    slipdir = fcc_critical(:, ndir)
end if

end subroutine

subroutine slipsys_nor(slipnor, nslipnor)
integer, intent(in) :: nslipnor
integer, dimension(:), intent(out) :: slipnor
integer :: nnor
```

```
slipnor = 0
nnor = 0
if ( (nslipnor .lt. 0) .or. (nslipnor .ge. 4)) then
    print *, " slipnor index fault "
    call abort
end if
nnor = nslipnor + 1
slipnor = fcc_slipnor(:, nnor)
end subroutine

subroutine voigt(v, symm)
real, dimension(:,:), intent(in) :: symm
real, dimension(:), intent(out) :: v

integer :: i, j

do i = 1, 3
  do j = 1, i
    if (j .eq. i) then
        v(i) = symm(i, j)
    else
        v(9-i-j) = symm(i, j)
    end if
  end do
end do

end subroutine

subroutine voigt2(symm, v)
real, dimension(:), intent(in) :: v
real, dimension(:,:), intent(out) :: symm

integer :: i, j

do i = 1, 3
  do j = 1, 3
    if (i .eq. j) then
        symm(i, j) = v(i)
    else
        symm(i, j) = v(9-i-j)
    end if
  end do
end do

end subroutine

subroutine invsym(a, n)
real, dimension(:,:), allocatable :: a
integer :: n
```

```
      character :: uplo
      integer :: nrhs
      integer :: lda
      integer, dimension(:), allocatable :: ipiv
      real, dimension(:,:), allocatable :: b
      integer :: ldb
      real, dimension(:), allocatable :: work
      integer :: lwork
      integer :: info

      real, dimension(:,:), allocatable :: x,y
      integer :: i, j

      uplo = 'U'
      info = -1
      lda = n

      ldb = n
      nrhs = n

      lwork = lda*n*ldb*nrhs

      allocate(b(ldb, nrhs))
      allocate(ipiv(n))
      allocate(work(lwork))
      ipiv = 0
      work = 0

      b = 0
      do i = 1, n
        do j = 1, n
          if (i==j) then
            b(i, j) = 1
          end if
        end do
      end do

      allocate(x(lda, n))
      allocate(y(ldb, nrhs))
      x = a
      y = b

      call ssysv(uplo, n, nrhs, a, lda, ipiv,
 1      b, ldb, work, lwork, info)

      if (info == 0) then
          a = b
          return
      end if
```

```
     if (info > 0) then
         print *, " singular matrix "
         a = x
         return
     end if

     if (info < 0) then
         print *,  -info, " illegal value "
         a = y
         return
     end if

     deallocate(b, ipiv, work)
     deallocate(x, y)
     end subroutine

     subroutine run_forward(neq, strains, t0, dt,
1      rparams, iparams)
     integer, intent(in) :: neq
     real, dimension(:) :: strains
     real, intent(in) :: t0
     real, intent(in) :: dt
     real, dimension(:), intent(in) :: rparams
     integer, dimension(:), intent(in) :: iparams

     integer :: i, j
     real :: t, h
     real, dimension(:), allocatable :: y0, ydot
     real, dimension(:), allocatable :: y1, y2
     real, dimension(:), allocatable :: y3, y4

     allocate(y0(neq), ydot(neq))
     allocate(y1(neq), y2(neq))
     allocate(y3(neq), y4(neq))
     y0 = 0
     ydot = 0
     y1 = 0
     y2 = 0
     y3 = 0
     y4 = 0

     y0 = strains(1:neq)
     y1 = y0
     y2 = y0
     y3 = y0
     y4 = y0

     ! rk4
       t = t0
       call shear_rate(neq, t, y0, ydot,
```

```
     1           rparams, iparams)
        y1 = ydot

        t = t0 + dt/2.0
        y2 = y0 + dt/2.0*y1
        call shear_rate(neq, t, y2, ydot,
     2           rparams, iparams)
        y2 = ydot

        t = t0 + dt/2.0
        y3 = y0 + dt/2.0*y2
        call shear_rate(neq, t, y3, ydot,
     3           rparams, iparams)
        y3 = ydot

        t = t0 + dt
        y4 = y0 + dt*y3
        call shear_rate(neq, t, y4, ydot,
     4           rparams, iparams)
        y4 = ydot



      strains(1:neq) = y0 + dt/6.0*(y1 + 2.0*y2 + 2.0*y3 + y4)

      deallocate(y0, ydot)
      deallocate(y1, y2)
      deallocate(y3, y4)
      end subroutine


      subroutine shear_rate(neq, t, strains, dstrains,
     1     rparams, iparams)
      integer, intent(in) :: neq
      real, intent(in) :: t
      real, dimension(:), intent(in) :: strains
      real, dimension(:), intent(in) :: rparams
      integer, dimension(:), intent(in) :: iparams
      real, dimension(:), intent(out) :: dstrains

      real, dimension(:), allocatable :: dstress
      integer, dimension(:), allocatable :: iflags
      real, dimension(:), allocatable :: nslipsys

      real, dimension(:), allocatable :: pgamma
      real, dimension(:,:), allocatable :: H
      real, dimension(:), allocatable :: dtau
      real, dimension(:), allocatable :: dgamma

      integer :: i, j, k, l
```

```
integer :: n

real, dimension(:), allocatable :: pstrain
real :: p
integer, dimension(:), allocatable :: slipdir, slipnor

real, dimension(:,:), allocatable :: v, Lp
real, dimension(:), allocatable :: dpstrain

allocate(dstress(6))
allocate(iflags(12))

dstress = rparams(1:6)

iflags = iparams(1:12)

n = 0
do i = 1, 12
  if (iflags(i) >= 0) then
      n = n + 1
  end if
end do
!print *, n
allocate(nslipsys(n))
allocate(pgamma(n))
allocate(H(n,n))
allocate(dtau(n))
allocate(dgamma(n))

n = 0
do i = 1, 12
if (iflags(i) >= 0) then
      n = n + 1
      ! index from 0
      nslipsys(n) = i-1
  end if
end do

allocate(pstrain(6))
pstrain = 0
p = 0

pstrain = strains(1:6)
p = strains(7)

allocate(slipdir(3), slipnor(3))
slipdir = 0
slipnor = 0

allocate(v(3,3), Lp(3,3))
```

```
allocate(dpstrain(6))
v = 0
Lp = 0
dpstrain = 0


pgamma = 0
dtau = 0
do j = 1, n
  i = nslipsys(j)
  call slipsys_index(k, l, i)
  !slipnor = fcc_slipnor(:, l+1)
  call slipsys_nor(slipnor, l)
  call slipsys_dir(slipdir, k, l)

  call voigt2(v, pstrain)
  ! shear strain
  pgamma(j) = schmid_shear(v, slipdir, slipnor)
  ! shear stress
  call voigt2(v, dstress)
  dtau(j) = schmid_shear(v, slipdir, slipnor)

end do

H = 0
call hard_moduli(H, n, p, pgamma,
bassani_h0, bassani_tau0, bassani_taus, bassani_hs, bassani_q,
iflags)
call invsym(H, n)

dgamma = 0
dgamma = matmul(H, dtau)

Lp = 0
do j = 1, n
  i = nslipsys(j)
  call slipsys_index(k, l, i)
  !slipnor = fcc_slipnor(:, l+1)
  call slipsys_nor(slipnor, l)
  call slipsys_dir(slipdir, k, l)

  call slipsys_symSchmid(v, slipdir, slipnor)
  Lp = Lp + dgamma(j)*v

end do

call voigt(dpstrain, Lp)

dstrains(1:6) = dpstrain
dstrains(7) = sum(abs(dgamma))
```

```
      deallocate(dstress, iflags, nslipsys)
      deallocate(pgamma, H, dtau, dgamma)
      deallocate(pstrain, slipdir, slipnor)
      deallocate(v, Lp)
      deallocate(dpstrain)
      end subroutine



C----------------------------------------------------------------------


    end module bassani1
```

Asaro2.f

```
      module asaro2
      implicit none

      integer, dimension(:,:), allocatable :: fcc_slipnor
      integer, dimension(:,:), allocatable :: fcc_primary
      integer, dimension(:,:), allocatable :: fcc_conjugate
      integer, dimension(:,:), allocatable :: fcc_crossglide
      integer, dimension(:,:), allocatable :: fcc_critical

      real, dimension(:,:), allocatable :: fcc_C

      real :: fcc_c11, fcc_c12, fcc_c44
      real :: asaro_h0, asaro_tau0, asaro_taus
      real :: asaro_q

      public :: fcc_slipnor
      public :: fcc_primary, fcc_conjugate
      public :: fcc_crossglide, fcc_critical

      private :: fcc_c11, fcc_c12, fcc_c44
      private :: asaro_h0, asaro_tau0, asaro_taus
      private :: asaro_q
    contains

      subroutine init(c11, c12, c44, h0, tau0, taus, q)
      real, intent(in) :: c11, c12, c44, h0, tau0, taus
      real, intent(in) :: q
          allocate(fcc_slipnor(3, 4))
          allocate(fcc_primary(3, 3))
          allocate(fcc_conjugate(3, 3))
          allocate(fcc_crossglide(3, 3))
          allocate(fcc_critical(3, 3))

          !allocate(fcc_C(6,6))
```

```
          fcc_slipnor = reshape((/
     1              1, 1, -1,
     2              1, -1, 1,
     3              1, -1, -1,
     4              1,  1,  1
     5             /), shape(fcc_slipnor))

          fcc_primary = reshape((/
     1               1, 0, 1,
     2              0, -1, -1,
     3              -1, 1, 0
     4              /), shape(fcc_primary))

          fcc_conjugate = reshape((/
     1              -1, -1, 0,
     2              0, 1, 1,
     3              1, 0, -1
     4              /), shape(fcc_conjugate))

          fcc_crossglide = reshape((/
     1              -1, 0, -1,
     2              1, 1, 0,
     3              0, -1, 1
     4              /), shape(fcc_crossglide))

          fcc_critical = reshape((/
     1              -1, 0, 1,
     2              0, 1, -1,
     3              1, -1, 0
     4              /), shape(fcc_critical))


          fcc_c11 = c11
          fcc_c12 = c12
          fcc_c44 = c44

          asaro_h0 = h0
          asaro_tau0 = tau0
          asaro_taus = taus

          asaro_q = q
          !fcc_C = 0

      end subroutine

      subroutine elastic_moduli(C, c11, c12, c44)
      real, intent(in) :: c11, c12, c44
      real, dimension(:,:), intent(out) :: C
```

```
integer :: i,j

C(1,1) = c11
C(2,2) = c11
C(3,3) = c11

C(4,4) = c44
C(5,5) = c44
C(6,6) = c44

C(1,2) = c12
C(1,3) = c12
C(2,3) = c12

! isotropic
! c44 = (c11-c12)/2.0

do i = 1, 6
  do j = 1, i
    C(i, j) = C(j, i)
  end do
end do

end subroutine

! axb = c
subroutine cross_product(c, b, a)
integer, dimension(:), intent(in) :: b, a
integer, dimension(:), intent(out) :: c

c(1) = a(2)*b(3) - a(3)*b(2)
c(2) = a(3)*b(1) - a(1)*b(3)
c(3) = a(1)*b(2) - a(2)*b(1)

end subroutine

subroutine slip_system(slipsys, slipdir, slipnor)
integer, dimension(:), intent(in) :: slipdir, slipnor
real, dimension(:,:), intent(out) :: slipsys

integer, dimension(3) :: sliplin
real :: er, es, et
er = 0
es = 0
et = 0

call cross_product(sliplin, slipdir, slipnor)

er = sqrt(real(dot_product(slipnor, slipnor)))
es = sqrt(real(dot_product(slipdir, slipdir)))
```

```
et = sqrt(real(dot_product(sliplin, sliplin)))

slipsys(:,1) = slipnor/er
slipsys(:,2) = slipdir/es
slipsys(:,3) = sliplin/et

end subroutine

! (x, y, z) -> (r, s, t)
subroutine rotation_r(R, ex, ey, ez, er, es, et)
real, dimension(:), intent(in) :: ex, ey, ez
real, dimension(:), intent(in) :: er, es, et
real, dimension(:,:), intent(out) :: R

R(1,1) = dot_product(ex, er)
R(2,1) = dot_product(ex, es)
R(3,1) = dot_product(ex, et)

R(1,2) = dot_product(ey, er)
R(2,2) = dot_product(ey, es)
R(3,2) = dot_product(ey, et)

R(1,3) = dot_product(ez, er)
R(2,3) = dot_product(ez, es)
R(3,3) = dot_product(ez, et)

end subroutine

subroutine rotation_q(Q, ex, ey, ez, er, es, et)
real, dimension(:), intent(in) :: ex, ey, ez
real, dimension(:), intent(in) :: er, es, et
real, dimension(:,:), intent(out) :: Q

real :: l1, l2, l3
real :: m1, m2, m3
real :: n1, n2, n3
real, dimension(:,:), allocatable :: R
l1 = 0
l2 = 0
l3 = 0
m1 = 0
m2 = 0
m3 = 0
n1 = 0
n2 = 0
n3 = 0

allocate(R(3,3))
R = 0
```

```
l1 = dot_product(ex, er)
l2 = dot_product(ex, es)
l3 = dot_product(ex, et)

m1 = dot_product(ey, er)
m2 = dot_product(ey, es)
m3 = dot_product(ey, et)

n1 = dot_product(ez, er)
n2 = dot_product(ez, es)
n3 = dot_product(ez, et)

R(1,1) = l1
R(2,1) = l2
R(3,1) = l3
R(1,2) = m1
R(2,2) = m2
R(3,2) = m3
R(1,3) = n1
R(2,3) = n2
R(3,3) = n3

Q(1:3, 1:3) = R*R
! voigt
Q(4, 1) = l2*l3
Q(5, 1) = l1*l3
Q(6, 1) = l1*l2

Q(4, 2) = m2*m3
Q(5, 2) = m1*m3
Q(6, 2) = m1*m2

Q(4, 3) = n2*n3
Q(5, 3) = n1*n3
Q(6, 3) = n1*n2

Q(1, 4) = 2.0*m1*n1
Q(2, 4) = 2.0*m2*n2
Q(3, 4) = 2.0*m3*n3

Q(1, 5) = 2.0*l1*n1
Q(2, 5) = 2.0*l2*n2
Q(3, 5) = 2.0*l3*n3

Q(1, 6) = 2.0*l1*m1
Q(2, 6) = 2.0*l2*m2
Q(3, 6) = 2.0*l3*m3

Q(4, 4) = m2*n3 + n2*m3
Q(5, 4) = m1*n3 + n1*m3
```

```
      Q(6, 4) = m1*n2 + n1*m2

      Q(4, 5) = l2*n3 + n2*l3
      Q(5, 5) = l1*n3 + n1*l3
      Q(6, 5) = l1*n2 + n1*l2

      Q(4, 6) = l2*m3 + m2*l3
      Q(5, 6) = l1*m3 + m1*l3
      Q(6, 6) = l1*m2 + m1*l2

      deallocate(R)
      end subroutine

      subroutine slip_system_r(R, slipdir, slipnor)
      integer, dimension(:), intent(in) :: slipdir, slipnor
      real, dimension(:,:), intent(out) :: R

      real, dimension(:), allocatable :: ex, ey, ez
      real, dimension(:,:), allocatable :: slipsys
      allocate(ex(3), ey(3), ez(3))
      allocate(slipsys(3,3))

      ex = (/1, 0, 0/)
      ey = (/0, 1, 0/)
      ez = (/0, 0, 1/)
      slipsys = 0

      call slip_system(slipsys, slipdir, slipnor)

      call rotation_r(R, ex, ey, ez,
     1 slipsys(:,1), slipsys(:,2), slipsys(:,3))


      deallocate(slipsys)
      deallocate(ex, ey, ez)
      end subroutine

      subroutine slip_system_q(Q, slipdir, slipnor)
      integer, dimension(:), intent(in) :: slipdir, slipnor
      real, dimension(:,:), intent(out) :: Q

      real, dimension(:), allocatable :: ex, ey, ez
      real, dimension(:,:), allocatable :: slipsys
      allocate(ex(3), ey(3), ez(3))
      allocate(slipsys(3,3))

      ex = (/1, 0, 0/)
      ey = (/0, 1, 0/)
      ez = (/0, 0, 1/)
      slipsys = 0
```

```
     call slip_system(slipsys, slipdir, slipnor)

     call rotation_q(Q, ex, ey, ez,
1 slipsys(:,1), slipsys(:,2), slipsys(:,3))


     deallocate(slipsys)
     deallocate(ex, ey, ez)
     end subroutine

     subroutine slipsys_Schmid(schmid, slipdir, slipnor)
     integer, dimension(:), intent(in) :: slipdir, slipnor
     real, dimension(:,:), intent(out) :: schmid

     real, dimension(:,:), allocatable :: slipsys
     real, dimension(:), allocatable :: en, em
     integer :: i, j
     allocate(slipsys(3,3))
     allocate(en(3), em(3))
     slipsys = 0
     en = 0
     em = 0
     call slip_system(slipsys, slipdir, slipnor)
     en = slipsys(:,1)
     em = slipsys(:,2)
     do i = 1, 3
       do j = 1, 3
         schmid(i,j) = em(i)*en(j)
       end do
     end do

     deallocate(slipsys)
     deallocate(en, em)
     end subroutine

     subroutine slipsys_symSchmid(symSchmid, slipdir, slipnor)
     integer, dimension(:), intent(in) :: slipdir, slipnor
     real, dimension(:,:), intent(out) :: symSchmid

     real, dimension(:,:), allocatable :: schmid
     allocate(schmid(3,3))
     schmid = 0
     call slipsys_Schmid(schmid, slipdir, slipnor)
     symSchmid = 0.5*(schmid + transpose(schmid))
     deallocate(schmid)
     end subroutine

     function schmid_shear(stress, slipdir, slipnor)
     integer, dimension(:), intent(in) :: slipdir, slipnor
```

```
real, dimension(:,:), intent(in) :: stress
real :: schmid_shear

real :: shr1, shr2, shr3
real, dimension(:,:), allocatable :: symSchmid
allocate(symSchmid(3,3))
symSchmid = 0
call slipsys_symSchmid(symSchmid, slipdir, slipnor)
shr1 = 0
shr2 = 0
shr3 = 0
shr1= dot_product(stress(:,1), symSchmid(:,1))
shr2= dot_product(stress(:,2), symSchmid(:,2))
shr3= dot_product(stress(:,3), symSchmid(:,3))

schmid_shear = shr1 + shr2 + shr3
deallocate(symSchmid)
end function

function hard(p, h0, tau0, taus)
real, intent(in) :: p
real, intent(in) :: h0, tau0, taus
real :: hard

hard = h0*(1.0/cosh(h0*p/(taus-tau0)))**2

end function


subroutine hard_moduli(H, N, p, h0, tau0, taus, q)
real, intent(in) :: p
real, intent(in) :: h0, tau0, taus, q
integer, intent(in) :: N
real, dimension(:,:), intent(out) :: H

integer :: i, j

do i = 1, N
  do j = 1, N
    if (i .eq. j) then
        H(i, j) = hard(p, h0, tau0, taus)
    else
        H(i, j) = q*hard(p, h0, tau0, taus)
    end if
  end do
end do

end subroutine

subroutine slipsys_index(nslipdir, nslipnor, n)
```

```
integer, intent(in) :: n
integer, intent(out) :: nslipdir, nslipnor
nslipdir = 0
nslipnor = 0

if ( (n .lt. 0) .or. (n .ge. 12)) then
    print *, " slipsys index fault "
    call abort
end if

nslipnor = n/3
nslipdir = mod(n, 3)

end subroutine

subroutine slipsys_dir(slipdir, nslipdir, nslipnor)
integer, intent(in) :: nslipdir, nslipnor
integer, dimension(:), intent(out) :: slipdir
integer :: ndir
slipdir = 0
ndir = 0
if ( (nslipnor .lt. 0) .or. (nslipnor .ge. 4)) then
    print *, " slipnor index fault "
    call abort
end if
ndir = nslipdir + 1
if ( (nslipdir .lt. 0) .or. (nslipdir .ge. 3)) then
    print *, " slipdir index fault "
    call abort
end if

if (nslipnor .eq. 0) then
    slipdir = fcc_primary(:, ndir)
else if (nslipnor .eq. 1) then
    slipdir = fcc_conjugate(:, ndir)
else if (nslipnor .eq. 2) then
    slipdir = fcc_crossglide(:, ndir)
else if (nslipnor .eq. 3) then
    slipdir = fcc_critical(:, ndir)
end if

end subroutine

subroutine slipsys_nor(slipnor, nslipnor)
integer, intent(in) :: nslipnor
integer, dimension(:), intent(out) :: slipnor
integer :: nnor
slipnor = 0
nnor = 0
if ( (nslipnor .lt. 0) .or. (nslipnor .ge. 4)) then
```

```fortran
    print *, " slipnor index fault "
    call abort
end if
nnor = nslipnor + 1
slipnor = fcc_slipnor(:, nnor)
end subroutine

subroutine voigt(v, symm)
real, dimension(:,:), intent(in) :: symm
real, dimension(:), intent(out) :: v

integer :: i, j

do i = 1, 3
  do j = 1, i
    if (j .eq. i) then
        v(i) = symm(i, j)
    else
        v(9-i-j) = symm(i, j)
    end if
  end do
end do

end subroutine

subroutine voigt2(symm, v)
real, dimension(:), intent(in) :: v
real, dimension(:,:), intent(out) :: symm

integer :: i, j

do i = 1, 3
  do j = 1, 3
    if (i .eq. j) then
        symm(i, j) = v(i)
    else
        symm(i, j) = v(9-i-j)
    end if
  end do
end do

end subroutine

subroutine invsym(a, n)
real, dimension(:,:), allocatable :: a
integer :: n

character :: uplo
integer :: nrhs
integer :: lda
```

```
      integer, dimension(:), allocatable :: ipiv
      real, dimension(:,:), allocatable :: b
      integer :: ldb
      real, dimension(:), allocatable :: work
      integer :: lwork
      integer :: info

      real, dimension(:,:), allocatable :: x,y
      integer :: i, j

      uplo = 'U'
      info = -1
      lda = n

      ldb = n
      nrhs = n

      lwork = lda*n*ldb*nrhs

      allocate(b(ldb, nrhs))
      allocate(ipiv(n))
      allocate(work(lwork))
      ipiv = 0
      work = 0

      b = 0
      do i = 1, n
        do j = 1, n
          if (i==j) then
            b(i, j) = 1
          end if
        end do
      end do

      allocate(x(lda, n))
      allocate(y(ldb, nrhs))
      x = a
      y = b

      call ssysv(uplo, n, nrhs, a, lda, ipiv,
 1      b, ldb, work, lwork, info)

      if (info == 0) then
          a = b
          return
      end if

      if (info > 0) then
          print *, " singular matrix "
          a = x
```

```fortran
              return
          end if

          if (info < 0) then
              print *,  -info, " illegal value "
              a = y
              return
          end if

          deallocate(b, ipiv, work)
          deallocate(x, y)
          end subroutine

          subroutine run_forward(neq, strains, t0, dt,
     1      rparams, iparams)
          integer, intent(in) :: neq
          real, dimension(:) :: strains
          real, intent(in) :: t0
          real, intent(in) :: dt
          real, dimension(:), intent(in) :: rparams
          integer, dimension(:), intent(in) :: iparams

          integer :: i, j
          real :: t, h
          real, dimension(:), allocatable :: y0, ydot
          real, dimension(:), allocatable :: y1, y2
          real, dimension(:), allocatable :: y3, y4

          allocate(y0(neq), ydot(neq))
          allocate(y1(neq), y2(neq))
          allocate(y3(neq), y4(neq))
          y0 = 0
          ydot = 0
          y1 = 0
          y2 = 0
          y3 = 0
          y4 = 0

          y0 = strains(1:neq)
          y1 = y0
          y2 = y0
          y3 = y0
          y4 = y0

          ! rk4
            t = t0
            call shear_rate(neq, t, y0, ydot,
     1          rparams, iparams)
            y1 = ydot
```

```
    t = t0 + dt/2.0
    y2 = y0 + dt/2.0*y1
    call shear_rate(neq, t, y2, ydot,
        rparams, iparams)
    y2 = ydot

    t = t0 + dt/2.0
    y3 = y0 + dt/2.0*y2
    call shear_rate(neq, t, y3, ydot,
        rparams, iparams)
    y3 = ydot

    t = t0 + dt
    y4 = y0 + dt*y3
    call shear_rate(neq, t, y4, ydot,
        rparams, iparams)
    y4 = ydot



  strains(1:neq) = y0 + dt/6.0*(y1 + 2.0*y2 + 2.0*y3 + y4)

  deallocate(y0, ydot)
  deallocate(y1, y2)
  deallocate(y3, y4)
  end subroutine


  subroutine shear_rate(neq, t, strains, dstrains,
      rparams, iparams)
  integer, intent(in) :: neq
  real, intent(in) :: t
  real, dimension(:), intent(in) :: strains
  real, dimension(:), intent(in) :: rparams
  integer, dimension(:), intent(in) :: iparams
  real, dimension(:), intent(out) :: dstrains

  real, dimension(:), allocatable :: dstress
  integer, dimension(:), allocatable :: iflags
  real, dimension(:), allocatable :: nslipsys

  real, dimension(:), allocatable :: pgamma
  real, dimension(:,:), allocatable :: H
  real, dimension(:), allocatable :: dtau
  real, dimension(:), allocatable :: dgamma

  integer :: i, j, k, l
  integer :: n

  real, dimension(:), allocatable :: pstrain
```

```
real :: p
integer, dimension(:), allocatable :: slipdir, slipnor

real, dimension(:,:), allocatable :: v, Lp
real, dimension(:), allocatable :: dpstrain

allocate(dstress(6))
allocate(iflags(12))

dstress = rparams(1:6)

iflags = iparams(1:12)

n = 0
do i = 1, 12
  if (iflags(i) >= 0) then
      n = n + 1
  end if
end do
!print *, n
allocate(nslipsys(n))
allocate(pgamma(n))
allocate(H(n,n))
allocate(dtau(n))
allocate(dgamma(n))

n = 0
do i = 1, 12
if (iflags(i) >= 0) then
      n = n + 1
      ! index from 0
      nslipsys(n) = i-1
  end if
end do

allocate(pstrain(6))
pstrain = 0
p = 0

pstrain = strains(1:6)
p = strains(7)

allocate(slipdir(3), slipnor(3))
slipdir = 0
slipnor = 0

allocate(v(3,3), Lp(3,3))
allocate(dpstrain(6))
v = 0
Lp = 0
```

```fortran
      dpstrain = 0


      pgamma = 0
      dtau = 0
      do j = 1, n
        i = nslipsys(j)
        call slipsys_index(k, l, i)
        !slipnor = fcc_slipnor(:, l+1)
        call slipsys_nor(slipnor, l)
        call slipsys_dir(slipdir, k, l)

        call voigt2(v, pstrain)
        ! shear strain
        pgamma(j) = schmid_shear(v, slipdir, slipnor)
        ! shear stress
        call voigt2(v, dstress)
        dtau(j) = schmid_shear(v, slipdir, slipnor)

      end do

      H = 0
      call hard_moduli(H, n, p,
     1 asaro_h0, asaro_tau0, asaro_taus, asaro_q)
      call invsym(H, n)

      dgamma = 0
      dgamma = matmul(H, dtau)

      Lp = 0
      do j = 1, n
        i = nslipsys(j)
        call slipsys_index(k, l, i)
        !slipnor = fcc_slipnor(:, l+1)
        call slipsys_nor(slipnor, l)
        call slipsys_dir(slipdir, k, l)

        call slipsys_symSchmid(v, slipdir, slipnor)
        Lp = Lp + dgamma(j)*v

      end do

      call voigt(dpstrain, Lp)

      dstrains(1:6) = dpstrain
      dstrains(7) = sum(abs(dgamma))

      deallocate(dstress, iflags, nslipsys)
      deallocate(pgamma, H, dtau, dgamma)
      deallocate(pstrain, slipdir, slipnor)
```

```
        deallocate(v, Lp)
        deallocate(dpstrain)
        end subroutine


C----------------------------------------------------------------------


      end module asaro2
```

References

[1] HP Fortran Programmer's Reference, Edition, Fourth, 2003.

[2] Latent hardening in single crystals-I. Theory and experiments, Wu, Tien-Yue and Bassani, John L and Laird, Campbell, Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences, 435, 1893, 1–19, 1991, The Royal Society London.

[3] Latent hardening in single crystals. II. Analytical characterization and predictions, Bassani, John L and Wu, Tien-Yue, Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences, 435, 1893, 21–41, 1991, The Royal Society London.