Supplementary Material

The following Fortran 2003 code should be copied into a text editor window and saved under the filename "library.f95" in the same directory as the calling program (e.g., the sample program from Appendix C).

```
!###############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!---------------------------------
module mod_constants
implicit none
integer,  parameter :: DP = selected_real_kind(15,307)
real(DP), parameter :: PI = 3.14159265358979324_DP
end module mod_constants
!###############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!---------------------------------
module mod_utilities
use mod_constants
implicit none
private
public :: clip
public :: sgn
public :: det2
public :: det3
public :: det4
public :: angle
public :: numChar
public :: PolarToCartesian
public :: CartesianToPolar
public :: LatexSciNotation
public :: FiniteDifference
interface clip
    module procedure iclip
    module procedure dclip
end interface clip
interface sgn
    module procedure isgn
    module procedure dsgn
end interface sgn
contains
!===============================================================================
    elemental function iclip (i,j,k) result (m)
    implicit none
    integer, intent(in)  :: i, j, k
    integer              :: m
    if (i < j) then
        m = j
    else if (i > k) then
        m = k
    else
        m = i
    end if
    end function iclip
!===============================================================================
    elemental function dclip (x,y,z) result (f)
    implicit none
    real(DP), intent(in)  :: x, y, z
    real(DP)              :: f
```

```
    if (x < y) then
        f = y
    else if (x > z) then
        f = z
    else
        f = x
    end if
    end function dclip
!=======================================================================
    function isgn(i) result(j)
    implicit none
    integer, intent(in) :: i
    integer             :: j
    if (i > 0) then
        j = 1
    else if (i < 0) then
        j = -1
    else
        j = 0
    end if
    end function isgn
!=======================================================================
    function dsgn(x) result(f)
    implicit none
    real(DP), intent(in) :: x
    real(DP)             :: f
    real(DP), parameter  :: tol = 1.0e-8_DP
    if (x > tol) then
        f = 1.0_DP
    else if (x < -tol) then
        f = -1.0_DP
    else
        f = 0.0_DP
    end if
    end function dsgn
!=======================================================================
    function det2(arg_a) result(arg_det)
    real(DP), dimension(2,2), intent(in) :: arg_a
    real(DP)                             :: arg_det
    arg_det = arg_a(1,1) * arg_a(2,2) - arg_a(1,2) * arg_a(2,1)
    end function det2
!=======================================================================
    function det3(arg_a) result(arg_det)
    real(DP), dimension(3,3), intent(in) :: arg_a
    real(DP)                             :: arg_det
    arg_det =   arg_a(1,1) * arg_a(2,2) * arg_a(3,3) &
              + arg_a(1,3) * arg_a(2,1) * arg_a(3,2) &
              + arg_a(1,2) * arg_a(2,3) * arg_a(3,1) &
              - arg_a(1,1) * arg_a(2,3) * arg_a(3,2) &
              - arg_a(1,2) * arg_a(2,1) * arg_a(3,3) &
              - arg_a(1,3) * arg_a(2,2) * arg_a(3,1)
    end function det3
!=======================================================================
    function det4(arg_a) result(arg_det)
    real(DP), dimension(4,4), intent(in) :: arg_a
    real(DP)                             :: arg_det
    real(DP), dimension(4,3,3)           :: b
    integer                              :: j
    b(1,:,:) = arg_a(2:4,2:4)
    b(4,:,:) = arg_a(2:4,1:3)
```

```
    b(2,:,1) = arg_a(2:4,1)
    b(3,:,3) = arg_a(2:4,4)
    b(2,:,2:3) = arg_a(2:4,3:4)
    b(3,:,1:2) = arg_a(2:4,1:2)
    arg_det = 0.0_DP
    do j = 1, 4
        arg_det = arg_det + arg_a(1,j) * real((-1)**j,DP) * det3(b(j,:,:))
    end do
    end function det4
!=======================================================================================
    function angle(r) result(a)
    implicit none
    real(DP), dimension(2), intent(in) :: r
    real(DP)                           :: a, d
    d = sqrt(r(1)**2 + r(2)**2)
    if (d < 1.0e-9_DP) then
        a = 0.0d0
    else if (r(2) >= 0.0_DP) then
        a = acos(r(1) / d)
    else
        a = 2.0d0 * pi - acos(r(1) / d)
    end if
    end function angle
!=======================================================================================
    subroutine PolarToCartesian (t,vr,vt,vx,vy)
    implicit none
    real(DP), intent(in)  :: t, vr, vt
    real(DP), intent(out) :: vx, vy
    vx = vr * cos(t) - vt * sin(t)
    vy = vr * sin(t) + vt * cos(t)
    end subroutine PolarToCartesian
!=======================================================================================
    subroutine CartesianToPolar (t,vx,vy,vr,vt)
    implicit none
    real(DP), intent(in)  :: t, vx, vy
    real(DP), intent(out) :: vr, vt
    vr =  vx * cos(t) + vy * sin(t)
    vt = -vx * sin(t) + vy * cos(t)
    end subroutine CartesianToPolar
!=======================================================================================
    subroutine numChar(arg_n,arg_name)
    implicit none
    character(len=8)              :: long
    character(len=*), intent(out) :: arg_name
    integer,          intent(in)  :: arg_n
    integer                       :: i, m
    if (len(arg_name) > 8) then
        print*, 'Error 1 in {numChar}'
        stop
    end if
    m = 0
    do i = 0, len(arg_name) - 1
        m = m + 9 * 10**i
    end do
    if (arg_n < 0 .or. arg_n > m) then
        print*, 'Error 2 in {numChar}'
        stop
    end if
    write (long,100) arg_n
    arg_name = long(9-len(arg_name):8)
```

```
100 format (i8.8)
    end subroutine numChar
!================================================================================
    function LatexSciNotation(arg_x) result(arg_latex)
    real(DP), intent(in)  :: arg_x
    character(len=53)     :: arg_latex
    character(len=13)     :: mantis
    character(len=23)     :: sign_mantis
    character(len=1)      :: charac, sign_charac
    real(DP)              :: y, z
    integer               :: m
    real(DP), parameter   :: tol = 1.0e-12_DP
    if (arg_x < 0.0) then
        sign_mantis = '                       -'
    else
        sign_mantis = '\mbox{}\hspace{0.114in}'
    end if
    if (abs(arg_x) < tol) then
        arg_latex = ' '
    else
        y = log10(abs(arg_x))
        if (y < 0.0) then
            m = int(y) - 1
        else
            m = int(y)
        end if
        if (m < 0) then
            sign_charac = '-'
        else
            sign_charac = ' '
        end if
        z = abs(arg_x) / 10.0**m
        write (mantis,100) z
        call numChar(abs(m),charac)
        arg_latex = '$' // sign_mantis // mantis // ' \times 10^{' &
                        // sign_charac // charac // '}$'
    end if
100 format(f13.11)
    end function LatexSciNotation
!================================================================================
    function FiniteDifference(arg_x,arg_k,arg_h,func) result(arg_f)
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_x
    real(DP), intent(in) :: arg_h
    real(DP)             :: arg_f
    interface
        function func(dum_x) result(dum_f)
        use mod_constants
        real(DP), intent(in) :: dum_x
        real(DP)             :: dum_f
        end function func
    end interface
    select case(arg_k)
        case(0)
            arg_f = func(arg_x)
        case(1)
            arg_f = (func(arg_x + arg_h) - func(arg_x - arg_h)) / (2.0_DP * arg_h)
        case(2)
            arg_f = (func(arg_x + arg_h) - 2.0_DP * func(arg_x) + func(arg_x - arg_h)) / arg_h**2
        case default
```

```
            arg_f = 0.0_DP
    end select
    end function FiniteDifference
!=======================================================================
end module mod_utilities
!#######################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_fullmat
use mod_constants
implicit none
type, public :: typ_fullmat
    private
    integer                                :: n
    integer,  dimension(:),   allocatable :: i
    real(DP), dimension(:,:), allocatable :: a
    real(DP)                               :: det
contains
    procedure :: setMatrix => fullmat_setMatrix
    procedure :: solSystem => fullmat_solSystem
    procedure :: getDeterm => fullmat_getDeterm
end type typ_fullmat
private :: fullmat_setMatrix
private :: fullmat_solSystem
private :: fullmat_getDeterm
contains
!=======================================================================
    subroutine fullmat_setMatrix(this,arg_a)
    class(typ_fullmat),       intent(inout) :: this
    real(DP), dimension(:,:), intent(in)    :: arg_a
    real(DP), dimension(:,:), allocatable   :: u
    integer                                 :: i, k
    integer,                  parameter     :: nCut = 0
    if (size(arg_a,1) /= size(arg_a,2)) then
        print*, 'Error in {fullmat_setMatrix}'
        print*, 'Matrix is not square'
        stop
    end if
    this%n = size(arg_a,1)
    if (allocated(this%a)) then
        deallocate(this%a)
    end if
    if (allocated(this%i)) then
        deallocate(this%i)
    end if
    allocate(this%a(this%n,this%n))
    allocate(this%i(this%n))
    this%a = arg_a
    do i = 1, this%n
        this%i(i) = i
    end do
    if (this%n <= nCut) then
        allocate(u(this%n,this%n))
        u = this%a
    end if
    do i = 1, this%n - 1
        call fullmat_pivot(this,i)
        do k = i + 1, this%n
            this%a(this%i(k),i) = this%a(this%i(k),i) / this%a(this%i(i),i)
            this%a(this%i(k),i+1:this%n) = this%a(this%i(k),i+1:this%n) &
```

```
                                            - this%a(this%i(k),i) * this%a(this%i(i),i+1:this%n)
            if (this%n <= nCut) then
                u(this%i(k),:) = u(this%i(k),:) - this%a(this%i(k),i) * u(this%i(i),:)
            end if
        end do
        if (this%n <= nCut) then
            print 100, i
            call fullmat_print_matrix(u)
            print*
        end if
    end do
    this%det = 1.0_DP
    do i = 1, this%n
        this%det = this%det * this%a(this%i(i),i)
    end do
100 format('Gaussian elimination - pass: ', i1)
    end subroutine fullmat_setMatrix
!================================================================================
    subroutine fullmat_solSystem(this,arg_v)
    class(typ_fullmat),     intent(in)    :: this
    real(DP), dimension(:), intent(inout) :: arg_v
    real(DP), dimension(:), allocatable   :: x
    integer                               :: i, j
    allocate(x(this%n))
    do i = 1, this%n
        x(this%i(i)) = arg_v(this%i(i))
        do j = 1, i - 1
            x(this%i(i)) = x(this%i(i)) - this%a(this%i(i),j) * x(this%i(j))
        end do
    end do
    do i = this%n, 1, -1
        arg_v(i) = x(this%i(i))
        do j = i + 1, this%n
            arg_v(i) = arg_v(i) - this%a(this%i(i),j) * arg_v(j)
        end do
        arg_v(i) = arg_v(i) / this%a(this%i(i),i)
    end do
    deallocate(x)
    end subroutine fullmat_solSystem
!================================================================================
    function fullmat_getDeterm(this) result(arg_det)
    class(typ_fullmat),     intent(in)    :: this
    real(DP)                              :: arg_det
    arg_det = this%det
    end function fullmat_getDeterm
!================================================================================
    subroutine fullmat_pivot(this,arg_i)
    class(typ_fullmat), intent(inout) :: this
    integer,              intent(in)    :: arg_i
    real(DP)                            :: maxVal
    integer                             :: k, m, iTemp
    m = 0
    maxVal = 0.0_DP
    do k = arg_i, this%n
        if (abs(this%a(this%i(k),arg_i)) > maxVal) then
            m = k
            maxVal = abs(this%a(this%i(k),arg_i))
        end if
    end do
    iTemp = this%i(arg_i)
```

```
    this%i(arg_i) = this%i(m)
    this%i(m) = iTemp
    end subroutine fullmat_pivot
!=======================================================================
    subroutine fullmat_print_matrix(arg_a)
    real(DP), dimension(:,:), intent(in) :: arg_a
    integer                              :: i
    do i = 1, size(arg_a,1)
        print 100, arg_a(i,:)
    end do
100 format(10(1x, f10.5))
    end subroutine fullmat_print_matrix
!=======================================================================
end module mod_fullmat
!#######################################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_stokesEq2d
use mod_constants
use mod_utilities
implicit none
save
private
public :: stokesEq2d_Cartesian
public :: stokesEq2d_Polar
contains
!=======================================================================
    subroutine stokesEq2d_Cartesian(sub,filename)
    implicit none
    character(len=*), intent(in) :: filename
    integer                      :: j, jMax
    real(DP)                     :: a, b, h, x, xInc, xMin, xMax
    real(DP), dimension(2)       :: p, q, pMin, pMax
    real(DP), dimension(4)       :: e
    real(DP)                     :: eMax
    interface
        subroutine sub(dum_r,dum_f)
            use mod_constants
            real(DP), dimension(2),   intent(in)  :: dum_r
            real(DP), dimension(0:3), intent(out) :: dum_f
        end subroutine sub
    end interface
    eMax = 0.0_DP
    open(unit = 30, file = filename)
    h = 1.0e-4_DP
    pMin(1) = -2.0_DP
    pMax(1) =  2.0_DP
    pMin(2) =  0.2_DP
    pMax(2) =  4.0_DP
    jMax = 12
    write(30,100) h
    do j = 1, jMax
        call random_number(q)
        p = pMin + (pMax - pMin) * q
        call stokesEq2d_CartesianPoint(sub,h,p,e)
        write(30,200) p, abs(e)
        if (maxval(abs(e)) > eMax) then
            eMax = maxval(abs(e))
        end if
    end do
```

```
    xInc = 0.2_DP
    xMin = 1.2_DP
    xMax = 4.0_DP
    jMax = nint((xMax - xMin) / xInc)
    write(30,300) h
    do j = 0, jMax
        x = xMin + (xMax - xMin) * dble(j) / dble(jMax)
        call stokesEq2d_CartesianShear(sub,h,x,a,b)
        write(30,400) x, a, b, b / a - 1.0_DP
        if (abs(b / a - 1.0_DP) > eMax) then
            eMax = abs(b / a - 1.0_DP)
        end if
    end do
    write(30,500) eMax
    close(30)
100 format (/ 'Finite-difference check of the Stokes equations'      /  &
           'Relative errors (Cartesian coordinates)'                 /  &
           'Step size: h = ', d8.2                                   // &
           2x, 'position-x', 2x, 'position-y', 2x, 'momentum-x',        &
           2x, 'momentum-y', 2x, 'harmonic-p', 2x, 'continuity'      / &
           6(2x, 10('-')))
200 format (6(2x, d10.3))
300 format (/ 'Finite-difference check of the shear stress' /  &
           '(Cartesian coordinates)' / 'Step size: h = ', d8.2  // &
           2x, '           x', 16x, '  analytical', &
           2x, '   numerical',  2x, '  difference' / &
           2x, 12('-'), 14x, 3(2x, 12('-')))
400 format (2x, d12.5, 14x, 3(2x, d12.5))
500 format(/ 'Maximum relative error: ', e12.5)
    end subroutine stokesEq2d_Cartesian
!===============================================================================
    subroutine stokesEq2d_CartesianPoint(sub,arg_h,arg_p,arg_e)
    implicit none
    integer                             :: i, j
    real(DP),               intent(in)  :: arg_h
    real(DP), dimension(2), intent(in)  :: arg_p
    real(DP), dimension(4), intent(out) :: arg_e
    real(DP), dimension(2,-1:1,-1:1)    :: p
    real(DP), dimension(0:3,-1:1,-1:1)  :: f
    real(DP), dimension(2)              :: grd_pressure, lap_velocity
    real(DP)                            :: lap_pressure, div_velocity
    real(DP)                            :: pr_xx, vx_x
    real(DP)                            :: err_momentum_x, err_momentum_y
    real(DP)                            :: err_laplacianp, err_continuity
    interface
        subroutine sub(dum_r,dum_f)
        use mod_constants
        real(DP), dimension(2),   intent(in)  :: dum_r
        real(DP), dimension(0:3), intent(out) :: dum_f
        end subroutine sub
    end interface
    do i = -1, 1
    do j = -1, 1
        p(:,i,j) = arg_p + arg_h * (/ dble(i), dble(j) /)
        call sub(p(:,i,j),f(:,i,j))
    end do
    end do
    grd_pressure(1) = (f(0,1,0) - f(0,-1,0)) / (2.0_DP * arg_h)
    grd_pressure(2) = (f(0,0,1) - f(0,0,-1)) / (2.0_DP * arg_h)
    lap_velocity(1) = (f(1,1,0) + f(1,-1,0) + f(1,0,1) + f(1,0,-1) - 4.0_DP * f(1,0,0)) / arg_h**2
```

```
    lap_velocity(2) = (f(2,1,0) + f(2,-1,0) + f(2,0,1) + f(2,0,-1) - 4.0_DP * f(2,0,0)) / arg_h**2
    lap_pressure    = (f(0,1,0) + f(0,-1,0) + f(0,0,1) + f(0,0,-1) - 4.0_DP * f(0,0,0)) / arg_h**2
    div_velocity    = (f(1,1,0) - f(1,-1,0) + f(2,0,1) - f(2,0,-1)) / (2.0_DP * arg_h)
    vx_x            = (f(1,1,0) - f(1,-1,0)) / (2.0_DP *  arg_h)
    pr_xx           = (f(0,1,0) + f(0,-1,0) - 2.0_DP * f(0,0,0)) / arg_h**2
    err_momentum_x = abs((grd_pressure(1) - lap_velocity(1)) / grd_pressure(1))
    err_momentum_y = abs((grd_pressure(2) - lap_velocity(2)) / grd_pressure(2))
    err_laplacianp = abs(lap_pressure / pr_xx)
    err_continuity = abs(div_velocity / vx_x)
    arg_e(1) = err_momentum_x
    arg_e(2) = err_momentum_y
    arg_e(3) = err_laplacianp
    arg_e(4) = err_continuity
    end subroutine stokesEq2d_CartesianPoint
!================================================================================
    subroutine stokesEq2d_CartesianShear(sub,arg_h,arg_x,arg_a,arg_b)
    implicit none
    integer                             :: i, j
    real(DP), intent(in)                :: arg_h, arg_x
    real(DP), intent(out)               :: arg_a, arg_b
    real(DP), dimension(2)              :: r
    real(DP), dimension(0:3,-1:1,0:2)   :: f
    real(DP)                            :: vx_y, vy_x
    interface
        subroutine sub(dum_r,dum_f)
            use mod_constants
            real(DP), dimension(2),   intent(in)  :: dum_r
            real(DP), dimension(0:3), intent(out) :: dum_f
        end subroutine sub
    end interface
    do i = -1, 1
    do j =  0, 2
        r(1) = dble(i) * arg_h + arg_x
        r(2) = dble(j) * arg_h
        call sub(r,f(:,i,j))
    end do
    end do
    vx_y = (-3.0_DP * f(1,0,0) + 4.0_DP * f(1,0,1) - f(1,0,2)) / (2.0_DP * arg_h)
    vy_x = (f(2,1,0) - f(2,-1,0)) / (2.0_DP * arg_h)
    arg_a = f(3,0,0)
    arg_b = vx_y + vy_x
    end subroutine stokesEq2d_CartesianShear
!================================================================================
    subroutine stokesEq2d_Polar(sub,filename)
    implicit none
    character(len=*), intent(in) :: filename
    integer                      :: i, iMax, j, jMax
    real(DP)                     :: a, b, h, q, r, s, t
    real(DP)                     :: rInc, rMin, rMax
    real(DP)                     :: tInc, tMin, tMax
    real(DP)                     :: eMax
    real(DP), dimension(4)       :: e
    interface
        subroutine sub(dum_r,dum_t,dum_pr,dum_vr,dum_vt,dum_st)
            use mod_constants
            real(DP), intent(in)  :: dum_r, dum_t
            real(DP), intent(out) :: dum_pr, dum_vr, dum_vt, dum_st
        end subroutine sub
    end interface
    eMax = 0.0_DP
```

```
    open(unit = 30, file = filename)
    h = 1.0e-4_DP
    rMin = 0.2_DP
    rMax = 5.0_DP
    tMin = 0.0_DP
    tMax = pi
    jMax = 12
    write(30,100) h
    do j = 1, jMax
        call random_number(q)
        call random_number(s)
        r = rMin + (rMax - rMin) * q
        t = tMin + (tMax - tMin) * s
        call stokesEq2d_PolarPoint(sub,h,r,t,e)
        write(30,200) r, t, abs(e)
        if (maxval(abs(e)) > eMax) then
            eMax = maxval(abs(e))
        end if
    end do
    rInc = 0.2_DP
    rMin = 1.2_DP
    rMax = 4.0_DP
    tInc = pi / 6.0_DP
    tMin = 0.0_DP
    tMax = pi / 3.0_DP
    iMax = nint((tMax - tMin) / tInc)
    jMax = nint((rMax - rMin) / rInc)
    write(30,300) h
    do i = 0, iMax
    do j = 0, jMax
        t = tMin + (tMax - tMin) * dble(i) / dble(iMax)
        r = rMin + (rMax - rMin) * dble(j) / dble(jMax)
        call stokesEq2d_PolarShear(sub,h,r,t,a,b)
        write(30,400) r, t, a, b, b / a - 1.0_DP
        if (abs(b / a - 1.0_DP) > eMax) then
            eMax = abs(b / a - 1.0_DP)
        end if
    end do
    end do
    write(30,500) eMax
    close(30)
100 format (/ 'Finite-difference check of the Stokes equations'      /  &
            'Relative errors (polar coordinates)'                    /  &
            'Step size in r: h = ', d8.2                             // &
            2x, 'position-r', 2x, 'position-t', 2x, 'momentum-r',       &
            2x, 'momentum-t', 2x, 'harmonic-p', 2x, 'continuity'      / &
            6(2x, 10('-')))
200 format (6(2x, d10.3))
300 format (/ 'Finite-difference check of the shear stress' /  &
            '(Polar coordinates)' / 'Step size: h = ', d8.2  // &
            2x, '           r', 2x, '           t', &
            2x, '  analytical', 2x, '   numerical', &
            2x, '  difference' / 5(2x, 12('-')))
400 format (5(2x, d12.5))
500 format(/ 'Maximum relative error: ', e12.5)
    end subroutine stokesEq2d_Polar
!=========================================================================
    subroutine stokesEq2d_PolarPoint(sub,arg_h,arg_r,arg_t,arg_e)
    implicit none
    integer                             :: i, j
```

```
real(DP),               intent(in)  :: arg_h, arg_r, arg_t
real(DP), dimension(4), intent(out) :: arg_e
real(DP), dimension(-1:1)           :: r, t
real(DP), dimension(-1:1,-1:1)      :: pr, vr, vt, st
real(DP)                            :: pr_r, pr_rr, pr_t, pr_tt
real(DP)                            :: vr_r, vr_rr, vr_t, vr_tt
real(DP)                            :: vt_r, vt_rr, vt_t, vt_tt
real(DP)                            :: hr, ht
real(DP)                            :: lap_pressure, div_velocity
real(DP)                            :: lhs_momentum_r, lhs_momentum_t
real(DP)                            :: rhs_momentum_r, rhs_momentum_t
real(DP)                            :: err_momentum_r, err_momentum_t
real(DP)                            :: err_laplacianp, err_continuity
real(DP), parameter                 :: par_tol = 1.0d-6
interface
    subroutine sub (dum_r,dum_t,dum_pr,dum_vr,dum_vt,dum_st)
    use mod_constants
    real(DP), intent(in)  :: dum_r, dum_t
    real(DP), intent(out) :: dum_pr, dum_vr, dum_vt, dum_st
    end subroutine sub
end interface
if (arg_r < par_tol) then
    print*, 'Error 1 in {stokesEq2d_PolarPoint}'
    stop
end if
r = 0.0_DP
t = 0.0_DP
pr = 0.0_DP
vr = 0.0_DP
vt = 0.0_DP
hr = arg_h
ht = arg_h / arg_r
do i = -1, 1
    r(i) = arg_r + dble(i) * hr
    t(i) = arg_t + dble(i) * ht
end do
do i = -1, 1
do j = -1, 1
    call sub(r(i),t(j),pr(i,j),vr(i,j),vt(i,j),st(i,j))
end do
end do
pr_r  = (pr(1,0) - pr(-1,0)) / (2.0_DP * hr)
vr_r  = (vr(1,0) - vr(-1,0)) / (2.0_DP * hr)
vt_r  = (vt(1,0) - vt(-1,0)) / (2.0_DP * hr)
pr_t  = (pr(0,1) - pr(0,-1)) / (2.0_DP * ht)
vr_t  = (vr(0,1) - vr(0,-1)) / (2.0_DP * ht)
vt_t  = (vt(0,1) - vt(0,-1)) / (2.0_DP * ht)
pr_rr = (pr(1,0) - 2.0_DP * pr(0,0) + pr(-1,0)) / hr**2
vr_rr = (vr(1,0) - 2.0_DP * vr(0,0) + vr(-1,0)) / hr**2
vt_rr = (vt(1,0) - 2.0_DP * vt(0,0) + vt(-1,0)) / hr**2
pr_tt = (pr(0,1) - 2.0_DP * pr(0,0) + pr(0,-1)) / ht**2
vr_tt = (vr(0,1) - 2.0_DP * vr(0,0) + vr(0,-1)) / ht**2
vt_tt = (vt(0,1) - 2.0_DP * vt(0,0) + vt(0,-1)) / ht**2
div_velocity = vr_r + vr(0,0) / r(0) + vt_t / r(0)
lap_pressure = pr_rr + pr_r / r(0) + pr_tt / r(0)**2
lhs_momentum_r = pr_r
rhs_momentum_r = vr_rr + vr_r / r(0) - vr(0,0) / r(0)**2 + (vr_tt - 2.0_DP * vt_t) / r(0)**2
lhs_momentum_t = pr_t / r(0)
rhs_momentum_t = vt_rr + vt_r / r(0) - vt(0,0) / r(0)**2 + (vt_tt + 2.0_DP * vr_t) / r(0)**2
err_momentum_r = abs((lhs_momentum_r - rhs_momentum_r) / lhs_momentum_r)
```

```
    err_momentum_t = abs((lhs_momentum_t - rhs_momentum_t) / lhs_momentum_t)
    err_laplacianp = abs(lap_pressure * r(0)**2 / pr_tt)
    err_continuity = abs(div_velocity / (vr(0,0) / r(0)))
    arg_e(1) = err_momentum_r
    arg_e(2) = err_momentum_t
    arg_e(3) = err_laplacianp
    arg_e(4) = err_continuity
    end subroutine stokesEq2d_PolarPoint
!=============================================================================
    subroutine stokesEq2d_PolarShear(sub,arg_h,arg_r,arg_t,arg_a,arg_b)
    implicit none
    integer                         :: i, j
    real(DP), intent(in)            :: arg_h, arg_r, arg_t
    real(DP), intent(out)           :: arg_a, arg_b
    real(DP)                        :: r, t, ur_t, ut_r, ut
    real(DP), dimension(-1:1,-1:2) :: pr, vr, vt, st
    interface
        subroutine sub (dum_r,dum_t,dum_pr,dum_vr,dum_vt,dum_st)
            use mod_constants
            real(DP), intent(in)  :: dum_r, dum_t
            real(DP), intent(out) :: dum_pr, dum_vr, dum_vt, dum_st
        end subroutine sub
    end interface
    do i = -1, 1
    do j = -1, 2
        r = dble(i) * arg_h + arg_r
        t = dble(j) * arg_h + arg_t
        call sub(r,t,pr(i,j),vr(i,j),vt(i,j),st(i,j))
    end do
    end do
    ut = vt(0,0)
    ur_t = (vr(0,1) - vr(0,-1)) / (2.0_DP * arg_h)
!    ur_t = (-3.0_DP * vr(0,0) + 4.0_DP * vr(0,1) - vr(0,2)) / (2.0_DP * arg_h)
    ut_r = (vt(1,0) - vt(-1,0)) / (2.0_DP * arg_h)
    arg_a = st(0,0)
    arg_b = ut_r + (ur_t - ut) / arg_r
    end subroutine stokesEq2d_PolarShear
!=============================================================================
end module mod_stokesEq2d
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_generalSolutionPlus1
use mod_constants
use mod_utilities
use  mod_stokesEq2d
implicit none
private
public :: generalSolutionPlus1_SetAngle
public :: generalSolutionPlus1_Coeff_p
public :: generalSolutionPlus1_Coeff_r
public :: generalSolutionPlus1_Coeff_t
public :: generalSolutionPlus1_Coeff_s
public :: generalSolutionPlus1_Cartesian
public :: generalSolutionPlus1_Polar
public :: generalSolutionPlus1_Check
public :: generalSolutionPlus1_Boundary
real(DP)                :: sav_theta = PI / 2.0_DP
real(DP), dimension(10) :: sav_d
contains
```

```
!==============================================================================
    subroutine generalSolutionPlus1_SetAngle(angle)
    implicit none
    integer, intent(in) :: angle
    sav_theta = real(angle,DP) * PI / 180.0_DP
    end subroutine generalSolutionPlus1_SetAngle
!==============================================================================
    subroutine generalSolutionPlus1_Coeff_p(t,fp0,fp1,fp2)
    implicit none
    real(DP),                intent(in)  :: t
    real(DP), dimension(10), intent(out) :: fp0, fp1, fp2
    fp0 = 0.0_DP
    fp1 = 0.0_DP
    fp2 = 0.0_DP
!
    fp2(4) = -4.0_DP / 3.0_DP
!
    fp0( 3) =  8.0_DP * t
    fp0( 4) =  4.0_DP * t**2 + 8.0_DP
    fp0(10) = -4.0_DP
    end subroutine generalSolutionPlus1_Coeff_p
!==============================================================================
    subroutine generalSolutionPlus1_Coeff_r(t,fr0,fr1,fr2)
    implicit none
    real(DP),                intent(in)  :: t
    real(DP), dimension(10), intent(out) :: fr0, fr1, fr2
    real(DP)                             :: c, s
    c = cos(2 * t)
    s = sin(2 * t)
    fr0 = 0.0_DP
    fr1 = 0.0_DP
    fr2 = 0.0_DP
!
    fr2(1) =  2.0_DP * s
    fr2(2) = -2.0_DP * c
    fr2(4) = -1.0_DP
!
    fr1(1) =  4.0_DP * t * c + 2.0_DP * s
    fr1(2) =  4.0_DP * t * s - 2.0_DP * c
    fr1(4) =  2.0_DP
    fr1(5) =  2.0_DP * s
    fr1(6) = -2.0_DP * c
!
    fr0( 1) = -2.0_DP * t**2 * s + 2.0_DP * t * c
    fr0( 2) =  2.0_DP * t**2 * c + 2.0_DP * t * s
    fr0( 3) =  2.0_DP * t
    fr0( 4) =  t**2
    fr0( 5) =  2.0_DP * t * c + s
    fr0( 6) =  2.0_DP * t * s - c
    fr0( 7) =  2.0_DP * s
    fr0( 8) = -2.0_DP * c
    fr0( 9) =  0.0_DP
    fr0(10) = -1.0_DP
    end subroutine generalSolutionPlus1_Coeff_r
!==============================================================================
    subroutine generalSolutionPlus1_Coeff_t(t,ft0,ft1,ft2)
    implicit none
    real(DP),                intent(in)  :: t
    real(DP), dimension(10), intent(out) :: ft0, ft1, ft2
    real(DP)                             :: c, s
```

```
    c = cos(2 * t)
    s = sin(2 * t)
    ft0 = 0.0_DP
    ft1 = 0.0_DP
    ft2 = 0.0_DP
!
    ft2(1) = 2.0_DP * c
    ft2(2) = 2.0_DP * s
    ft2(3) = 2.0_DP
    ft2(4) = 2.0_DP * t
!
    ft1(1) = -4.0_DP * t * s + 2.0_DP * c
    ft1(2) =  4.0_DP * t * c + 2.0_DP * s
    ft1(3) = -2.0_DP
    ft1(4) = -2.0_DP * t
    ft1(5) =  2.0_DP * c
    ft1(6) =  2.0_DP * s
!
    ft0( 1) = -2.0_DP * t**2 * c - 2.0_DP * t * s
    ft0( 2) = -2.0_DP * t**2 * s + 2.0_DP * t * c
    ft0( 3) = -2.0_DP - 2.0_DP * t**2
    ft0( 4) = -2.0_DP * t - 2.0_DP * t**3 / 3.0_DP
    ft0( 5) = -2.0_DP * t * s + c
    ft0( 6) =  2.0_DP * t * c + s
    ft0( 7) =  2.0_DP * c
    ft0( 8) =  2.0_DP * s
    ft0( 9) =  2.0_DP
    ft0(10) =  2.0_DP * t
    end subroutine generalSolutionPlus1_Coeff_t
!================================================================================
    subroutine generalSolutionPlus1_Coeff_s(t,fs0,fs1,fs2)
    implicit none
    real(DP),                intent(in)  :: t
    real(DP), dimension(10), intent(out) :: fs0, fs1, fs2
    real(DP)                             :: c, s
    c = cos(2 * t)
    s = sin(2 * t)
    fs0 = 0.0_DP
    fs1 = 0.0_DP
    fs2 = 0.0_DP
!
    fs2(1) = 4.0_DP * c
    fs2(2) = 4.0_DP * s
!
    fs1(1) = 12.0_DP * c - 8.0_DP * t * s
    fs1(2) = 12.0_DP * s + 8.0_DP * t * c
    fs1(3) = 4.0_DP
    fs1(4) = 4.0_DP * t
    fs1(5) = 4.0_DP * c
    fs1(6) = 4.0_DP * s
!
    fs0(1) = 4.0_DP * c - 12.0_DP * t * s - 4.0_DP * t**2 * c
    fs0(2) = 4.0_DP * s + 12.0_DP * t * c - 4.0_DP * t**2 * s
    fs0(5) = 6.0_DP * c -  4.0_DP * t * s
    fs0(6) = 6.0_DP * s +  4.0_DP * t * c
    fs0(7) = 4.0_DP * c
    fs0(8) = 4.0_DP * s
    end subroutine generalSolutionPlus1_Coeff_s
!================================================================================
    subroutine generalSolutionPlus1_Cartesian(d,p,f)
```

```
    real(DP), dimension(10),  intent(in)  :: d
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(p,p))
    t = angle(p)
    call generalSolutionPlus1_Polar(d,r,t,pr,vr,vt,st)
    call PolarToCartesian(t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine generalSolutionPlus1_Cartesian
!=============================================================================
    subroutine generalSolutionPlus1_Polar(d,r,t,pr,vr,vt,st)
    implicit none
    real(DP), dimension(10), intent(in)  :: d
    real(DP),                intent(in)  :: r, t
    real(DP),                intent(out) :: pr, vr, vt, st
    real(DP), dimension(10)              :: fs0, fs1, fs2
    real(DP), dimension(10)              :: fp0, fp1, fp2
    real(DP), dimension(10)              :: fr0, fr1, fr2
    real(DP), dimension(10)              :: ft0, ft1, ft2
    real(DP)                             :: lnr1, lnr2
    lnr1 = log(r)
    lnr2 = log(r)**2
!
    call generalSolutionPlus1_Coeff_s(t,fs0,fs1,fs2)
    call generalSolutionPlus1_Coeff_p(t,fp0,fp1,fp2)
    call generalSolutionPlus1_Coeff_r(t,fr0,fr1,fr2)
    call generalSolutionPlus1_Coeff_t(t,ft0,ft1,ft2)
!
    pr = (dot_product(d,fp2) * lnr2 + dot_product(d,fp1) * lnr1 + dot_product(d,fp0)) * lnr1
    vr = (dot_product(d,fr2) * lnr2 + dot_product(d,fr1) * lnr1 + dot_product(d,fr0)) * r
    vt = (dot_product(d,ft2) * lnr2 + dot_product(d,ft1) * lnr1 + dot_product(d,ft0)) * r
    st =  dot_product(d,fs2) * lnr2 + dot_product(d,fs1) * lnr1 + dot_product(d,fs0)
    end subroutine generalSolutionPlus1_Polar
!=============================================================================
    subroutine generalSolutionPlus1_Boundary(d)
    implicit none
    real(DP), dimension(10), intent(in) :: d
    real(DP), dimension(10)             :: fs0, fs1, fs2
    real(DP), dimension(10)             :: fp0, fp1, fp2
    real(DP), dimension(10)             :: fr0, fr1, fr2
    real(DP), dimension(10)             :: ft0, ft1, ft2
    real(DP), dimension(10)             :: gs0, gs1, gs2
    real(DP), dimension(10)             :: gp0, gp1, gp2
    real(DP), dimension(10)             :: gr0, gr1, gr2
    real(DP), dimension(10)             :: gt0, gt1, gt2
    real(DP), dimension(10)             :: hs0, hs1, hs2
    real(DP), dimension(10)             :: hp0, hp1, hp2
    real(DP), dimension(10)             :: hr0, hr1, hr2
    real(DP), dimension(10)             :: ht0, ht1, ht2
    real(DP)                            :: t1, t2, t3
    integer                             :: a1, a2, a3
    t1 = 0.0_DP
    t2 = sav_theta / 2.0_DP
    t3 = sav_theta
    a1 = nint(t1 * 180.0_DP / PI)
    a2 = nint(t2 * 180.0_DP / PI)
    a3 = nint(t3 * 180.0_DP / PI)
!
    call generalSolutionPlus1_Coeff_s(t1,fs0,fs1,fs2)
```

```
    call generalSolutionPlus1_Coeff_p(t1,fp0,fp1,fp2)
    call generalSolutionPlus1_Coeff_r(t1,fr0,fr1,fr2)
    call generalSolutionPlus1_Coeff_t(t1,ft0,ft1,ft2)
!
    call generalSolutionPlus1_Coeff_s(t2,gs0,gs1,gs2)
    call generalSolutionPlus1_Coeff_p(t2,gp0,gp1,gp2)
    call generalSolutionPlus1_Coeff_r(t2,gr0,gr1,gr2)
    call generalSolutionPlus1_Coeff_t(t2,gt0,gt1,gt2)
!
    call generalSolutionPlus1_Coeff_s(t3,hs0,hs1,hs2)
    call generalSolutionPlus1_Coeff_p(t3,hp0,hp1,hp2)
    call generalSolutionPlus1_Coeff_r(t3,hr0,hr1,hr2)
    call generalSolutionPlus1_Coeff_t(t3,ht0,ht1,ht2)
!
    print 300
    print 100, 'st', a1, dot_product(d,fs2), dot_product(d,fs1), dot_product(d,fs0), '     '
    print 100, 'pr', a1, dot_product(d,fp2), dot_product(d,fp1), dot_product(d,fp0), 'ln(r)'
    print 100, 'vr', a1, dot_product(d,fr2), dot_product(d,fr1), dot_product(d,fr0), 'r    '
    print 100, 'vt', a1, dot_product(d,ft2), dot_product(d,ft2), dot_product(d,ft2), 'r    '
    print 200
    print 100, 'st', a3, dot_product(d,hs2), dot_product(d,hs1), dot_product(d,hs0), '     '
    print 100, 'pr', a3, dot_product(d,hp2), dot_product(d,hp1), dot_product(d,hp0), 'ln(r)'
    print 100, 'vr', a3, dot_product(d,hr2), dot_product(d,hr1), dot_product(d,hr0), 'r    '
    print 100, 'vt', a3, dot_product(d,ht2), dot_product(d,ht1), dot_product(d,ht0), 'r    '
    print 300
100 format (a, '(r,', i3, ') = [(', f10.6, ') ln(r)^2 + (', f10.6, ') ln(r) + (,',  f10.6, ')] ', a,  ')')
200 format ()
300 format (79('-'))
!
    end subroutine generalSolutionPlus1_Boundary
!=========================================================================
    subroutine generalSolutionPlus1_Check
    implicit none
    sav_d(1: 4) = (/  1.37_DP,  5.76_DP, -6.32_DP, -7.98_DP /)
    sav_d(5: 6) = (/  4.84_DP, -4.21_DP                     /)
    sav_d(7:10) = (/ -7.51_DP,  3.54_DP,  4.03_DP,  3.72_DP /)
    call stokesEq2d_Cartesian(generalSolutionPlus1_FieldCartesian,'generalSolutionPlus1_Cartesian.txt')
    call stokesEq2d_Polar(generalSolutionPlus1_FieldPolar,'generalSolutionPlus1_Polar.txt')
    end subroutine generalSolutionPlus1_Check
!=========================================================================
    subroutine generalSolutionPlus1_FieldCartesian(p,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    call generalSolutionPlus1_Cartesian(sav_d,p,f)
    end subroutine generalSolutionPlus1_FieldCartesian
!=========================================================================
    subroutine generalSolutionPlus1_FieldPolar(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    call generalSolutionPlus1_Polar(sav_d,r,t,pr,vr,vt,st)
    end subroutine generalSolutionPlus1_FieldPolar
!=========================================================================
end module mod_generalSolutionPlus1
!#########################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_generalSolutionMinus1
use mod_constants
```

```
use mod_utilities
use mod_stokesEq2d
implicit none
private
public :: generalSolutionMinus1_SetAngle
public :: generalSolutionMinus1_Coeff_p
public :: generalSolutionMinus1_Coeff_r
public :: generalSolutionMinus1_Coeff_t
public :: generalSolutionMinus1_Coeff_s
public :: generalSolutionMinus1_Cartesian
public :: generalSolutionMinus1_Polar
public :: generalSolutionMinus1_Check
public :: generalSolutionMinus1_Boundary
real(DP)                :: sav_theta = PI / 2.0_DP
real(DP), dimension(11) :: sav_d
contains
!===============================================================================
    subroutine generalSolutionMinus1_SetAngle(angle)
    implicit none
    integer, intent(in) :: angle
    sav_theta = real(angle,DP) * PI / 180.0_DP
    end subroutine generalSolutionMinus1_SetAngle
!===============================================================================
    subroutine generalSolutionMinus1_Coeff_p(t,fp0,fp1,fp2)
    implicit none
    real(DP)                              :: s, c
    real(DP),                intent(in)   :: t
    real(DP), dimension(11), intent(out)  :: fp0, fp1, fp2
    real(DP), dimension(4)                :: fpa0, fpa1, fpa2, fpb0, fpb1, fpc0
    c = cos(2.0_DP * t)
    s = sin(2.0_DP * t)
    fpa2 = 0.0_DP
    fpa1 = 0.0_DP
    fpa0 = 0.0_DP
    fpb1 = 0.0_DP
    fpb0 = 0.0_DP
    fpc0 = 0.0_DP
    fpa2(1) =  4.0_DP * s
    fpa2(2) = -4.0_DP * c
    fpa1(1) = -8.0_DP * t * c - 8.0_DP * s
    fpa1(2) = -8.0_DP * t * s + 8.0_DP * c
    fpa0(1) = -4.0_DP * t**2 * s + 8.0_DP * t * c
    fpa0(2) =  4.0_DP * t**2 * c + 8.0_DP * t * s
    fpb0(1) = -4.0_DP * t * c - 4.0_DP * s
    fpb0(2) = -4.0_DP * t * s + 4.0_DP * c
    fpb1 = fpa2
    fpc0 = fpa2
    fp0 = 0.0_DP
    fp1 = 0.0_DP
    fp2 = 0.0_DP
    fp2(1: 4) = fpa2
    fp1(1: 4) = fpa1
    fp1(5: 8) = fpb1
    fp0(1: 4) = fpa0
    fp0(5: 8) = fpb0
    fp0( 9) = fpc0(1)
    fp0(10) = fpc0(2)
    fp0(11) = fpc0(4)
    end subroutine generalSolutionMinus1_Coeff_p
!===============================================================================
```

```
    subroutine generalSolutionMinus1_Coeff_r(t,fr0,fr1,fr2)
    implicit none
    real(DP)                              :: s, c
    real(DP),                intent(in)   :: t
    real(DP), dimension(11), intent(out)  :: fr0, fr1, fr2
    real(DP), dimension(4)                :: fra0, fra1, fra2, frb0, frb1, frc0
    c = cos(2.0_DP * t)
    s = sin(2.0_DP * t)
    fra2 = 0.0_DP
    fra1 = 0.0_DP
    fra0 = 0.0_DP
    frb1 = 0.0_DP
    frb0 = 0.0_DP
    frc0 = 0.0_DP
    fra2(1) =  2.0_DP * s
    fra2(2) = -2.0_DP * c
    fra2(4) = -1.0_DP
    fra1(1) = -4.0_DP * t * c - 2.0_DP * s
    fra1(2) = -4.0_DP * t * s + 2.0_DP * c
    fra0(1) = -2.0_DP * t**2 * s + 2.0_DP * t * c
    fra0(2) =  2.0_DP * t**2 * c + 2.0_DP * t * s
    fra0(3) =  2.0_DP * t
    fra0(4) =  t**2
    frb0(1) = -2.0_DP * t * c - s
    frb0(2) = -2.0_DP * t * s + c
    frb1 = fra2
    frc0 = fra2
    fr0 = 0.0_DP
    fr1 = 0.0_DP
    fr2 = 0.0_DP
    fr2(1: 4) = fra2
    fr1(1: 4) = fra1
    fr1(5: 8) = frb1
    fr0(1: 4) = fra0
    fr0(5: 8) = frb0
    fr0( 9) = frc0(1)
    fr0(10) = frc0(2)
    fr0(11) = frc0(4)
    end subroutine generalSolutionMinus1_Coeff_r
!===============================================================================
    subroutine generalSolutionMinus1_Coeff_t(t,ft0,ft1,ft2)
    implicit none
    real(DP)                              :: s, c
    real(DP),                intent(in)   :: t
    real(DP), dimension(11), intent(out)  :: ft0, ft1, ft2
    real(DP), dimension(4)                :: fta0, fta1, fta2, ftb0, ftb1, ftc0
    c = cos(2.0_DP * t)
    s = sin(2.0_DP * t)
    fta2 = 0.0_DP
    fta1 = 0.0_DP
    fta0 = 0.0_DP
    ftb1 = 0.0_DP
    ftb0 = 0.0_DP
    ftc0 = 0.0_DP
    fta1(1) =  2.0_DP * c
    fta1(2) =  2.0_DP * s
    fta1(3) =  2.0_DP
    fta1(4) =  2.0_DP * t
    fta0(1) =  2.0_DP * t * s
    fta0(2) = -2.0_DP * t * c
```

```
    ftb0(1) = c
    ftb0(2) = s
    ftb0(3) = 1.0_DP
    ftb0(4) = t
    ft0 = 0.0_DP
    ft1 = 0.0_DP
    ft2 = 0.0_DP
    ft2(1: 4) = fta2
    ft1(1: 4) = fta1
    ft1(5: 8) = ftb1
    ft0(1: 4) = fta0
    ft0(5: 8) = ftb0
    ft0( 9) = ftc0(1)
    ft0(10) = ftc0(2)
    ft0(11) = ftc0(4)
    end subroutine generalSolutionMinus1_Coeff_t
!===============================================================================
    subroutine generalSolutionMinus1_Coeff_s(t,fs0,fs1,fs2)
    implicit none
    real(DP)                              :: s, c
    real(DP),                intent(in)   :: t
    real(DP), dimension(11), intent(out)  :: fs0, fs1, fs2
    real(DP), dimension(4)                :: fsa0, fsa1, fsa2, fsb0, fsb1, fsc0
    c = cos(2.0_DP * t)
    s = sin(2.0_DP * t)
    fsa2 = 0.0_DP
    fsa1 = 0.0_DP
    fsa0 = 0.0_DP
    fsb1 = 0.0_DP
    fsb0 = 0.0_DP
    fsc0 = 0.0_DP
    fsa2(1) =  4.0_DP * c
    fsa2(2) =  4.0_DP * s
    fsa1(1) =  8.0_DP * t * s - 12.0_DP * c
    fsa1(2) = -8.0_DP * t * c - 12.0_DP * s
    fsa1(3) = -4.0_DP
    fsa1(4) = -4.0_DP * t
    fsa0(1) = -4.0_DP * t**2 * c - 12.0_DP * t * s + 4.0_DP * c
    fsa0(2) = -4.0_DP * t**2 * s + 12.0_DP * t * c + 4.0_DP * s
    fsa0(3) =  4.0_DP
    fsa0(4) =  4.0_DP * t
    fsb0(1) =  4.0_DP * t * s - 6.0_DP * c
    fsb0(2) = -4.0_DP * t * c - 6.0_DP * s
    fsb0(3) = -2.0_DP
    fsb0(4) = -2.0_DP * t
    fsb1 = fsa2
    fsc0 = fsa2
    fs0 = 0.0_DP
    fs1 = 0.0_DP
    fs2 = 0.0_DP
    fs2(1: 4) = fsa2
    fs1(1: 4) = fsa1
    fs1(5: 8) = fsb1
    fs0(1: 4) = fsa0
    fs0(5: 8) = fsb0
    fs0( 9) = fsc0(1)
    fs0(10) = fsc0(2)
    fs0(11) = fsc0(4)
    end subroutine generalSolutionMinus1_Coeff_s
!===============================================================================
```

```
    subroutine generalSolutionMinus1_Cartesian(d,p,f)
    real(DP), dimension(11),  intent(in)  :: d
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(p,p))
    t = angle(p)
    call generalSolutionMinus1_Polar(d,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine generalSolutionMinus1_Cartesian
!================================================================================
    subroutine generalSolutionMinus1_Polar(d,r,t,pr,vr,vt,st)
    implicit none
    real(DP),                parameter   :: TOL = 1.0e-8_DP
    real(DP), dimension(11), intent(in)  :: d
    real(DP),                intent(in)  :: r, t
    real(DP),                intent(out) :: pr, vr, vt, st
    real(DP), dimension(11)              :: fs0, fs1, fs2
    real(DP), dimension(11)              :: fp0, fp1, fp2
    real(DP), dimension(11)              :: fr0, fr1, fr2
    real(DP), dimension(11)              :: ft0, ft1, ft2
    real(DP)                             :: lnr1, lnr2
    if (r < TOL) then
        st = 0.0_DP
        pr = 0.0_DP
        vr = 0.0_DP
        vt = 0.0_DP
    else
        lnr2 = log(r)**2
        lnr1 = log(r)
!
        call generalSolutionMinus1_Coeff_s(t,fs0,fs1,fs2)
        call generalSolutionMinus1_Coeff_p(t,fp0,fp1,fp2)
        call generalSolutionMinus1_Coeff_r(t,fr0,fr1,fr2)
        call generalSolutionMinus1_Coeff_t(t,ft0,ft1,ft2)
!
        st = (dot_product(d,fs2) * lnr2 + dot_product(d,fs1) * lnr1 + dot_product(d,fs0)) / r**2
        pr = (dot_product(d,fp2) * lnr2 + dot_product(d,fp1) * lnr1 + dot_product(d,fp0)) / r**2
        vr = (dot_product(d,fr2) * lnr2 + dot_product(d,fr1) * lnr1 + dot_product(d,fr0)) / r
        vt = (dot_product(d,ft2) * lnr2 + dot_product(d,ft1) * lnr1 + dot_product(d,ft0)) / r
    end if
    end subroutine generalSolutionMinus1_Polar
!================================================================================
    subroutine generalSolutionMinus1_Boundary(d)
    implicit none
    real(DP), dimension(11), intent(in) :: d
    real(DP), dimension(11)             :: fs0, fs1, fs2
    real(DP), dimension(11)             :: fp0, fp1, fp2
    real(DP), dimension(11)             :: fr0, fr1, fr2
    real(DP), dimension(11)             :: ft0, ft1, ft2
    real(DP), dimension(11)             :: gs0, gs1, gs2
    real(DP), dimension(11)             :: gp0, gp1, gp2
    real(DP), dimension(11)             :: gr0, gr1, gr2
    real(DP), dimension(11)             :: gt0, gt1, gt2
    real(DP), dimension(11)             :: hs0, hs1, hs2
    real(DP), dimension(11)             :: hp0, hp1, hp2
    real(DP), dimension(11)             :: hr0, hr1, hr2
    real(DP), dimension(11)             :: ht0, ht1, ht2
    real(DP)                            :: t1, t2, t3
```

```
    integer                             :: a1, a2, a3
    t1 = 0.0_DP
    t2 = sav_theta / 2.0_DP
    t3 = sav_theta
    a1 = nint(t1 * 180.0_DP / PI)
    a2 = nint(t2 * 180.0_DP / PI)
    a3 = nint(t3 * 180.0_DP / PI)
!
    call generalSolutionMinus1_Coeff_s(t1,fs0,fs1,fs2)
    call generalSolutionMinus1_Coeff_p(t1,fp0,fp1,fp2)
    call generalSolutionMinus1_Coeff_r(t1,fr0,fr1,fr2)
    call generalSolutionMinus1_Coeff_t(t1,ft0,ft1,ft2)
!
    call generalSolutionMinus1_Coeff_s(t2,gs0,gs1,gs2)
    call generalSolutionMinus1_Coeff_p(t2,gp0,gp1,gp2)
    call generalSolutionMinus1_Coeff_r(t2,gr0,gr1,gr2)
    call generalSolutionMinus1_Coeff_t(t2,gt0,gt1,gt2)
!
    call generalSolutionMinus1_Coeff_s(t3,hs0,hs1,hs2)
    call generalSolutionMinus1_Coeff_p(t3,hp0,hp1,hp2)
    call generalSolutionMinus1_Coeff_r(t3,hr0,hr1,hr2)
    call generalSolutionMinus1_Coeff_t(t3,ht0,ht1,ht2)
!
    print 300
    print 100, 'st', a1, dot_product(d,fs2), dot_product(d,fs1), dot_product(d,fs0), '-2'
    print 100, 'pr', a1, dot_product(d,fp2), dot_product(d,fp1), dot_product(d,fp0), '-2'
    print 100, 'vr', a1, dot_product(d,fr2), dot_product(d,fr1), dot_product(d,fr0), '-1'
    print 100, 'vt', a1, dot_product(d,ft2), dot_product(d,ft1), dot_product(d,ft0), '-1'
    print 200
    print 100, 'st', a3, dot_product(d,hs2), dot_product(d,hs1), dot_product(d,hs0), '-2'
    print 100, 'pr', a3, dot_product(d,hp2), dot_product(d,hp1), dot_product(d,hp0), '-2'
    print 100, 'vr', a3, dot_product(d,hr2), dot_product(d,hr1), dot_product(d,hr0), '-1'
    print 100, 'vt', a3, dot_product(d,ht2), dot_product(d,ht1), dot_product(d,ht0), '-1'
    print 300
!
100 format (a, '(r,', i3, ')=[(', f16.12, ')lnr2+(', f16.12, ')ln+(', f16.12, ')]r^', a)
200 format ()
300 format (79('-'))
!
    end subroutine generalSolutionMinus1_Boundary
!==============================================================================
    subroutine generalSolutionMinus1_Check
    implicit none
    sav_d(1: 4) = (/  1.37_DP,  5.76_DP, -6.32_DP, -7.98_DP /)
    sav_d(5: 8) = (/  4.84_DP, -4.21_DP,  8.91_DP, -9.62_DP /)
    sav_d(9:11) = (/ -7.51_DP,  3.54_DP,  4.03_DP           /)
    call stokesEq2d_Cartesian(generalSolutionMinus1_FieldCartesian,'generalSolutionMinus1_Cartesian.txt')
    call stokesEq2d_Polar(generalSolutionMinus1_FieldPolar,'generalSolutionMinus1_Polar.txt')
    end subroutine generalSolutionMinus1_Check
!==============================================================================
    subroutine generalSolutionMinus1_FieldCartesian(p,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    call generalSolutionMinus1_Cartesian(sav_d,p,f)
    end subroutine generalSolutionMinus1_FieldCartesian
!==============================================================================
    subroutine generalSolutionMinus1_FieldPolar(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
```

```
    real(DP), intent(out) :: pr, vr, vt, st
    call generalSolutionMinus1_Polar(sav_d,r,t,pr,vr,vt,st)
    end subroutine generalSolutionMinus1_FieldPolar
!=======================================================================
end module mod_generalSolutionMinus1
!#######################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_generalSolution
use mod_constants
use mod_utilities
use mod_stokesEq2d
implicit none
private
public :: generalSolution_SetAngle
public :: generalSolution_Coeff_p
public :: generalSolution_Coeff_r
public :: generalSolution_Coeff_t
public :: generalSolution_Coeff_s
public :: generalSolution_Cartesian
public :: generalSolution_Polar
public :: generalSolution_Check
public :: generalSolution_Boundary
real(DP)                :: sav_theta = PI / 2.0_DP
real(DP)                :: sav_w
real(DP), dimension(12) :: sav_d
contains
!=======================================================================
    function kap(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    real(DP)               :: mu, nu
    mu = w + 1.0_DP
    nu = w - 1.0_DP
    f(1) = mu**i * cos(mu * t)
    f(2) = mu**i * sin(mu * t)
    f(3) = nu**i * cos(nu * t)
    f(4) = nu**i * sin(nu * t)
    end function kap
!=======================================================================
    function lam(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    real(DP)               :: mu, nu
    mu = w + 1.0_DP
    nu = w - 1.0_DP
    f(1) = -mu**i * sin(mu * t)
    f(2) =  mu**i * cos(mu * t)
    f(3) = -nu**i * sin(nu * t)
    f(4) =  nu**i * cos(nu * t)
    end function lam
!=======================================================================
    function alp(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    select case(i)
        case(1)
```

```
            f = (w + 1.0_DP)**2 * lam(1,t,w) - lam(3,t,w)
        case(2)
            f = 2 * (w + 1.0_DP)**2 * (lam(0,t,w) - t * kap(1,t,w)) &
                - 6.0_DP * lam(2,t,w) + 2.0_DP * t * kap(3,t,w)
        case(3)
            f = 6.0_DP * (t * kap(2,t,w) - lam(1,t,w)) + t**2 * lam(3,t,w) &
                - (w + 1.0_DP)**2 * t * (2.0_DP * kap(0,t,w) + t * lam(1,t,w))
        case(4)
            f = t * kap(3,t,w) - 3.0_DP * lam(2,t,w) &
                + (w + 1.0_DP)**2 * (lam(0,t,w) - t * kap(1,t,w))
        case default
            f = 0.0d0
    end select
    end function alp
!==============================================================================
    subroutine generalSolution_SetAngle(angle)
    implicit none
    integer, intent(in) :: angle
    sav_theta = real(angle,DP) * PI / 180.0_DP
    end subroutine generalSolution_SetAngle
!==============================================================================
    subroutine generalSolution_Coeff_p(t,w,fp0,fp1,fp2)
    implicit none
    real(DP),                intent(in)  :: t, w
    real(DP), dimension(12), intent(out) :: fp0, fp1, fp2
    real(DP), dimension(4)               :: fpa0, fpa1, fpa2, fpb0, fpb1, fpc0
    fpa2 = -alp(1,t,w) / (w - 1.0_DP)
    fpb1 = fpa2
    fpc0 = fpa2
    fpa1 = -alp(2,t,w) / (w - 1.0_DP) + 2.0_DP / (w - 1.0_DP)**2 * alp(1,t,w) &
           - 4.0_DP * (w + 1.0_DP) / (w - 1.0_DP) * lam(1,t,w)
    fpa0 = -alp(3,t,w) / (w - 1.0_DP) + alp(2,t,w) / (w - 1.0_DP)**2 &
           - 2.0_DP / (w - 1.0_DP)**3 * alp(1,t,w) &
           - 4.0_DP * (w + 1.0_DP) / (w - 1.0_DP) * (lam(0,t,w) - t * kap(1,t,w)) &
           + 2.0_DP * (3.0_DP + w) / (w - 1.0_DP)**2 * lam(1,t,w)
    fpb0 = -alp(4,t,w) / (w - 1.0_DP) + alp(1,t,w) / (w - 1.0_DP)**2 &
           - 2.0_DP * (w + 1.0_DP) / (w - 1.0_DP) * lam(1,t,w)
    fp0 = 0.0_DP
    fp1 = 0.0_DP
    fp2 = 0.0_DP
    fp2(1: 4) = fpa2
    fp1(1: 4) = fpa1
    fp1(5: 8) = fpb1
    fp0(1: 4) = fpa0
    fp0(5: 8) = fpb0
    fp0(9:12) = fpc0
    end subroutine generalSolution_Coeff_p
!==============================================================================
    subroutine generalSolution_Coeff_r(t,w,fr0,fr1,fr2)
    implicit none
    real(DP),                intent(in)  :: t, w
    real(DP), dimension(12), intent(out) :: fr0, fr1, fr2
    real(DP), dimension(4)               :: fra0, fra1, fra2, frb0, frb1, frc0
    fra2 = -lam(1,t,w)
    frb1 = fra2
    frc0 = fra2
    fra1 = 2.0_DP * t * kap(1,t,w) - 2.0_DP * lam(0,t,w)
    fra0 = 2.0_DP * t * kap(0,t,w) + t**2 * lam(1,t,w)
    frb0 = t * kap(1,t,w) - lam(0,t,w)
    fr0 = 0.0_DP
```

```
    fr1 = 0.0_DP
    fr2 = 0.0_DP
    fr2(1: 4) = fra2
    fr1(1: 4) = fra1
    fr1(5: 8) = frb1
    fr0(1: 4) = fra0
    fr0(5: 8) = frb0
    fr0(9:12) = frc0
    end subroutine generalSolution_Coeff_r
!================================================================================
    subroutine generalSolution_Coeff_t(t,w,ft0,ft1,ft2)
    implicit none
    real(DP),                intent(in)  :: t, w
    real(DP), dimension(12), intent(out) :: ft0, ft1, ft2
    real(DP), dimension(4)               :: fta0, fta1, fta2, ftb0, ftb1, ftc0
    fta2 = (w + 1.0_DP) * kap(0,t,w)
    ftb1 = fta2
    ftc0 = fta2
    fta1 = 2.0_DP * (kap(0,t,w) + (w + 1.0_DP) * t * lam(0,t,w))
    fta0 = 2.0_DP * t * lam(0,t,w) - (w + 1.0_DP) * t**2 * kap(0,t,w)
    ftb0 = kap(0,t,w) + (w + 1.0_DP) * t * lam(0,t,w)
    ft0 = 0.0_DP
    ft1 = 0.0_DP
    ft2 = 0.0_DP
    ft2(1: 4) = fta2
    ft1(1: 4) = fta1
    ft1(5: 8) = ftb1
    ft0(1: 4) = fta0
    ft0(5: 8) = ftb0
    ft0(9:12) = ftc0
    end subroutine generalSolution_Coeff_t
!================================================================================
    subroutine generalSolution_Coeff_s(t,w,fs0,fs1,fs2)
    implicit none
    real(DP),                intent(in)  :: t, w
    real(DP), dimension(12), intent(out) :: fs0, fs1, fs2
    real(DP), dimension(4)               :: fsa0, fsa1, fsa2, fsb0, fsb1, fsc0
    fsa2 = kap(2,t,w) - (1.0_DP - w**2) * kap(0,t,w)
    fsb1 = fsa2
    fsc0 = fsa2
    fsa1 = 4.0_DP * (w * kap(0,t,w) + kap(1,t,w)) + 2.0_DP * t * lam(2,t,w) &
           - 2.0_DP * (1.0_DP - w**2) * t * lam(0,t,w)
    fsa0 = (4.0_DP + (1.0_DP - w**2) * t**2) * kap(0,t,w) &
           + 4.0_DP * t * (w * lam(0,t,w) + lam(1,t,w)) - t**2 * kap(2,t,w)
    fsb0 = 2.0_DP * (kap(1,t,w) + w * kap(0,t,w)) + t * lam(2,t,w) &
           - (1.0_DP - w**2) * t * lam(0,t,w)
    fs0 = 0.0_DP
    fs1 = 0.0_DP
    fs2 = 0.0_DP
    fs2(1: 4) = fsa2
    fs1(1: 4) = fsa1
    fs1(5: 8) = fsb1
    fs0(1: 4) = fsa0
    fs0(5: 8) = fsb0
    fs0(9:12) = fsc0
    end subroutine generalSolution_Coeff_s
!================================================================================
    subroutine generalSolution_Cartesian(w,d,p,f)
    real(DP),                intent(in)  :: w
    real(DP), dimension(12),  intent(in)  :: d
```

```
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(p,p))
    t = angle(p)
    call generalSolution_Polar(w,d,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine generalSolution_Cartesian
!==============================================================================
    subroutine generalSolution_Polar(w,d,r,t,pr,vr,vt,st)
    implicit none
    real(DP),                 parameter   :: TOL = 1.0e-8_DP
    real(DP), dimension(12), intent(in)  :: d
    real(DP),                 intent(in)  :: r, t, w
    real(DP),                 intent(out) :: pr, vr, vt, st
    real(DP), dimension(12)               :: fs0, fs1, fs2
    real(DP), dimension(12)               :: fp0, fp1, fp2
    real(DP), dimension(12)               :: fr0, fr1, fr2
    real(DP), dimension(12)               :: ft0, ft1, ft2
    real(DP)                              :: lnr1, lnr2
    if (r < TOL) then
        st = 0.0_DP
        pr = 0.0_DP
        vr = 0.0_DP
        vt = 0.0_DP
    else
        lnr2 = log(r)**2
        lnr1 = log(r)
!
        call generalSolution_Coeff_s(t,w,fs0,fs1,fs2)
        call generalSolution_Coeff_p(t,w,fp0,fp1,fp2)
        call generalSolution_Coeff_r(t,w,fr0,fr1,fr2)
        call generalSolution_Coeff_t(t,w,ft0,ft1,ft2)
!
        st = (dot_product(d,fs2) * lnr2 + dot_product(d,fs1) * lnr1 + dot_product(d,fs0)) * r**(w - 1.0_DP)
        pr = (dot_product(d,fp2) * lnr2 + dot_product(d,fp1) * lnr1 + dot_product(d,fp0)) * r**(w - 1.0_DP)
        vr = (dot_product(d,fr2) * lnr2 + dot_product(d,fr1) * lnr1 + dot_product(d,fr0)) * r**w
        vt = (dot_product(d,ft2) * lnr2 + dot_product(d,ft1) * lnr1 + dot_product(d,ft0)) * r**w
    end if
    end subroutine generalSolution_Polar
!==============================================================================
    subroutine generalSolution_Boundary(w,d)
    implicit none
    real(DP),                 intent(in) :: w
    real(DP), dimension(12), intent(in) :: d
    real(DP), dimension(12)             :: fs0, fs1, fs2
    real(DP), dimension(12)             :: fp0, fp1, fp2
    real(DP), dimension(12)             :: fr0, fr1, fr2
    real(DP), dimension(12)             :: ft0, ft1, ft2
    real(DP), dimension(12)             :: gs0, gs1, gs2
    real(DP), dimension(12)             :: gp0, gp1, gp2
    real(DP), dimension(12)             :: gr0, gr1, gr2
    real(DP), dimension(12)             :: gt0, gt1, gt2
    real(DP), dimension(12)             :: hs0, hs1, hs2
    real(DP), dimension(12)             :: hp0, hp1, hp2
    real(DP), dimension(12)             :: hr0, hr1, hr2
    real(DP), dimension(12)             :: ht0, ht1, ht2
    real(DP)                            :: t1, t2, t3
    integer                             :: a1, a2, a3
```

```
    t1 = 0.0_DP
    t2 = sav_theta / 2.0_DP
    t3 = sav_theta
    a1 = nint(t1 * 180.0_DP / PI)
    a2 = nint(t2 * 180.0_DP / PI)
    a3 = nint(t3 * 180.0_DP / PI)
!
    call generalSolution_Coeff_s(t1,w,fs0,fs1,fs2)
    call generalSolution_Coeff_p(t1,w,fp0,fp1,fp2)
    call generalSolution_Coeff_r(t1,w,fr0,fr1,fr2)
    call generalSolution_Coeff_t(t1,w,ft0,ft1,ft2)
!
    call generalSolution_Coeff_s(t2,w,gs0,gs1,gs2)
    call generalSolution_Coeff_p(t2,w,gp0,gp1,gp2)
    call generalSolution_Coeff_r(t2,w,gr0,gr1,gr2)
    call generalSolution_Coeff_t(t2,w,gt0,gt1,gt2)
!
    call generalSolution_Coeff_s(t3,w,hs0,hs1,hs2)
    call generalSolution_Coeff_p(t3,w,hp0,hp1,hp2)
    call generalSolution_Coeff_r(t3,w,hr0,hr1,hr2)
    call generalSolution_Coeff_t(t3,w,ht0,ht1,ht2)
!
    print 300
    print 100, 'st', a1, dot_product(d,fs2), dot_product(d,fs1), dot_product(d,fs0), w - 1.0_DP
    print 100, 'pr', a1, dot_product(d,fp2), dot_product(d,fp1), dot_product(d,fp0), w - 1.0_DP
    print 100, 'vr', a1, dot_product(d,fr2), dot_product(d,fr1), dot_product(d,fr0), w
    print 100, 'vt', a1, dot_product(d,ft2), dot_product(d,ft1), dot_product(d,ft0), w
    print 200
    print 100, 'st', a3, dot_product(d,hs2), dot_product(d,hs1), dot_product(d,hs0), w - 1.0_DP
    print 100, 'pr', a3, dot_product(d,hp2), dot_product(d,hp1), dot_product(d,hp0), w - 1.0_DP
    print 100, 'vr', a3, dot_product(d,hr2), dot_product(d,hr1), dot_product(d,hr0), w
    print 100, 'vt', a3, dot_product(d,ht2), dot_product(d,ht1), dot_product(d,ht0), w
    print 300
!
100 format (a, '(r,', i3, ') = [(', f10.6, ') ln(r)^2 + (', f10.6, ') ln(r) + (', f10.6, ')] r^(', f4.1, ')')
200 format ()
300 format (79('-'))
!
    end subroutine generalSolution_Boundary
!================================================================================
    subroutine generalSolution_Check
    implicit none
    sav_w = 2.37_DP
    sav_d(1: 4) = (/  1.37_DP,  5.76_DP, -6.32_DP, -7.98_DP /)
    sav_d(5: 8) = (/  4.84_DP, -4.21_DP,  8.91_DP, -9.62_DP /)
    sav_d(9:12) = (/ -7.51_DP,  3.54_DP,  4.03_DP,  3.72_DP /)
    call stokesEq2d_Cartesian(generalSolution_FieldCartesian,'generalSolution_Cartesian.txt')
    call stokesEq2d_Polar(generalSolution_FieldPolar,'generalSolution_Polar.txt')
    end subroutine generalSolution_Check
!================================================================================
    subroutine generalSolution_FieldCartesian(p,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    call generalSolution_Cartesian(sav_w,sav_d,p,f)
    end subroutine generalSolution_FieldCartesian
!================================================================================
    subroutine generalSolution_FieldPolar(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
```

```
    real(DP), intent(out) :: pr, vr, vt, st
    call generalSolution_Polar(sav_w,sav_d,r,t,pr,vr,vt,st)
    end subroutine generalSolution_FieldPolar
!=============================================================================
end module mod_generalSolution
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_genSol
use mod_constants
use mod_utilities
use mod_generalSolutionPlus1
use mod_generalSolutionMinus1
use mod_generalSolution
implicit none
private
public :: genSol_Cartesian
public :: genSol_Polar
public :: genSol_SetAngle
contains
!=============================================================================
    subroutine genSol_Cartesian(w,d,p,f)
    implicit none
    real(DP),                 parameter   :: TOL = 1.0e-6_DP
    real(DP),                 intent(in)  :: w
    real(DP), dimension(12),  intent(in)  :: d
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(10)               :: dPlus1
    real(DP), dimension(11)               :: dMinus1
    if (abs(w - 1.0_DP) < TOL) then
        dPlus1(1: 6) = d(1: 6)
        dPlus1(7:10) = d(9:12)
        call generalSolutionPlus1_Cartesian(dPlus1,p,f)
    else if(abs(w + 1.0_DP) < TOL) then
        dMinus1(1:10) = d(1:10)
        dMinus1(  11) = d(  12)
        call generalSolutionMinus1_Cartesian(dMinus1,p,f)
    else
        call generalSolution_Cartesian(w,d,p,f)
    end if
    end subroutine genSol_Cartesian
!=============================================================================
    subroutine gensol_Polar(w,d,r,t,pr,vr,vt,st)
    implicit none
    real(DP),                 parameter   :: TOL = 1.0e-6_DP
    real(DP),                 intent(in)  :: r, t, w
    real(DP), dimension(12),  intent(in)  :: d
    real(DP),                 intent(out) :: pr, vr, vt, st
    real(DP), dimension(10)               :: dPlus1
    real(DP), dimension(11)               :: dMinus1
    if (abs(w - 1.0_DP) < TOL) then
        dPlus1(1: 6) = d(1: 6)
        dPlus1(7:10) = d(9:12)
        call generalSolutionPlus1_Polar(dPlus1,r,t,pr,vr,vt,st)
    else if(abs(w + 1.0_DP) < TOL) then
        dMinus1 = d(1:11)
        call generalSolutionMinus1_Polar(dMinus1,r,t,pr,vr,vt,st)
    else
        call generalSolution_Polar(w,d,r,t,pr,vr,vt,st)
```

```
    end if
    end subroutine gensol_Polar
!=============================================================================
    subroutine genSol_SetAngle(angle)
    implicit none
    integer, intent(in) :: angle
    call generalSolutionPlus1_SetAngle(angle)
    call generalSolutionMinus1_SetAngle(angle)
    call generalSolution_SetAngle(angle)
    call generalSolutionPlus1_Check
    call generalSolutionMinus1_Check
    call generalSolution_Check
    end subroutine genSol_SetAngle
!=============================================================================
end module mod_genSol
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outer0
use mod_constants
use mod_utilities
use mod_stokesEq2d
implicit none
private
type, public :: typ_outer0
    private
    real(DP) :: theta = PI / 2.0_DP
contains
    procedure :: SetAngle  => outer0_SetAngle
    procedure :: Cartesian => outer0_Cartesian
    procedure :: Polar     => outer0_Polar
    procedure :: Boundary  => outer0_Boundary
    procedure :: Check     => outer0_Check
end type typ_outer0
contains
!=============================================================================
    subroutine outer0_SetAngle(this,arg_angle)
    implicit none
    class(typ_outer0), intent(inout) :: this
    integer,           intent(in)    :: arg_angle
    this%theta = real(arg_angle,DP) * PI / 180.0_DP
    end subroutine outer0_SetAngle
!=============================================================================
    subroutine outer0_Cartesian(this,arg_p,arg_f)
    implicit none
    real(DP),                 parameter   :: TOL = 1.0d-9
    class(typ_outer0),        intent(in)  :: this
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP)                              :: r, t, x, y
    real(DP)                              :: a, b, c, d, e
    real(DP)                              :: pr, vx, vy, st
    real(DP)                              :: th, sn, cs
    r = sqrt(dot_product(arg_p,arg_p))
    if (r < TOL) then
        arg_f = 0.0d0
    else
        th = this%theta
        sn = sin(th)
        cs = cos(th)
```

```
        a = -1.0_DP
        b = -(th + cs * sn) / (th**2 - sn**2)
        c =  (th + cs * sn) / (th**2 - sn**2)
        d =  sn**2          / (th**2 - sn**2)
        e = th**2 + cs**2 - 1.0_DP
        x = arg_p(1)
        y = arg_p(2)
        t = angle(arg_p)
        vx = -b - d * t - (d * x * y + c * x**2) / r**2
        vy =  a + c * t - (c * x * y + d * y**2) / r**2
        pr = 1.0_DP - 2.0_DP * (c * x + d * y) / r**2
        st = 2.0_DP * (x * (cs**2 - 1.0_DP) + y * (th + cs * sn)) / (e * r**2)
        arg_f = (/ pr, vx, vy, st /)
    end if
    end subroutine outer0_Cartesian
!===============================================================================
    subroutine outer0_Polar(this,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    class(typ_outer0), intent(in)  :: this
    real(DP),          intent(in)  :: arg_r, arg_t
    real(DP),          intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP)                       :: r, s, c, t, sn, cs, th
    real(DP)                       :: alpha, beta, gamma
    th = this%theta
    sn = sin(th)
    cs = cos(th)
    t = arg_t
    s = sin(t)
    c = cos(t)
    r = arg_r
    alpha = cs**2 - 1.0_DP
    beta  = th + cs * sn
    gamma = th**2 + cs**2 - 1.0_DP
    arg_pr = 2.0_DP * (alpha * s - beta * c) / (gamma * r) + 1.0_DP
    arg_st = 2.0_DP * (alpha * c + beta * s) / (gamma * r)
    arg_vr = (beta * (t * s    ) + alpha * (t * c    ) - s * th**2) / gamma
    arg_vt = (beta * (t * c - s) - alpha * (t * s + c) - c * th**2) / gamma
    end subroutine outer0_Polar
!===============================================================================
    subroutine outer0_Boundary(this,arg_pr,arg_vr,arg_vt,arg_st)
    class(typ_outer0), intent(in)  :: this
    real(DP),          intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    real(DP)                       :: alpha, beta, gamma, th, cs, sn
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    arg_st = 0.0_DP
    th = this%theta
    sn = sin(th)
    cs = cos(th)
    alpha = cs**2 - 1.0_DP
    beta  = th + cs * sn
    gamma = th**2 + cs**2 - 1.0_DP
    arg_st =  2.0_DP * alpha / gamma
    arg_pr = -2.0_DP * beta  / gamma
    arg_vr =  0.0_DP
    arg_vt = -1.0_DP
!
100 format (79('-'))
200 format (a, '(r,', i1, ') = (', f17.10, ') r^(', f5.1, ')')
```

```
250 format (a, '(r,', i1, ') = (', f17.10, ') r^(', f5.1, ') + 1')
    end subroutine outer0_Boundary
!===============================================================================
    subroutine outer0_Check(this)
    implicit none
    class(typ_outer0), intent(in) :: this
    call outer0_CHECK_CARTESIAN(this)
    call outer0_CHECK_POLAR(this)
    end subroutine outer0_Check
!===============================================================================
    subroutine outer0_CHECK_CARTESIAN(this)
    implicit none
    class(typ_outer0), intent(in) :: this
    real(DP)                      :: x, y, vx, vy
    real(DP)                      :: r, t, pr, vr, vt, st
    real(DP), dimension(2)        :: p
    real(DP), dimension(0:3)      :: f
10      print*, 'x, y ?'
        read *,  x, y
        if (x < 0.0_DP) then
            return
        end if
        p = (/ x, y /)
        call outer0_Cartesian(this,p,f)
        print 100, 'c ', f
        print*
        r = sqrt(x**2 + y**2)
        t = angle(p)
        pr = f(0)
        vx = f(1)
        vy = f(2)
        call CartesianToPolar(t,vx,vy,vr,vt)
        print 100, 'p ', pr, vr, vt
        call outer0_Polar(this,r,t,pr,vr,vt,st)
        print 100, 'p ', pr, vr, vt
        goto 10
100 format(a, 3(1x, f12.7))
    end subroutine outer0_CHECK_CARTESIAN
!===============================================================================
    subroutine outer0_CHECK_POLAR(this)
    implicit none
    class(typ_outer0), intent(in) :: this
    integer                       :: angle
    real(DP)                      :: r, t
    real(DP)                      :: pr, vr, vt, st
    real(DP)                      :: x, y, vx, vy
    real(DP), dimension(2)        :: p
    real(DP), dimension(0:3)      :: f
10      print*, 'r, angle ?'
        read *,  r, angle
        if (r < 0.0_DP) then
            return
        end if
        t = real(angle,DP) * PI / 180.0_DP
        call outer0_Polar(this,r,t,pr,vr,vt,st)
        print 100, 'p ', pr, vr, vt
        print*
        call PolarToCartesian(t,vr,vt,vx,vy)
        print 100, 'c ', pr, vx, vy
        p(1) = r * cos(t)
```

```
        p(2) = r * sin(t)
        call outer0_Cartesian(this,p,f)
        print 100, 'c ', f(0:2)
        goto 10
100 format(a, 3(1x, f12.7))
    end subroutine outer0_CHECK_POLAR
!=======================================================================
end module mod_outer0
!#######################################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outer1
use mod_constants
use mod_utilities
use mod_generalSolutionMinus1
implicit none
private
type, public :: typ_outer1
    private
    integer :: k = 1
    integer :: angle = 90
contains
    procedure :: SetAngle    => outer1_SetAngle
    procedure :: SetSolution => outer1_SetSolution
    procedure :: Boundary    => outer1_Boundary
    procedure :: Cartesian   => outer1_Cartesian
    procedure :: Polar       => outer1_Polar
end type typ_outer1
contains
!=======================================================================
    subroutine outer1_SetAngle(this,arg_angle)
    implicit none
    class(typ_outer1), intent(inout) :: this
    integer,           intent(in)    :: arg_angle
    this%angle = clip(arg_angle,45,180)
    end subroutine outer1_SetAngle
!=======================================================================
    subroutine outer1_SetSolution(this,arg_k)
    implicit none
    class(typ_outer1), intent(inout) :: this
    integer,           intent(in)    :: arg_k
    this%k = clip(arg_k,1,3)
    end subroutine outer1_SetSolution
!=======================================================================
    subroutine outer1_Boundary(this,arg_pr,arg_vr,arg_vt,arg_st)
    class(typ_outer1),        intent(in)  :: this
    real(DP), dimension(0:1), intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    arg_st = 0.0_DP
    if (this%k == 1) then
        if (this%angle == 135) then
            arg_st(1) =  8.0_DP / (4.0_DP + 3.0_DP * PI)
            arg_pr(1) = -8.0_DP / (4.0_DP + 3.0_DP * PI)
            arg_pr(0) = -8.0_DP / (4.0_DP + 3.0_DP * PI)
            arg_vt(0) =  1.0_DP
        else if (this%angle == 90) then
            arg_st(1) = 2.0_DP
            arg_pr(0) = PI / 2.0_DP
```

```
            arg_vt(0) = 1.0_DP
        else if (this%angle == 45) then
            arg_st(1) =   9.319584732650_DP
            arg_st(0) = -15.167889203639_DP
            arg_pr(1) =   9.319584732649_DP
            arg_pr(0) =  -5.848304470989_DP
            arg_vt(0) =   1.0_DP
            arg_st(1) =   8.0_DP / (4.0_DP - PI)
            arg_st(0) = -(88.0_DP + PI) / (4.0_DP - PI) / 7.0_DP
            arg_pr(1) =   8.0_DP / (4.0_DP - PI)
            arg_pr(0) = -(32.0_DP + PI) / (4.0_DP - PI) / 7.0_DP
            arg_vt(0) =   1.0_DP
        end if
    else if (this%k == 2) then
        if (this%angle == 135) then
            arg_st(0) = 1.0_DP
            arg_pr(0) = 1.0_DP
            arg_vr(0) = 1.0_DP
        else if (this%angle == 90) then
            arg_pr(0) = 1.0_DP
            arg_vr(0) = 1.0_DP
        else if (this%angle == 45) then
            arg_pr(0) = 2.0_DP
            arg_vr(0) = 1.0_DP
        end if
    else
        if (this%angle == 135) then
            arg_st(0) =  1.0_DP
            arg_pr(0) = -1.0_DP
        else if (this%angle == 90) then
            arg_st(0) =  1.0_DP
        else if (this%angle == 45) then
            arg_st(0) =  1.0_DP
            arg_pr(0) =  1.0_DP
        end if
    end if
!
100 format(79('-'))
200 format(a, '(r,', i1, ') = [ (', f17.10, ') ln(r) + (' f17.10, ') ] r^(', f5.1, ')')
300 format(i3, 1x, f12.7, 3x, 4(4x, f12.7, 1x, f12.7))
    end subroutine outer1_Boundary
!===============================================================================
    subroutine outer1_Cartesian(this,arg_p,arg_f)
    implicit none
    class(typ_outer1),        intent(in)  :: this
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(11)               :: d
    call outer1_COEFFICIENTS(this,d)
    call generalSolutionMinus1_Cartesian(d,arg_p,arg_f)
    end subroutine outer1_Cartesian
!===============================================================================
    subroutine outer1_Polar(this,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    class(typ_outer1), intent(in)  :: this
    real(DP),          intent(in)  :: arg_r, arg_t
    real(DP),          intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(11)        :: d
    call outer1_COEFFICIENTS(this,d)
    call generalSolutionMinus1_Polar(d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
```

```
    end subroutine outer1_Polar
!===============================================================================
    subroutine outer1_COEFFICIENTS(this,arg_d)
    class(typ_outer1),       intent(in)  :: this
    real(DP), dimension(11), intent(out) :: arg_d
    arg_d = 0.0_DP
    if (this%k == 1) then
        if (this%angle == 135) then
            arg_d( 5) =  2.0_DP               / (4.0_DP + 3.0_DP * PI)
            arg_d( 6) =  2.0_DP               / (4.0_DP + 3.0_DP * PI)
            arg_d( 7) = (2.0_DP + 3.0_DP * PI) / (4.0_DP + 3.0_DP * PI)
            arg_d( 8) = -4.0_DP               / (4.0_DP + 3.0_DP * PI)
            arg_d( 9) = (8.0_DP + 3.0_DP * PI) / (4.0_DP + 3.0_DP * PI) / 2.0_DP
            arg_d(10) =  4.0_DP               / (4.0_DP + 3.0_DP * PI)
            arg_d(11) = -6.0_DP               / (4.0_DP + 3.0_DP * PI)
        else if (this%angle == 90) then
            arg_d( 5) =  1.0_DP / 2.0_DP
            arg_d( 7) =  1.0_DP / 2.0_DP
            arg_d( 9) =  1.0_DP
            arg_d(10) = -PI     / 8.0_DP
            arg_d(11) =  PI     / 4.0_DP
        else if (this%angle == 45) then
            arg_d( 5) =  2.3298961831624853680_DP
            arg_d( 6) = -2.3298961831623499208_DP
            arg_d( 7) = -1.3298961831625004670_DP
            arg_d( 8) =  4.6597923663248641546_DP
            arg_d( 9) = -0.96207611774735968257_DP
            arg_d(10) = -0.86782006541512501929_DP
            arg_d(11) = -0.59425605233209999323_DP
            arg_d( 5) =  2.0_DP                          / (4.0_DP - PI)
            arg_d( 6) = -2.0_DP                          / (4.0_DP - PI)
            arg_d( 7) =  (2.0_DP - PI)                   / (4.0_DP - PI)
            arg_d( 8) =  4.0_DP                          / (4.0_DP - PI)
            arg_d( 9) =  3.0_DP * (8.0_DP - 5.0_DP * PI) / (4.0_DP - PI) / 28.0_DP
            arg_d(10) =  (PI - 24.0_DP)                  / (4.0_DP - PI) / 28.0_DP
            arg_d(11) = -(4.0_DP + PI)                   / (4.0_DP - PI) / 14.0_DP
        end if
    else if (this%k == 2) then
        if (this%angle == 135) then
            arg_d( 9) =  1.0_DP / 4.0_DP
            arg_d(10) = -1.0_DP / 4.0_DP
            arg_d(11) = -1.0_DP / 2.0_DP
        else if (this%angle == 90) then
            arg_d(10) = -1.0_DP / 4.0_DP
            arg_d(11) = -1.0_DP / 2.0_DP
        else if (this%angle == 45) then
            arg_d(10) = -1.0_DP / 2.0_DP
        end if
    else
        if (this%angle == 135) then
            arg_d( 9) =  1.0_DP / 4.0_DP
            arg_d(10) =  1.0_DP / 4.0_DP
            arg_d(11) = -1.0_DP / 2.0_DP
        else if (this%angle == 90) then
            arg_d( 9) =  1.0_DP / 4.0_DP
        else if (this%angle == 45) then
            arg_d( 9) =  1.0_DP / 4.0_DP
            arg_d(10) = -1.0_DP / 4.0_DP
            arg_d(11) =  1.0_DP / 2.0_DP
        end if
```

```
    end if
    end subroutine outer1_COEFFICIENTS
!=======================================================================
end module mod_outer1
!#######################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outer1Check
use mod_constants
use mod_stokesEq2d
use mod_outer1
implicit none
private
public :: outer1Check
type(typ_outer1)       :: sav_outer1
real(DP), dimension(3) :: sav_c = (/ 3.0_DP, 2.0_DP, 4.0_DP /)
contains
!=======================================================================
    subroutine outer1Check
    implicit none
    integer angle
    print*, 'outer1Check'
    print*, 'angle ?'
    read *,  angle
    call sav_outer1%SetAngle(angle)
    call stokesEq2d_Cartesian(outer1Check_CARTESIAN,'outer1_Cartesian.txt')
    call stokesEq2d_Polar(outer1Check_POLAR,'outer1_Polar.txt')
    call outer1Check_BOUNDARY(angle)
    end subroutine outer1Check
!=======================================================================
    subroutine outer1Check_BOUNDARY(arg_angle)
    integer,  intent(in)     :: arg_angle
    integer                  :: i, iMax, k
    real(DP)                 :: r, rMax, rMin, w, t
    real(DP)                 :: st1, pr1, vr1, vt1
    real(DP)                 :: st2, pr2, vr2, vt2
    real(DP)                 :: st3, pr3, vr3, vt3
    real(DP)                 :: error, maxErrorMax
    real(DP), dimension(2)   :: errorMax
    real(DP), dimension(0:1) :: pr, vr, vt, st
    open(unit = 20, file = 'outer1_Boundary.txt')
    rMin = 0.5_DP
    rMax = 1.5_DP
    iMax = 10
    do k = 1, 3
        call sav_outer1%SetSolution(k)
        call sav_outer1%Boundary(pr,vr,vt,st)
        write(20,100) k
        write(20,200) 'st', st(1), st(0)
        write(20,200) 'pr', pr(1), pr(0)
        write(20,250) 'vr', vr(1), vr(0)
        write(20,250) 'vt', vt(1), vt(0)
        write(20,300), arg_angle, arg_angle
        errorMax(k) = 0.0_DP
        do i = 0, iMax
            r = rMin + (rMax - rMin) * real(i,DP) / real(iMax,DP)
            st1 = (st(1) * log(r) + st(0)) / r**2
            pr1 = (pr(1) * log(r) + pr(0)) / r**2
            vr1 = (vr(1) * log(r) + vr(0)) / r
            vt1 = (vt(1) * log(r) + vt(0)) / r
```

```
            t = 0.0_DP
            call sav_outer1%Polar(r,t,pr2,vr2,vt2,st2)
            t = real(arg_angle,DP) * PI / 180.0_DP
            call sav_outer1%Polar(r,t,pr3,vr3,vt3,st3)
            write(20,400) r, vr3, vt3, st1, st2, pr1, pr2, vr1, vr2, vt1, vt2
            error = max(abs(vr3),abs(vt3),abs(st1-st2),abs(pr1-pr2),abs(vr1-vr2),abs(vt1-vt2))
            if (error > errorMax(k)) then
                errorMax(k) = error
            end if
        end do
        write(20,500) errorMax(k)
    end do
    write(20,600)
    maxErrorMax = maxval(errorMax)
    write(20,700) maxErrorMax
    close(20)
100 format (157('-') / 'First Order Outer Solution: ', i1 /)
200 format (a, '(r,0) = [ (', f10.6, ') ln(r) + (', f10.6, ') ] / r^2')
250 format (a, '(r,0) = [ (', f10.6, ') ln(r) + (', f10.6, ') ] / r')
300 format (/ '           r', 4x, '   vr(r,' i3, ')', 1x, '   vt(r,', i3, ')', 4x, &
              ' stress(r,0)', 1x, '  stess(r,0)', 4x, '     pr(r,0)', 1x, &
              '     pr(r,0)', 4x, '     vr(r,0)', 1x, '     vr(r,0)', 4x, &
              '     vt(r,0)', 1x, '     vt(r,0)' /                        &
              12('-'), 5(4x, 12('-'), 1x, 12('-')))
400 format(f12.7, 5(4x, f12.7, 1x, f12.7))
500 format(/ 'Maximum Error: ', e12.5)
600 format(157('-'))
700 format('Overall Maximum Error: ', e12.5 / 157('-'))
    end subroutine outer1Check_BOUNDARY
!=============================================================================
    subroutine outer1Check_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(3,0:3)            :: f
    integer                               :: k
    do k = 1, 3
        call sav_outer1%SetSolution(k)
        call sav_outer1%Cartesian(arg_p,f(k,:))
    end do
    arg_f = matmul(sav_c,f)
    end subroutine outer1Check_CARTESIAN
!=============================================================================
    subroutine outer1Check_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)   :: arg_r, arg_t
    real(DP), intent(out)  :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(3) :: st, pr, vr, vt
    integer                :: k
    do k = 1, 3
        call sav_outer1%SetSolution(k)
        call sav_outer1%Polar(arg_r,arg_t,pr(k),vr(k),vt(k),st(k))
    end do
    arg_pr = dot_product(sav_c,pr)
    arg_vr = dot_product(sav_c,vr)
    arg_vt = dot_product(sav_c,vt)
    arg_st = dot_product(sav_c,st)
    end subroutine outer1Check_POLAR
!=============================================================================
end module mod_outer1Check
```

```
!################################################################################
! Copyright 2013 by Ludwig C. Nitsche
!---------------------------------
module mod_outerBuild
use mod_constants
use mod_outer0
use mod_genSol
implicit none
private
public :: outerBuild_Boundary
public :: outerBuild_Cartesian
public :: outerBuild_Polar
type(typ_outer0) :: sav_outer0
contains
!================================================================================
    subroutine outerBuild_Boundary(arg_k,arg_w,arg_pr,arg_vr,arg_vt,arg_st)
    integer,                  intent(in)  :: arg_k
    real(DP),                 intent(out) :: arg_w
    real(DP), dimension(0:2), intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    arg_st = 0.0_DP
    if (arg_k == 1) then
        call sav_outer0%Boundary(arg_pr(0),arg_vr(0),arg_vt(0),arg_st(0))
        arg_w = 0.0_DP
    else if (arg_k == 2) then
        arg_st(1) = 2.0_DP
        arg_pr(0) = PI / 2.0_DP
        arg_vt(0) = 1.0_DP
        arg_w = -1.0_DP
    else if (arg_k == 3) then
        arg_pr(0) = 1.0_DP
        arg_vr(0) = 1.0_DP
        arg_w = -1.0_DP
    else if (arg_k == 4) then
        arg_st(0) = 4.0_DP
        arg_w = -1.0_DP
    else if (arg_k == 5) then
        arg_st(0) = -6.0_DP
        arg_vt(0) =  1.0_DP
        arg_w = -2.0_DP
    else if (arg_k == 6) then
        arg_st(1) = -6.0_DP
        arg_st(0) =  2.0_DP
        arg_pr(0) = PI / 2.0_DP
        arg_vt(1) =  1.0_DP
        arg_w = -2.0_DP
    else if (arg_k == 7) then
        arg_pr(0) = 2.0_DP
        arg_vr(0) = 1.0_DP
        arg_w = -2.0_DP
    else if (arg_k == 8) then
        arg_st(0) = -PI / 2.0_DP
        arg_pr(1) =  2.0_DP
        arg_vr(1) =  1.0_DP
        arg_w = -2.0_DP
    else if (arg_k == 9) then
        arg_st(0) = -9.0_DP
        arg_vt(0) =  1.0_DP
```

```
        arg_w = -3.0_DP
    else if (arg_k == 10) then
        arg_st(1) = -9.0_DP
        arg_st(0) =  3.0_DP / 2.0_DP
        arg_pr(0) = -3.0_DP * PI / 8.0_DP
        arg_vt(1) =  1.0_DP
        arg_w = -3.0_DP
    else if (arg_k == 11) then
        arg_pr(0) = 3.0_DP / 2.0_DP
        arg_vr(0) = 1.0_DP
        arg_w = -3.0_DP
    else if (arg_k == 12) then
        arg_st(0) =  3.0_DP * PI / 8.0_DP
        arg_pr(1) =  3.0_DP / 2.0_DP
        arg_pr(0) = -1.0_DP / 8.0_DP
        arg_vr(1) =  1.0_DP
        arg_w = -3.0_DP
    else if (arg_k == 13) then
        arg_st(0) = -10.0_DP
        arg_vt(0) =   1.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 14) then
        arg_st(1) = -10.0_DP
        arg_st(0) =   2.0_DP
        arg_pr(0) = PI / 4.0_DP
        arg_vt(1) =   1.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 15) then
        arg_pr(0) = 2.0_DP
        arg_vr(0) = 1.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 16) then
        arg_st(0) = -PI / 4.0_DP
        arg_pr(1) =  2.0_DP
        arg_vr(1) =  1.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 17) then
        arg_st(0) = -25.0_DP / 2.0_DP
        arg_vt(0) =   1.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 18) then
        arg_st(1) = -25.0_DP / 2.0_DP
        arg_st(0) =  15.0_DP / 8.0_DP
        arg_pr(0) = -PI * 5.0_DP / 24.0_DP
        arg_vt(1) =   1.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 19) then
        arg_pr(0) = 5.0_DP / 3.0_DP
        arg_vr(0) = 1.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 20) then
        arg_st(0) = PI * 5.0_DP / 24.0_DP
        arg_pr(1) =      5.0_DP /  3.0_DP
        arg_pr(0) =     -1.0_DP / 18.0_DP
        arg_vr(1) = 1.0_DP
        arg_w = -5.0_DP
    else
        print*, 'Error in {outerBuild_Boundary}'
        stop
    end if
```

```
    end subroutine outerBuild_Boundary
!================================================================================
    subroutine outerBuild_Cartesian(arg_k,arg_p,arg_f)
    implicit none
    integer,                  intent(in)  :: arg_k
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(12)               :: d
    real(DP)                              :: w
    if (arg_k == 1) then
        call sav_outer0%Cartesian(arg_p,arg_f)
    else
        call outerBuild_COEFFICIENTS(arg_k,w,d)
        call gensol_Cartesian(w,d,arg_p,arg_f)
    end if
    end subroutine outerBuild_Cartesian
!================================================================================
    subroutine outerBuild_Polar(arg_k,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    integer,  intent(in)    :: arg_k
    real(DP), intent(in)    :: arg_r, arg_t
    real(DP), intent(out)   :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(12) :: d
    real(DP)                :: w
    if (arg_k == 1) then
        call sav_outer0%Polar(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    else
        call outerBuild_COEFFICIENTS(arg_k,w,d)
        call genSol_Polar(w,d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end if
    end subroutine outerBuild_Polar
!================================================================================
    subroutine outerBuild_COEFFICIENTS(arg_k,arg_w,arg_d)
    integer,                 intent(in)  :: arg_k
    real(DP),                intent(out) :: arg_w
    real(DP), dimension(12), intent(out) :: arg_d
    arg_d = 0.0_DP
    if (arg_k == 2) then
        arg_d( 5) = 0.5_DP
        arg_d( 7) = 0.5_DP
        arg_d( 9) = 1.0_DP
        arg_d(10) = -PI / 8.0_DP
        arg_d(11) =  PI / 4.0_DP
        arg_w = -1.0_DP
    else if (arg_k == 3) then
        arg_d(10) = -0.25_DP
        arg_d(11) = -0.5_DP
        arg_w = -1.0_DP
    else if (arg_k == 4) then
        arg_d(9) = 1.0_DP
        arg_w = -1.0_DP
    else if (arg_k == 5) then
        arg_d( 9) = -0.75_DP
        arg_d(11) = -0.25_DP
        arg_w = -2.0_DP
    else if (arg_k == 6) then
        arg_d( 5) = -0.75_DP
        arg_d( 7) = -0.25_DP
        arg_d( 9) = -0.875_DP
        arg_d(10) = -PI / 16.0_DP * 3.0_DP
```

```
        arg_d(11) = -0.125_DP
        arg_d(12) =  PI / 16.0_DP
        arg_w = -2.0_DP
    else if (arg_k == 7) then
        arg_d(10) = 0.25_DP
        arg_d(12) = 0.25_DP
        arg_w = -2.0_DP
    else if (arg_k == 8) then
        arg_d( 6) =  0.25_DP
        arg_d( 8) =  0.25_DP
        arg_d( 9) =  PI / 16.0_DP
        arg_d(10) =  0.125_DP
        arg_d(11) = -PI / 16.0_DP
        arg_d(12) =  0.125_DP
        arg_w = -2.0_DP
    else if (arg_k == 9) then
        arg_d( 9) = -0.25_DP
        arg_d(11) = -0.25_DP
        arg_w = -3.0_DP
    else if (arg_k == 10) then
        arg_d( 5) = -0.25_DP
        arg_d( 7) = -0.25_DP
        arg_d( 9) = -0.125_DP
        arg_d(10) =  PI / 16.0_DP
        arg_d(11) = -0.125_DP
        arg_d(12) = -PI / 32.0_DP
        arg_w = -3.0_DP
    else if (arg_k == 11) then
        arg_d(10) = 0.25_DP
        arg_d(12) = 0.125_DP
        arg_w = -3.0_DP
    else if (arg_k == 12) then
        arg_d( 6) =  1.0_DP /  4.0_DP
        arg_d( 8) =  1.0_DP /  8.0_DP
        arg_d( 9) = -PI     / 32.0_DP
        arg_d(10) =  1.0_DP /  8.0_DP
        arg_d(11) =  PI     / 32.0_DP
        arg_d(12) =  1.0_DP / 32.0_DP
        arg_w = -3.0_DP
    else if (arg_k == 13) then
        arg_d( 9) = -5.0_DP / 24.0_DP
        arg_d(11) = -1.0_DP /  8.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 14) then
        arg_d( 5) = -      5.0_DP /  24.0_DP
        arg_d( 7) = -      1.0_DP /   8.0_DP
        arg_d( 9) = -     23.0_DP / 288.0_DP
        arg_d(10) = -PI * 15.0_DP / 576.0_DP
        arg_d(11) = -      1.0_DP /  32.0_DP
        arg_d(12) =  PI           /  64.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 15) then
        arg_d(10) = 1.0_DP / 8.0_DP
        arg_d(12) = 1.0_DP / 8.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 16) then
        arg_d( 6) =  1.0_DP /  8.0_DP
        arg_d( 8) =  1.0_DP /  8.0_DP
        arg_d( 9) =  PI     / 64.0_DP
        arg_d(10) =  1.0_DP / 32.0_DP
```

```
        arg_d(11) = -PI     / 64.0_DP
        arg_d(12) =  1.0_DP / 32.0_DP
        arg_w = -4.0_DP
    else if (arg_k == 17) then
        arg_d( 9) = -1.0_DP / 8.0_DP
        arg_d(11) = -1.0_DP / 8.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 18) then
        arg_d( 5) = -1.0_DP /  8.0_DP
        arg_d( 7) = -1.0_DP /  8.0_DP
        arg_d( 9) = -1.0_DP / 32.0_DP
        arg_d(10) =  PI     / 64.0_DP
        arg_d(11) = -1.0_DP / 32.0_DP
        arg_d(12) = -PI     / 96.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 19) then
        arg_d(10) = 1.0_DP /  8.0_DP
        arg_d(12) = 1.0_DP / 12.0_DP
        arg_w = -5.0_DP
    else if (arg_k == 20) then
        arg_d( 6) =  1.0_DP /  8.0_DP
        arg_d( 8) =  1.0_DP / 12.0_DP
        arg_d( 9) = -PI     / 96.0_DP
        arg_d(10) =  1.0_DP / 32.0_DP
        arg_d(11) =  PI     / 96.0_DP
        arg_d(12) =  1.0_DP / 72.0_DP
        arg_w = -5.0_DP
    else
        print*, 'Error in {outerBuild_COEFFICIENTS}'
        print*, arg_k
        stop
    end if
    end subroutine outerBuild_COEFFICIENTS
!=======================================================================================
end module mod_outerBuild
!#######################################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outerBuildCheck
use mod_constants
use mod_stokesEq2d
use mod_outerBuild
implicit none
private
public :: outerBuildCheck
real(DP), dimension(20) :: sav_c
contains
!=======================================================================================
    subroutine outerBuildCheck
    call random_number(sav_c)
    print 100, sav_c
    call stokesEq2d_Cartesian(outerBuildCheck_CARTESIAN,'outerBuild_Cartesian.txt')
    call stokesEq2d_Polar(outerBuildCheck_POLAR,'outerBuild_Polar.txt')
    call outerBuildCheck_BOUNDARY
100 format(/ 'outerBuildCheck: Random Coefficients' / 20(f12.7 /) /)
    end subroutine outerBuildCheck
!=======================================================================================
    subroutine outerBuildCheck_BOUNDARY
    integer                 :: i, iMax, k
    real(DP)                :: r, rMax, rMin, w, t
```

```
    real(DP)                 :: st1, pr1, vr1, vt1
    real(DP)                 :: st2, pr2, vr2, vt2
    real(DP)                 :: st3, pr3, vr3, vt3
    real(DP)                 :: error, maxErrorMax
    real(DP), dimension(20)  :: errorMax
    real(DP), dimension(0:2) :: pr, vr, vt, st
    open(unit = 20, file = 'outerBuild_Boundary.txt')
    rMin = 0.5_DP
    rMax = 1.5_DP
    iMax = 10
    do k = 1, 20
        call outerBuild_Boundary(k,w,pr,vr,vt,st)
        write(20,100) k
        write(20,200) 'st', st(2), st(1), st(0), w - 1.0_DP
        if (k == 1) then
            write(20,250) 'pr', pr(2), pr(1), pr(0), w - 1.0_DP
        else
            write(20,200) 'pr', pr(2), pr(1), pr(0), w - 1.0_DP
        end if
        write(20,200) 'vr', vr(2), vr(1), vr(0), w
        write(20,200) 'vt', vt(2), vt(1), vt(0), w
        write(20,300)
        errorMax(k) = 0.0_DP
        do i = 0, iMax
            r = rMin + (rMax - rMin) * real(i,DP) / real(iMax,DP)
            st1 = (st(2) * log(r)**2 + st(1) * log(r) + st(0)) * r**(w - 1.0_DP)
            pr1 = (pr(2) * log(r)**2 + pr(1) * log(r) + pr(0)) * r**(w - 1.0_DP)
            vr1 = (vr(2) * log(r)**2 + vr(1) * log(r) + vr(0)) * r**w
            vt1 = (vt(2) * log(r)**2 + vt(1) * log(r) + vt(0)) * r**w
            if (k == 1) then
                pr1 = pr1 + 1.0_DP
            end if
            t = 0.0_DP
            call outerBuild_Polar(k,r,t,pr2,vr2,vt2,st2)
            t = PI / 2.0_DP
            call outerBuild_Polar(k,r,t,pr3,vr3,vt3,st3)
            write(20,400) r, vr3, vt3, st1, st2, pr1, pr2, vr1, vr2, vt1, vt2
            error = max(abs(vr3),abs(vt3),abs(st1-st2),abs(pr1-pr2),abs(vr1-vr2),abs(vt1-vt2))
            if (error > errorMax(k)) then
                errorMax(k) = error
            end if
        end do
        write(20,500) errorMax(k)
    end do
    write(20,600)
    maxErrorMax = maxval(errorMax)
    write(20,700) maxErrorMax
    close(20)
100 format (157('-') / 'Outer Solution Building Block: ', i2 /)
200 format (a, '(r,0) = [ (', f10.6, ') ln(r)^2 + (', f10.6, ') ln(r) + (', f10.6, ') ] r^(', f4.1, ')')
250 format (a, '(r,0) = [ (', f10.6, ') ln(r)^2 + (', f10.6, ') ln(r) + (', f10.6, ') ] r^(', f4.1, ') + 1')
300 format (/ '           r', 4x, '    vr(r,90)', 1x, '    vt(r,90)', 4x, &
             ' stress(r,0)', 1x, '  stess(r,0)', 4x, '     pr(r,0)', 1x, &
             '     pr(r,0)', 4x, '     vr(r,0)', 1x, '     vr(r,0)', 4x, &
             '     vt(r,0)', 1x, '     vt(r,0)' /                       &
            12('-'), 5(4x, 12('-'), 1x, 12('-')))
400 format(f12.7, 5(4x, f12.7, 1x, f12.7))
500 format(/ 'Maximum Error: ', e12.5)
600 format(157('-'))
700 format('Overall Maximum Error: ', e12.5 / 157('-'))
```

```
    end subroutine outerBuildCheck_BOUNDARY
!=============================================================================
    subroutine outerBuildCheck_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(20,0:3)           :: f
    integer                               :: k
    do k = 1, 20
        call outerBuild_Cartesian(k,arg_p,f(k,:))
    end do
    arg_f = matmul(sav_c,f)
    end subroutine outerBuildCheck_CARTESIAN
!=============================================================================
    subroutine outerBuildCheck_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)    :: arg_r, arg_t
    real(DP), intent(out)   :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(20) :: st, pr, vr, vt
    integer                 :: k
    do k = 1, 20
        call outerBuild_Polar(k,arg_r,arg_t,pr(k),vr(k),vt(k),st(k))
    end do
    arg_pr = dot_product(sav_c,pr)
    arg_vr = dot_product(sav_c,vr)
    arg_vt = dot_product(sav_c,vt)
    arg_st = dot_product(sav_c,st)
    end subroutine outerBuildCheck_POLAR
!=============================================================================
end module mod_outerBuildCheck
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_innerBuild
use mod_constants
use mod_genSol
implicit none
private
public :: innerBuild_Boundary
public :: innerBuild_Cartesian
public :: innerBuild_Polar
contains
!=============================================================================
    subroutine innerBuild_Boundary(arg_k,arg_w,arg_pr,arg_vr,arg_vt,arg_st)
    integer,                  intent(in)  :: arg_k
    real(DP),                 intent(out) :: arg_w
    real(DP), dimension(0:2), intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    arg_st = 0.0_DP
    if (arg_k == 1) then
        arg_vr(0) =  0.5_DP
        arg_vt(0) =  1.0_DP
        arg_w = 0.5_DP
    else if (arg_k == 2) then
        arg_pr(0) =  1.0_DP
        arg_vr(0) =  1.0_DP / 4.0_DP
        arg_vt(0) =  1.0_DP / 6.0_DP
        arg_w = 1.5_DP
```

```
    else if (arg_k == 3) then
        arg_st(0) =  1.0_DP
        arg_vr(0) = -1.0_DP / 6.0_DP
        arg_vt(0) =  2.0_DP / 3.0_DP
        arg_w = 1.5_DP
    else if (arg_k == 4) then
        arg_pr(0) =  1.0_DP
        arg_vr(0) =  3.0_DP /  8.0_DP
        arg_vt(0) = -1.0_DP / 20.0_DP
        arg_w = 2.5_DP
    else if (arg_k == 5) then
        arg_st(0) =  1.0_DP
        arg_vr(0) =  1.0_DP / 20.0_DP
        arg_vt(0) =  3.0_DP / 10.0_DP
        arg_w = 2.5_DP
    else if (arg_k == 6) then
        arg_pr(0) =  1.0_DP
        arg_vr(0) =  5.0_DP / 12.0_DP
        arg_vt(0) =  1.0_DP / 42.0_DP
        arg_w = 3.5_DP
    else if (arg_k == 7) then
        arg_st(0) =  1.0_DP
        arg_vr(0) = -1.0_DP / 42.0_DP
        arg_vt(0) =  4.0_DP / 21.0_DP
        arg_w = 3.5_DP
    else if (arg_k == 8) then
        arg_pr(0) =  1.0_DP
        arg_vr(0) =  7.0_DP / 16.0_DP
        arg_vt(0) = -1.0_DP / 72.0_DP
        arg_w = 4.5_DP
    else if (arg_k == 9) then
        arg_st(0) =  1.0_DP
        arg_vr(0) =  1.0_DP / 72.0_DP
        arg_vt(0) =  5.0_DP / 36.0_DP
        arg_w = 4.5_DP
    else if (arg_k == 10) then
        arg_pr(0) =  1.0_DP
        arg_vr(0) =  9.0_DP /  20.0_DP
        arg_vt(0) =  1.0_DP / 110.0_DP
        arg_w = 5.5_DP
    else if (arg_k == 11) then
        arg_st(0) =  1.0_DP
        arg_vr(0) = -1.0_DP / 110.0_DP
        arg_vt(0) = 54.0_DP / 495.0_DP
        arg_w = 5.5_DP
    else if (arg_k == 12) then
        arg_pr(0) =   1.0_DP
        arg_vr(0) =  11.0_DP /  24.0_DP
        arg_vt(0) =  -1.0_DP / 156.0_DP
        arg_w = 6.5_DP
    else if (arg_k == 13) then
        arg_st(0) =      1.0_DP
        arg_vr(0) =      1.0_DP /    156.0_DP
        arg_vt(0) =  19943.0_DP / 222222.0_DP
        arg_w = 6.5_DP
    else
        print*, 'Error in {innerBuild_Boundary}'
        stop
    end if
    end subroutine innerBuild_Boundary
```

```
!==============================================================================
    subroutine innerBuild_Cartesian(arg_k,arg_p,arg_f)
    implicit none
    integer,                  intent(in)  :: arg_k
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(12)               :: d
    real(DP)                              :: w
    call innerBuild_COEFFICIENTS(arg_k,w,d)
    call gensol_Cartesian(w,d,arg_p,arg_f)
    end subroutine innerBuild_Cartesian
!==============================================================================
    subroutine innerBuild_Polar(arg_k,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    integer,  intent(in)    :: arg_k
    real(DP), intent(in)    :: arg_r, arg_t
    real(DP), intent(out)   :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(12) :: d
    real(DP)                :: w
    call innerBuild_COEFFICIENTS(arg_k,w,d)
    call genSol_Polar(w,d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end subroutine innerBuild_Polar
!==============================================================================
    subroutine innerBuild_COEFFICIENTS(arg_k,arg_w,arg_d)
    integer,                intent(in)  :: arg_k
    real(DP),               intent(out) :: arg_w
    real(DP), dimension(12), intent(out) :: arg_d
    arg_d = 0.0_DP
    if (arg_k == 1) then
        arg_d( 9) =  1.0_DP / 6.0_DP
        arg_d(10) = -1.0_DP / 3.0_DP
        arg_d(11) =  1.0_DP / 2.0_DP
        arg_w = 0.5_DP
    else if (arg_k == 2) then
        arg_d( 9) = -1.0_DP / 60.0_DP
        arg_d(10) = -1.0_DP / 15.0_DP
        arg_d(11) =  1.0_DP / 12.0_DP
        arg_d(12) = -1.0_DP /  6.0_DP
        arg_w = 1.5_DP
    else if (arg_k == 3) then
        arg_d( 9) =  1.0_DP / 10.0_DP
        arg_d(10) =  1.0_DP / 15.0_DP
        arg_d(11) =  1.0_DP /  6.0_DP
        arg_w = 1.5_DP
    else if (arg_k == 4) then
        arg_d( 9) =  3.0_DP / 280.0_DP
        arg_d(10) = -9.0_DP / 140.0_DP
        arg_d(11) = -1.0_DP /  40.0_DP
        arg_d(12) = -1.0_DP /  10.0_DP
        arg_w = 2.5_DP
    else if (arg_k == 5) then
        arg_d( 9) =  1.0_DP / 28.0_DP
        arg_d(10) = -1.0_DP / 70.0_DP
        arg_d(11) =  1.0_DP / 20.0_DP
        arg_w = 2.5_DP
    else if (arg_k == 6) then
        arg_d( 9) = - 5.0_DP / 756.0_DP
        arg_d(10) = -10.0_DP / 189.0_DP
        arg_d(11) =   1.0_DP / 84.0_DP
        arg_d(12) = - 1.0_DP / 14.0_DP
```

```
        arg_w = 3.5_DP
    else if (arg_k == 7) then
        arg_d( 9) = 1.0_DP /  54.0_DP
        arg_d(10) = 1.0_DP / 189.0_DP
        arg_d(11) = 1.0_DP /  42.0_DP
        arg_w = 3.5_DP
    else if (arg_k == 8) then
        arg_d( 9) =   7.0_DP / 1584.0_DP
        arg_d(10) = -35.0_DP /  792.0_DP
        arg_d(11) = - 1.0_DP /  144.0_DP
        arg_d(12) = - 1.0_DP /   18.0_DP
        arg_w = 4.5_DP
    else if (arg_k == 9) then
        arg_d( 9) =  1.0_DP /  88.0_DP
        arg_d(10) = -1.0_DP / 396.0_DP
        arg_d(11) =  1.0_DP /  72.0_DP
        arg_w = 4.5_DP
    else if (arg_k == 10) then
        arg_d( 9) = - 9.0_DP / 2860.0_DP
        arg_d(10) = -27.0_DP /  715.0_DP
        arg_d(11) =   1.0_DP /  220.0_DP
        arg_d(12) = - 1.0_DP /   22.0_DP
        arg_w = 5.5_DP
    else if (arg_k == 11) then
        arg_d( 9) =  1.0_DP / 130.0_DP
        arg_d(10) =  1.0_DP / 715.0_DP
        arg_d(11) =  1.0_DP / 110.0_DP
        arg_w = 5.5_DP
    else if (arg_k == 12) then
        arg_d( 9) =  31339.0_DP / 13333320.0_DP
        arg_d( 9) =  31339.0_DP          / (120.0_DP * 111111.0_DP)
        arg_d(10) = -31339.0_DP * 7.0_DP / ( 60.0_DP * 111111.0_DP)
        arg_d(11) = -1.0_DP / 312.0_DP
        arg_d(12) = -1.0_DP /  26.0_DP
        arg_w = 6.5_DP
    else if (arg_k == 13) then
        arg_d( 9) =  1.0_DP /  180.0_DP
        arg_d(10) = -1.0_DP / 1170.0_DP
        arg_d(11) =  1.0_DP /  156.0_DP
        arg_w = 6.5_DP
    else
        print*, 'Error in {innerBuild_COEFFICIENTS}'
        print*, arg_k
        stop
    end if
    end subroutine innerBuild_COEFFICIENTS
!=======================================================================
end module mod_innerBuild
!#######################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_innerBuildCheck
use mod_constants
use mod_stokesEq2d
use mod_innerBuild
implicit none
private
public :: innerBuildCheck
real(DP), dimension(13) :: sav_c
contains
```

```
!================================================================================
    subroutine innerBuildCheck
    call random_number(sav_c)
    print 100, sav_c
    call stokesEq2d_Cartesian(innerBuildCheck_CARTESIAN,'innerBuild_Cartesian.txt')
    call stokesEq2d_Polar(innerBuildCheck_POLAR,'innerBuild_Polar.txt')
    call innerBuildCheck_BOUNDARY
100 format(/ 'innerBuildCheck: Random Coefficients' / 13(f12.7 /) /)
    end subroutine innerBuildCheck
!================================================================================
    subroutine innerBuildCheck_BOUNDARY
    integer                  :: i, iMax, k
    real(DP)                 :: r, rMax, rMin, w, t
    real(DP)                 :: st1, pr1, vr1, vt1
    real(DP)                 :: st2, pr2, vr2, vt2
    real(DP)                 :: st3, pr3, vr3, vt3
    real(DP)                 :: error, maxErrorMax
    real(DP), dimension(13)  :: errorMax
    real(DP), dimension(0:2) :: pr, vr, vt, st
    open(unit = 20, file = 'innerBuild_Boundary.txt')
    rMin = 0.5_DP
    rMax = 1.5_DP
    iMax = 10
    do k = 1, 13
        call innerBuild_Boundary(k,w,pr,vr,vt,st)
        write(20,100) k
        write(20,200) 'st', st(2), st(1), st(0), w - 1.0_DP
        write(20,200) 'pr', pr(2), pr(1), pr(0), w - 1.0_DP
        write(20,200) 'vr', vr(2), vr(1), vr(0), w
        write(20,200) 'vt', vt(2), vt(1), vt(0), w
        write(20,300)
        errorMax(k) = 0.0_DP
        do i = 0, iMax
            r = rMin + (rMax - rMin) * real(i,DP) / real(iMax,DP)
            st1 = (st(2) * log(r)**2 + st(1) * log(r) + st(0)) * r**(w - 1.0_DP)
            pr1 = (pr(2) * log(r)**2 + pr(1) * log(r) + pr(0)) * r**(w - 1.0_DP)
            vr1 = (vr(2) * log(r)**2 + vr(1) * log(r) + vr(0)) * r**w
            vt1 = (vt(2) * log(r)**2 + vt(1) * log(r) + vt(0)) * r**w
            t = 0.0_DP
            call innerBuild_Polar(k,r,t,pr2,vr2,vt2,st2)
            t = PI / 2.0_DP
            call innerBuild_Polar(k,r,t,pr3,vr3,vt3,st3)
            write(20,400) r, vr3, vt3, st1, st2, pr1, pr2, vr1, vr2, vt1, vt2
            error = max(abs(vr3),abs(vt3),abs(st1-st2),abs(pr1-pr2),abs(vr1-vr2),abs(vt1-vt2))
            if (error > errorMax(k)) then
                errorMax(k) = error
            end if
        end do
        write(20,500) errorMax(k)
    end do
    write(20,600)
    maxErrorMax = maxval(errorMax)
    write(20,700) maxErrorMax
    close(20)
100 format (157('-') / 'Inner Solution Building Block: ', i2 /)
200 format (a, '(r,0) = [ (', f10.6, ') ln(r)^2 + (', f10.6, ') ln(r) + (', f10.6, ') ] r^(', f4.1, ')')
300 format (/ '           r', 4x, '    vr(r,90)', 1x, '    vt(r,90)', 4x, &
              ' stress(r,0)', 1x, '  stess(r,0)', 4x, '     pr(r,0)', 1x, &
              '     pr(r,0)', 4x, '     vr(r,0)', 1x, '     vr(r,0)', 4x, &
              '     vt(r,0)', 1x, '     vt(r,0)' /                       &
```

```
             12('-'), 5(4x, 12('-'), 1x, 12('-')))
400 format(f12.7, 5(4x, f12.7, 1x, f12.7))
500 format(/ 'Maximum Error: ', e12.5)
600 format(157('-'))
700 format('Overall Maximum Error: ', e12.5 / 157('-'))
    end subroutine innerBuildCheck_BOUNDARY
!==============================================================================
    subroutine innerBuildCheck_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(13,0:3)           :: f
    real(DP), dimension(13)               :: c
    integer                               :: k
    do k = 1, 13
        call innerBuild_Cartesian(k,arg_p,f(k,:))
    end do
    arg_f = matmul(sav_c,f)
    end subroutine innerBuildCheck_CARTESIAN
!==============================================================================
    subroutine innerBuildCheck_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)    :: arg_r, arg_t
    real(DP), intent(out)   :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP), dimension(13) :: st, pr, vr, vt
    integer                 :: k
    do k = 1, 13
        call innerBuild_Polar(k,arg_r,arg_t,pr(k),vr(k),vt(k),st(k))
    end do
    arg_pr = dot_product(sav_c,pr)
    arg_vr = dot_product(sav_c,vr)
    arg_vt = dot_product(sav_c,vt)
    arg_st = dot_product(sav_c,st)
    end subroutine innerBuildCheck_POLAR
!==============================================================================
end module mod_innerBuildCheck
!##############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!----------------------------------
module mod_similarity
use mod_constants
use mod_utilities
use mod_fullmat
implicit none
private
public :: similarity_determinant
public :: similarity_power
public :: similarity_Coefficients
public :: similarity_Boundary
public :: similarity_Cartesian
public :: similarity_Polar
real(DP), parameter :: SAV_TOL = 1.0e-10_DP
contains
!==============================================================================
    function similarity_determinant(arg_angle,arg_w) result(arg_f)
    implicit none
    real(DP), parameter      :: TOL = 1.0e-8_DP
    integer,  intent(in)     :: arg_angle
    real(DP), intent(in)     :: arg_w
    real(DP)                 :: arg_f
```

```
    real(DP), dimension(4)   :: fp, fr, ft, fs
    real(DP), dimension(4)   :: gp, gr, gt, gs
    real(DP), dimension(4,4) :: a
    real(DP)                 :: t1, t2
    t1 = 0.0_DP
    t2 = arg_angle * PI / 180.0_DP
    if (abs(arg_w - 1.0_DP) < TOL) then
        call similarity_FUNCTIONS_PLUS_1(t1,fp,fr,ft,fs)
        call similarity_FUNCTIONS_PLUS_1(t2,gp,gr,gt,gs)
    else
        call similarity_FUNCTIONS(t1,arg_w,fp,fr,ft,fs)
        call similarity_FUNCTIONS(t2,arg_w,gp,gr,gt,gs)
    end if
    a(1,:) = gr
    a(2,:) = gt
    a(3,:) = fp
    a(4,:) = fs
    arg_f = det4(a)
    end function similarity_determinant
!================================================================================
    function similarity_power(arg_angle,opt_k) result(arg_w)
    implicit none
    integer, intent(in)           :: arg_angle
    integer, intent(in), optional :: opt_k
    real(DP)                      :: arg_w
    real(DP)                      :: wL, wM, wH
    real(DP)                      :: dL, dM, dH
    real(DP)                      :: tol
    tol = SAV_TOL / 10.0_DP
    if (arg_angle < 10) then
        print*, 'Error #1 in {similarity_power}'
        stop
    else if (arg_angle < 20) then
        wL = 2.0_DP
        wH = 7.0_DP
    else if (arg_angle < 45) then
        wL = 1.001_DP
        wH = 4.0_DP
    else if (arg_angle == 45) then
        arg_w = 1.0_DP
        return
    else if (arg_angle < 90) then
        wL = 0.4_DP
        wH = 0.999_DP
    else if (arg_angle == 135) then
        if (present(opt_k)) then
            if (opt_k == 1) then
                wL = 0.4_DP
                wH = 0.5_DP
            else if (opt_k == 2) then
                arg_w = 1.0_DP
                return
            else
                wL = 1.5_DP
                wH = 1.6_DP
            end if
        else
            wL = 0.4_DP
            wH = 0.5_DP
        end if
```

```
    else if (arg_angle < 180) then
        wL = 0.4_DP
        wH = 0.505_DP
    else
        print*, 'Error #2 in {similarity_power}'
        stop
    end if
    dL = similarity_determinant(arg_angle,wL)
    dH = similarity_determinant(arg_angle,wH)
    if (dL * dH <= 0.0_DP) then
10      wM = (wL + wH) / 2.0_DP
        dM = similarity_determinant(arg_angle,wM)
        if (wH - wL > tol .or. abs(dM) > tol) then
            if (dL * dM > 0.0_DP) then
                wL = wM
            else
                wH = wM
            end if
            goto 10
        else
            arg_w = wM
        end if
    else
        print*, 'Error in {similarity_power}'
        print*, 'Interval does not bracket root'
        stop
    end if
    end function similarity_power
!=========================================================================
    subroutine similarity_Coefficients(arg_angle,arg_w,arg_co,arg_vt,arg_vr)
    implicit none
    real(DP),                 parameter   :: TOL = 1.0e-8_DP
    integer,                  intent(in)  :: arg_angle
    real(DP),                 intent(in)  :: arg_w
    real(DP), dimension(2,4), intent(out) :: arg_co
    real(DP), dimension(2),   intent(out) :: arg_vt
    real(DP), dimension(2),   intent(out) :: arg_vr
    real(DP), dimension(4)                :: fp, fr, ft, fs
    real(DP), dimension(4)                :: gp, gr, gt, gs
    real(DP), dimension(5,4)              :: a
    real(DP), dimension(5)                :: b
    real(DP), dimension(4)                :: x
    real(DP)                              :: t1, t2
    type(typ_fullmat)                     :: z
    t1 = 0.0_DP
    t2 = arg_angle * PI / 180.0_DP
    if (abs(arg_w - 1.0_DP) < TOL) then
        call similarity_FUNCTIONS_PLUS_1(t1,fp,fr,ft,fs)
        call similarity_FUNCTIONS_PLUS_1(t2,gp,gr,gt,gs)
    else
        call similarity_FUNCTIONS(t1,arg_w,fp,fr,ft,fs)
        call similarity_FUNCTIONS(t2,arg_w,gp,gr,gt,gs)
    end if
    a(1,:) = gr
    a(2,:) = gt
    a(3,:) = fp
    a(4,:) = fs
    a(5,:) = ft
    call z%SetMatrix(a(1:4,:))
    if (abs(z%getDeterm()) > SAV_TOL) then
```

```
        x = 0.0_DP
        x(3) = 1.0_DP
        call z%SolSystem(x)
        arg_co(1,:) = x
        arg_vt(1) = dot_product(x,ft)
        arg_vr(1) = dot_product(x,fr)
        x = 0.0_DP
        x(4) = 1.0_DP
        call z%SolSystem(x)
        arg_co(2,:) = x
        arg_vt(2) = dot_product(x,ft)
        arg_vr(2) = dot_product(x,fr)
    else
        b = 0.0
        b(5) = 1.0_DP
        call z%SetMatrix(matmul(transpose(a),a))
        x = matmul(transpose(a),b)
        call z%SolSystem(x)
        arg_co(1,:) = x
        arg_vt(1) = dot_product(x,ft)
        arg_vr(1) = dot_product(x,fr)
        arg_co(2,:) = 0.0_DP
        arg_vt(2) = 0.0_DP
        arg_vr(2) = 0.0_DP
    end if
100 format('Determinant: ', e12.5)
200 format(4(f21.14 /))
    end subroutine similarity_Coefficients
!===============================================================================
    subroutine similarity_Boundary(arg_angle,arg_w,arg_d)
    implicit none
    real(DP),               parameter  :: TOL = 1.0e-8_DP
    integer,                intent(in) :: arg_angle
    real(DP),               intent(in) :: arg_w
    real(DP), dimension(4), intent(in) :: arg_d
    real(DP), dimension(4)             :: fp, fr, ft, fs
    real(DP), dimension(4)             :: gp, gr, gt, gs
    real(DP)                           :: t1, t2
    integer                            :: a1, a2
    a1 = 0
    a2 = arg_angle
    t1 = real(a1,DP) * PI / 180.0_DP
    t2 = real(a2,DP) * PI / 180.0_DP
!
    if (abs(arg_w - 1.0_DP) < TOL) then
        call similarity_FUNCTIONS_PLUS_1(t1,fp,fr,ft,fs)
        call similarity_FUNCTIONS_PLUS_1(t2,gp,gr,gt,gs)
    else
        call similarity_FUNCTIONS(t1,arg_w,fp,fr,ft,fs)
        call similarity_FUNCTIONS(t2,arg_w,gp,gr,gt,gs)
    end if
!
100 format (a, '(r,', i3, ') = (', f17.10, ') r^(', f9.5, ')')
150 format (a, '(r,', i3, ') = (', f17.10, ') ln(r)')
200 format ()
300 format (79('-'))
    end subroutine similarity_Boundary
!===============================================================================
    subroutine similarity_Cartesian(arg_w,arg_d,arg_p,arg_f)
    real(DP),                intent(in)  :: arg_w
```

```
    real(DP), dimension(4),   intent(in)  :: arg_d
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(arg_p,arg_p))
    t = angle(arg_p)
    call similarity_Polar(arg_w,arg_d,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    arg_f = (/ pr, vx, vy, st /)
    end subroutine similarity_Cartesian
!================================================================================
    subroutine similarity_Polar(arg_w,arg_d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP),               parameter   :: TOL = 1.0e-8_DP
    real(DP), dimension(4), intent(in)  :: arg_d
    real(DP),               intent(in)  :: arg_w, arg_r, arg_t
    real(DP),               intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    real(DP), dimension(4)              :: fp, fr, ft, fs
    if (arg_r < TOL) then
        arg_st = 0.0_DP
        arg_pr = 0.0_DP
        arg_vr = 0.0_DP
        arg_vt = 0.0_DP
    else if (abs(arg_w - 1.0_DP) < TOL) then
        call similarity_FUNCTIONS_PLUS_1(arg_t,fp,fr,ft,fs)
        arg_st = dot_product(arg_d,fs)
        arg_pr = dot_product(arg_d,fp) * log(arg_r)
        arg_vr = dot_product(arg_d,fr) * arg_r
        arg_vt = dot_product(arg_d,ft) * arg_r
    else
        call similarity_FUNCTIONS(arg_t,arg_w,fp,fr,ft,fs)
        arg_st = dot_product(arg_d,fs) * arg_r**(arg_w - 1.0_DP)
        arg_pr = dot_product(arg_d,fp) * arg_r**(arg_w - 1.0_DP)
        arg_vr = dot_product(arg_d,fr) * arg_r**arg_w
        arg_vt = dot_product(arg_d,ft) * arg_r**arg_w
    end if
    end subroutine similarity_Polar
!================================================================================
    subroutine similarity_FUNCTIONS_PLUS_1(arg_t,arg_fp,arg_fr,arg_ft,arg_fs)
    implicit none
    real(DP),               intent(in)  :: arg_t
    real(DP), dimension(4), intent(out) :: arg_fp, arg_fr, arg_ft, arg_fs
    real(DP)                            :: c, s
    c = cos(2 * arg_t)
    s = sin(2 * arg_t)
    arg_fp(1) =  0.0_DP
    arg_fp(2) =  0.0_DP
    arg_fp(3) =  0.0_DP
    arg_fp(4) = -4.0_DP
    arg_fr(1) =  2.0_DP * s
    arg_fr(2) = -2.0_DP * c
    arg_fr(3) =  0.0_DP
    arg_fr(4) = -1.0_DP
    arg_ft(1) =  2.0_DP * c
    arg_ft(2) =  2.0_DP * s
    arg_ft(3) =  2.0_DP
    arg_ft(4) =  2.0_DP * arg_t
    arg_fs(1) =  4.0_DP * c
    arg_fs(2) =  4.0_DP * s
    arg_fs(3) =  0.0_DP
```

```
    arg_fs(4) =  0.0_DP
    end subroutine similarity_FUNCTIONS_PLUS_1
!=============================================================================
    subroutine similarity_FUNCTIONS(arg_t,arg_w,arg_fp,arg_fr,arg_ft,arg_fs)
    implicit none
    real(DP),               intent(in)  :: arg_t, arg_w
    real(DP), dimension(4), intent(out) :: arg_fp, arg_fr, arg_ft, arg_fs
    arg_fp = -similarity_ALPHA(1,arg_t,arg_w) / (arg_w - 1.0_DP)
    arg_fr = -similarity_LAMBDA(1,arg_t,arg_w)
    arg_ft = (arg_w + 1.0_DP) * similarity_KAPPA(0,arg_t,arg_w)
    arg_fs = similarity_KAPPA(2,arg_t,arg_w) - (1.0_DP - arg_w**2) * similarity_KAPPA(0,arg_t,arg_w)
    end subroutine similarity_FUNCTIONS
!=============================================================================
    function similarity_KAPPA(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    real(DP)               :: mu, nu
    mu = w + 1.0_DP
    nu = w - 1.0_DP
    f(1) = mu**i * cos(mu * t)
    f(2) = mu**i * sin(mu * t)
    f(3) = nu**i * cos(nu * t)
    f(4) = nu**i * sin(nu * t)
    end function similarity_KAPPA
!=============================================================================
    function similarity_LAMBDA(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    real(DP)               :: mu, nu
    mu = w + 1.0_DP
    nu = w - 1.0_DP
    f(1) = -mu**i * sin(mu * t)
    f(2) =  mu**i * cos(mu * t)
    f(3) = -nu**i * sin(nu * t)
    f(4) =  nu**i * cos(nu * t)
    end function similarity_LAMBDA
!=============================================================================
    function similarity_ALPHA(i,t,w) result(f)
    integer,  intent(in)   :: i
    real(DP), intent(in)   :: t, w
    real(DP), dimension(4) :: f
    select case(i)
        case(1)
            f = (w + 1.0_DP)**2 * similarity_LAMBDA(1,t,w) - similarity_LAMBDA(3,t,w)
        case(2)
            f = 2.0_DP * (w + 1.0_DP)**2 * (similarity_LAMBDA(0,t,w) &
                - t * similarity_KAPPA(1,t,w)) - 6.0_DP * similarity_LAMBDA(2,t,w) &
                + 2.0_DP * t * similarity_KAPPA(3,t,w)
        case(3)
            f = 6.0_DP * (t * similarity_KAPPA(2,t,w) - similarity_LAMBDA(1,t,w)) &
                + t**2 * similarity_LAMBDA(3,t,w) - (w + 1.0_DP)**2 * t * &
                (2.0_DP * similarity_KAPPA(0,t,w) + t * similarity_LAMBDA(1,t,w))
        case(4)
            f = t * similarity_KAPPA(3,t,w) - 3.0_DP * similarity_LAMBDA(2,t,w) &
                + (w + 1.0_DP)**2 * (similarity_LAMBDA(0,t,w) - t * similarity_KAPPA(1,t,w))
        case default
            f = 0.0d0
    end select
```

```
    end function similarity_ALPHA
!==============================================================================
end module mod_similarity
!##############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!----------------------------------
module mod_similarityCheck
use mod_constants
use mod_utilities
use mod_similarity
use mod_stokesEq2d
implicit none
private
public :: similarityCheck
real(DP)              :: sav_w
real(DP), dimension(4) :: sav_d
contains
!==============================================================================
    subroutine similarityCheck
    implicit none
    real(DP), parameter      :: TOL = 1.0e-6_DP
    integer                  :: angle
    real(DP), dimension(2,4) :: co
    real(DP), dimension(2)   :: vt, vr
    real(DP)                 :: w, wMax, wMin, wInc, power, det
    integer                  :: i, iMax, k
    integer, dimension(17)   :: angles
    character(len=3)         :: number
    sav_w = 2.7_DP
    sav_d = (/  1.37_DP,  5.76_DP, -6.32_DP, -7.98_DP /)
    call stokesEq2d_Cartesian(similarityCheck_CARTESIAN,'similarity_Cartesian.txt')
    call stokesEq2d_Polar(similarityCheck_POLAR,'similarity_Polar.txt')
    wInc = 1.0e-3_DP
    wMin = wInc
    wMax = 10.0_DP
    iMax = nint((wMax - wMin) / wInc)
    angles = (/ 10, 15, 20, 30, 35, 40, 45, 50, 60, 90, 120, 125, 130, 135, 140, 145, 150 /)
    do k = 1, 17
        call numChar(angles(k),number)
        open(unit = 30, file = 'similarity_Determinant_' // number // '.txt')
        do i = 0, iMax
            w = wMin + (wMax - wMin) * real(i,DP) / real(iMax,DP)
            if (abs(w - 1.0_DP) < TOL) then
                cycle
            end if
            det = similarity_determinant(angles(k),w)
            write(30,100) w, log10(abs(det)), sgn(det), det
        end do
        close(30)
    end do
    open(unit = 30, file = 'similarity_Power.txt')
    do angle = 10, 179
        write(30,200) real(angle,DP), real(angle,DP) * Pi / 180.0_DP, similarity_power(angle)
    end do
    close(30)
100 format(f12.7, 5x, e12.5, 5x, f12.7, 5x, e12.5)
200 format(f5.1, 1x, f12.7, 5x, f12.7)
    end subroutine similarityCheck
!==============================================================================
    subroutine similarityCheck_CARTESIAN(arg_p,arg_f)
```

```
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    call similarity_Cartesian(sav_w,sav_d,arg_p,arg_f)
    end subroutine similarityCheck_CARTESIAN
!===============================================================================
    subroutine similarityCheck_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    call similarity_Polar(sav_w,sav_d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end subroutine similarityCheck_PULAR
!===============================================================================
end module mod_similarityCheck
!###############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_innerSeries
use mod_constants
use mod_utilities
use mod_similarity
implicit none
private
type, public :: typ_innerSeries
    private
    integer                                 :: angle = 90
    integer                                 :: k = 1
    integer                                 :: n = 25
    real(DP)                                :: s = 8.3_DP
    real(DP), dimension(:,:,:), allocatable :: co
    real(DP), dimension(:,:),   allocatable :: vt
    real(DP), dimension(:,:),   allocatable :: vr
    real(DP), dimension(:,:),   allocatable :: a
    real(DP), dimension(:),     allocatable :: w
contains
    procedure :: SetAngle      => innerSeries_SetAngle
    procedure :: SetSolution   => innerSeries_SetSolution
    procedure :: SetOrder      => innerSeries_SetOrder
    procedure :: SetSlip       => innerSeries_SetSlip
    procedure :: Coefficients  => innerSeries_Coefficients
    procedure :: errorPressure => innerSeries_errorPressure
    procedure :: errorStress   => innerSeries_errorStress
    procedure :: Boundary      => innerSeries_Boundary
    procedure :: Cartesian     => innerSeries_Cartesian
    procedure :: Polar         => innerSeries_Polar
end type typ_innerSeries
contains
!===============================================================================
    subroutine innerSeries_SetAngle(this,arg_angle)
    class(typ_innerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_angle
    this%angle = arg_angle
    end subroutine innerSeries_SetAngle
!===============================================================================
    subroutine innerSeries_SetSolution(this,arg_k)
    implicit none
    class(typ_innerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_k
    this%k = clip(arg_k,1,3)
    end subroutine innerSeries_SetSolution
```

```
!==============================================================================
    subroutine innerSeries_SetOrder(this,arg_n)
    class(typ_innerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_n
    this%n = abs(arg_n)
    call innerSeries_DEALLOCATE(this)
    allocate(this%co(0:this%n,2,4))
    allocate(this%vt(0:this%n,2))
    allocate(this%vr(0:this%n,2))
    allocate(this%a(0:this%n,2))
    allocate(this%w(0:this%n))
    end subroutine innerSeries_SetOrder
!==============================================================================
    subroutine innerSeries_SetSlip(this,arg_s)
    class(typ_innerSeries), intent(inout) :: this
    real(DP),               intent(in)    :: arg_s
    integer                               :: n
    this%s = abs(arg_s)
    call innerSeries_RECURSION(this)
    end subroutine innerSeries_SetSlip
!==============================================================================
    subroutine innerSeries_Coefficients(this,arg_filename)
    class(typ_innerSeries), intent(inout) :: this
    character(len=*),       intent(in)    :: arg_filename
    integer                               :: n
    open(unit = 20, file = arg_filename)
    do n = 0, this%n
        write(20,100) n, this%a(n,:)
    end do
    close(20)
100 format('n = ', i3, 4x, 'a1 = ', e17.10, 4x, 'a2 = ', e17.10)
    end subroutine innerSeries_Coefficients
!==============================================================================
    function innerSeries_errorPressure(this,arg_r) result(arg_f)
    implicit none
    class(typ_innerSeries), intent(in) :: this
    real(DP),               intent(in) :: arg_r
    real(DP)                           :: arg_f
    arg_f =  (  this%a(this%n,1) * this%vt(this%n,1) &
             + this%a(this%n,2) * this%vt(this%n,2)) * arg_r**this%w(this%n)
    end function innerSeries_errorPressure
!==============================================================================
    function innerSeries_errorStress(this,arg_r) result(arg_f)
    implicit none
    class(typ_innerSeries), intent(in) :: this
    real(DP),               intent(in) :: arg_r
    real(DP)                           :: arg_f
    arg_f = -(  this%a(this%n,1) * this%vr(this%n,1) &
             + this%a(this%n,2) * this%vr(this%n,2)) * arg_r**this%w(this%n) / this%s
    end function innerSeries_errorStress
!==============================================================================
    subroutine innerSeries_Boundary(this,arg_filename)
    implicit none
    class(typ_innerSeries), intent(in) :: this
    character(len=*),       intent(in) :: arg_filename
    integer                            :: i, iMax
    real(DP)                           :: r, rMax, rMin
    real(DP)                           :: pr, vr, vt, st
    real(DP)                           :: e1, e2, f1, f2
    real(DP)                           :: error1, error2, distance, cutoff
```

```
    cutoff = 1.0e-2_DP
    rMin = 1.0e-3_DP
    rMax = 1.0e1_DP
    iMax = 100
    error1 = 0.0_DP
    error2 = 0.0_DP
    distance = 0.0_DP
    open(unit = 20, file = arg_filename)
    do i = 0, iMax
        r = exp(log(rMin) + log(rMax / rMin) * real(i,DP) / real(iMax,DP))
        call innerSeries_Polar(this,r,0.0_DP,pr,vr,vt,st)
        f1 = innerSeries_errorPressure(this,r)
        f2 = innerSeries_errorStress(this,r)
        e1 = pr + vt
        e2 = st - vr / this%s
        write(20,100) r, pr, vt, e1, f1, st, vr / this%s, e2, f2
        if (abs(e1 - f1) > error1) then
            error1 = abs(e1 - f1)
        end if
        if (abs(e2 - f2) > error2) then
            error2 = abs(e2 - f2)
        end if
        if (max(abs(e1),abs(e2)) > cutoff) then
            distance = r
        end if
    end do
    close(20)
    print 200, error1, error2, distance

100 format(e12.5, 2(8x, e15.8, 1x, e15.8, 8x, e12.5, 1x, e12.5))
200 format('innerSeries: Solution' / &
           5x, 'Error1   = ', e12.5 / 5x, 'Error2   = ', e12.5 / &
           5x, 'Distance = ', e12.5 /)
    end subroutine innerSeries_Boundary
!===============================================================================
    subroutine innerSeries_Cartesian(this,arg_p,arg_f)
    class(typ_innerSeries), intent(in)  :: this
    real(DP), dimension(2),     intent(in)  :: arg_p
    real(DP), dimension(0:3),   intent(out) :: arg_f
    real(DP), dimension(0:3)                :: f
    integer                                 :: k, n
    arg_f = 0.0_DP
    do n = 0, this%n
        do k = 1, 2
            call similarity_Cartesian(this%w(n),this%co(n,k,:),arg_p,f)
            arg_f = arg_f + this%a(n,k) * f
        end do
    end do
    end subroutine innerSeries_Cartesian
!===============================================================================
    subroutine innerSeries_Polar(this,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    class(typ_innerSeries), intent(in)  :: this
    real(DP),               intent(in)  :: arg_r, arg_t
    real(DP),               intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP)                            :: st, pr, vr, vt, cs, cp
    integer                             :: k, n
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
```

```
    arg_st = 0.0_DP
    do n = 0, this%n
        do k = 1, 2
            call similarity_Polar(this%w(n),this%co(n,k,:),arg_r,arg_t,pr,vr,vt,st)
            arg_pr = arg_pr + this%a(n,k) * pr
            arg_vr = arg_vr + this%a(n,k) * vr
            arg_vt = arg_vt + this%a(n,k) * vt
            arg_st = arg_st + this%a(n,k) * st
        end do
    end do
    end subroutine innerSeries_Polar
!=============================================================================
    subroutine innerSeries_RECURSION(this)
    implicit none
    class(typ_innerSeries), intent(inout) :: this
    real(DP), dimension(0:2)                  :: pr1, vr1, vt1, st1
    real(DP), dimension(0:2)                  :: pr2, vr2, vt2, st2
    real(DP)                                  :: w
    integer                                   :: i, j, n
    this%w(0) = similarity_power(this%angle,this%k)
    call similarity_Coefficients(this%angle,this%w(0),this%co(0,:,:),this%vt(0,:),this%vr(0,:))
    call similarity_Boundary(this%angle,this%w(0),this%co(0,1,:))
    call similarity_Boundary(this%angle,this%w(0),this%co(0,2,:))
    this%a(0,1) = -1.0_DP
    this%a(0,2) =  0.0_DP
    do n = 1, this%n
        this%w(n) = this%w(0) + real(n,DP)
        call similarity_Coefficients(this%angle,this%w(n),this%co(n,:,:),this%vt(n,:),this%vr(n,:))
        this%a(n,1) = -(this%a(n-1,1) * this%vt(n-1,1) + this%a(n-1,2) * this%vt(n-1,2))
        this%a(n,2) =  (this%a(n-1,1) * this%vr(n-1,1) + this%a(n-1,2) * this%vr(n-1,2)) / this%s
    end do
100 format(f12.7)
    end subroutine innerSeries_RECURSION
!=============================================================================
    subroutine innerSeries_DEALLOCATE(this)
    implicit none
    class(typ_innerSeries), intent(inout) :: this
    if (allocated(this%co)) then
        deallocate(this%co)
    end if
    if (allocated(this%vt)) then
        deallocate(this%vt)
    end if
    if (allocated(this%vr)) then
        deallocate(this%vr)
    end if
    if (allocated(this%a)) then
        deallocate(this%a)
    end if
    if (allocated(this%w)) then
        deallocate(this%w)
    end if
    end subroutine innerSeries_DEALLOCATE
!=============================================================================
end module mod_innerSeries
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_innerSeriesCheck
use mod_constants
```

```
use mod_innerSeries
use mod_stokesEq2d
implicit none
private
public :: innerSeriesCheck
type(typ_innerSeries) :: sav_a
contains
!================================================================================
    subroutine innerSeriesCheck
    implicit none
    integer                   :: angle, i, k, m, n
    real(DP), dimension(10)   :: r, e
    real(DP), dimension(7,13) :: c
    real(DP)                  :: s
    print*, 'innerSeriesCheck'
    print*, 'angle ?'
    read *,  angle
    print*, 'k ?'
    read *,  k
    print*, 'n ?'
    read *,  n
    print*, 's ?'
    read *,  s
    call sav_a%SetAngle(angle)
    call sav_a%SetSolution(k)
    call sav_a%SetOrder(n)
    call sav_a%SetSlip(s)
    call sav_a%Coefficients('innerSeries_Coefficients.txt')
    call sav_a%Boundary('innerSeries_Boundary' // char(48 + k) // '.txt')
    call stokesEq2d_Cartesian(innerSeriesCheck_CARTESIAN,'innerSeries_Cartesian' // char(48 + k) // '.txt')
    call stokesEq2d_Polar    (innerSeriesCheck_POLAR    ,'innerSeries_Polar'     // char(48 + k) // '.txt')
    r( 1) = 1.0e-2_DP
    r( 2) = 2.0e-2_DP
    r( 3) = 5.0e-2_DP
    r( 4) = 1.0e-1_DP
    r( 5) = 2.0e-1_DP
    r( 6) = 5.0e-1_DP
    r( 7) = 1.0e0_DP
    r( 8) = 2.0e0_DP
    r( 9) = 5.0e0_DP
    r(10) = 1.0e1_DP
    open(unit = 20, file = 'innerSeries_Errors' // char(48 + k) // '.txt')
    write(20,100) r
    do m = 0, n
        call sav_a%SetOrder(m)
        call sav_a%SetSlip(s)
        do i = 1, 10
            e(i) = sav_a%errorPressure(r(i))
        end do
        write(20,140) m, e
    end do
    write(20,120) r
    do m = 0, n
        call sav_a%SetOrder(m)
        call sav_a%SetSlip(s)
        do i = 1, 10
            e(i) = sav_a%errorStress(r(i))
        end do
        write(20,140) m, e
    end do
```

```
    close(20)
100 format(   'Series errors in pressure' // '  n', 10(1x, e9.2) / '---', 10(1x, 9('-')))
120 format(// 'Series errors in stress'   // '  n', 10(1x, e9.2) / '---', 10(1x, 9('-')))
140 format(i3, 110(1x, e9.2))
    end subroutine innerSeriesCheck
!=============================================================================
    subroutine innerSeriesCheck_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    call sav_a%Cartesian(arg_p,arg_f)
    end subroutine innerSeriesCheck_CARTESIAN
!=============================================================================
    subroutine innerSeriesCheck_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    call sav_a%Polar(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end subroutine innerSeriesCheck_POLAR
!=============================================================================
end module mod_innerSeriesCheck
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_similarity1
use mod_constants
use mod_utilities
use mod_generalSolution
use mod_fullmat
implicit none
private
public :: similarity1_Coefficients
public :: similarity1_Boundary
public :: similarity1_Cartesian
public :: similarity1_Polar
contains
!=============================================================================
    subroutine similarity1_Coefficients(arg_angle,arg_w,arg_co,arg_pr,arg_st)
    implicit none
    integer,                   intent(in)  :: arg_angle
    real(DP),                  intent(in)  :: arg_w
    real(DP), dimension(4,8),  intent(out) :: arg_co
    real(DP), dimension(4,0:1), intent(out) :: arg_pr
    real(DP), dimension(4,0:1), intent(out) :: arg_st
    real(DP), dimension(0:1,8)             :: fp, fr, ft, fs
    real(DP), dimension(0:1,8)             :: gp, gr, gt, gs
    real(DP), dimension(8,8)               :: a
    real(DP), dimension(8)                 :: x
    real(DP)                               :: t1, t2
    type(typ_fullmat)                      :: z
    t1 = 0.0_DP
    t2 = arg_angle * PI / 180.0_DP
    call similarity1_FUNCTIONS(t1,arg_w,fp,fr,ft,fs)
    call similarity1_FUNCTIONS(t2,arg_w,gp,gr,gt,gs)
    a(1,:) = ft(0,:)
    a(2,:) = ft(1,:)
    a(3,:) = fr(0,:)
    a(4,:) = fr(1,:)
    a(5,:) = gt(0,:)
    a(6,:) = gt(1,:)
```

```
    a(7,:) = gr(0,:)
    a(8,:) = gr(1,:)
    call z%SetMatrix(a)
    x = 0.0_DP
    x(1) = 1.0_DP
    call z%SolSystem(x)
    arg_co(1,:) = x
    arg_pr(1,0) = dot_product(x,fp(0,:))
    arg_pr(1,1) = dot_product(x,fp(1,:))
    arg_st(1,0) = dot_product(x,fs(0,:))
    arg_st(1,1) = dot_product(x,fs(1,:))
    x = 0.0_DP
    x(2) = 1.0_DP
    call z%SolSystem(x)
    arg_co(2,:) = x
    arg_pr(2,0) = dot_product(x,fp(0,:))
    arg_pr(2,1) = dot_product(x,fp(1,:))
    arg_st(2,0) = dot_product(x,fs(0,:))
    arg_st(2,1) = dot_product(x,fs(1,:))
    x = 0.0_DP
    x(3) = 1.0_DP
    call z%SolSystem(x)
    arg_co(3,:) = x
    arg_pr(3,0) = dot_product(x,fp(0,:))
    arg_pr(3,1) = dot_product(x,fp(1,:))
    arg_st(3,0) = dot_product(x,fs(0,:))
    arg_st(3,1) = dot_product(x,fs(1,:))
    x = 0.0_DP
    x(4) = 1.0_DP
    call z%SolSystem(x)
    arg_co(4,:) = x
    arg_pr(4,0) = dot_product(x,fp(0,:))
    arg_pr(4,1) = dot_product(x,fp(1,:))
    arg_st(4,0) = dot_product(x,fs(0,:))
    arg_st(4,1) = dot_product(x,fs(1,:))
    end subroutine similarity1_Coefficients
!=============================================================================
    subroutine similarity1_Boundary(arg_angle,arg_w,arg_d)
    implicit none
    integer,                intent(in) :: arg_angle
    real(DP),               intent(in) :: arg_w
    real(DP), dimension(8), intent(in) :: arg_d
    real(DP), dimension(0:1,8)         :: fp, fr, ft, fs
    real(DP), dimension(0:1,8)         :: gp, gr, gt, gs
    real(DP)                           :: fp0, fr0, ft0, fs0
    real(DP)                           :: fp1, fr1, ft1, fs1
    real(DP)                           :: gp0, gr0, gt0, gs0
    real(DP)                           :: gp1, gr1, gt1, gs1
    real(DP)                           :: t1, t2
    integer                            :: a1, a2
    a1 = 0
    a2 = arg_angle
    t1 = real(a1,DP) * PI / 180.0_DP
    t2 = real(a2,DP) * PI / 180.0_DP
!
    call similarity1_FUNCTIONS(t1,arg_w,fp,fr,ft,fs)
    call similarity1_FUNCTIONS(t2,arg_w,gp,gr,gt,gs)
!
    fs0 = dot_product(arg_d,fs(0,:))
    fs1 = dot_product(arg_d,fs(1,:))
```

```
    fp0 = dot_product(arg_d,fp(0,:))
    fp1 = dot_product(arg_d,fp(1,:))
    fr0 = dot_product(arg_d,fr(0,:))
    fr1 = dot_product(arg_d,fr(1,:))
    ft0 = dot_product(arg_d,ft(0,:))
    ft1 = dot_product(arg_d,ft(1,:))
!
    gs0 = dot_product(arg_d,gs(0,:))
    gs1 = dot_product(arg_d,gs(1,:))
    gp0 = dot_product(arg_d,gp(0,:))
    gp1 = dot_product(arg_d,gp(1,:))
    gr0 = dot_product(arg_d,gr(0,:))
    gr1 = dot_product(arg_d,gr(1,:))
    gt0 = dot_product(arg_d,gt(0,:))
    gt1 = dot_product(arg_d,gt(1,:))
!
    print 300
    print 100, 'st', a1, fs1, fs0, arg_w - 1.0_DP
    print 100, 'pr', a1, fp1, fp0, arg_w - 1.0_DP
    print 100, 'vr', a1, fr1, fr0, arg_w
    print 100, 'vt', a1, ft1, ft0, arg_w
    print 200
    print 100, 'st', a2, gs1, gs0, arg_w - 1.0_DP
    print 100, 'pr', a2, gp1, gp0, arg_w - 1.0_DP
    print 100, 'vr', a2, gr1, gr0, arg_w
    print 100, 'vt', a2, gt1, gt0, arg_w
    print 300
!
100 format (a, '(r,', i3, ') = [ (', f17.10, ') ln(r) + (' f17.10, ') ] r^(', f5.1, ')')
200 format ()
300 format (79('-'))
    end subroutine similarity1_Boundary
!=============================================================================
    subroutine similarity1_Cartesian(arg_w,arg_d,arg_p,arg_f)
    real(DP),                intent(in)  :: arg_w
    real(DP), dimension(8),  intent(in)  :: arg_d
    real(DP), dimension(2),  intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP)                             :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(arg_p,arg_p))
    t = angle(arg_p)
    call similarity1_Polar(arg_w,arg_d,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    arg_f = (/ pr, vx, vy, st /)
    end subroutine similarity1_Cartesian
!=============================================================================
    subroutine similarity1_Polar(arg_w,arg_d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP),               parameter   :: TOL = 1.0e-8_DP
    real(DP), dimension(8), intent(in)  :: arg_d
    real(DP),               intent(in)  :: arg_w, arg_r, arg_t
    real(DP),               intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    real(DP), dimension(0:1,8)          :: fp, fr, ft, fs
    real(DP)                            :: fp0, fr0, ft0, fs0
    real(DP)                            :: fp1, fr1, ft1, fs1
    real(DP)                            :: lnr
    if (arg_r < TOL) then
        arg_st = 0.0_DP
        arg_pr = 0.0_DP
        arg_vr = 0.0_DP
```

```
        arg_vt = 0.0_DP
    else
        call similarity1_FUNCTIONS(arg_t,arg_w,fp,fr,ft,fs)
        fs0 = dot_product(arg_d,fs(0,:))
        fs1 = dot_product(arg_d,fs(1,:))
        fp0 = dot_product(arg_d,fp(0,:))
        fp1 = dot_product(arg_d,fp(1,:))
        fr0 = dot_product(arg_d,fr(0,:))
        fr1 = dot_product(arg_d,fr(1,:))
        ft0 = dot_product(arg_d,ft(0,:))
        ft1 = dot_product(arg_d,ft(1,:))
        lnr = log(arg_r)
        arg_st = (fs1 * lnr + fs0) * arg_r**(arg_w - 1.0_DP)
        arg_pr = (fp1 * lnr + fp0) * arg_r**(arg_w - 1.0_DP)
        arg_vr = (fr1 * lnr + fr0) * arg_r**arg_w
        arg_vt = (ft1 * lnr + ft0) * arg_r**arg_w
    end if
    end subroutine similarity1_Polar
!=============================================================================
    subroutine similarity1_FUNCTIONS(arg_t,arg_w,arg_fp,arg_fr,arg_ft,arg_fs)
    implicit none
    real(DP),                   intent(in)  :: arg_t, arg_w
    real(DP), dimension(0:1,8), intent(out) :: arg_fp, arg_fr, arg_ft, arg_fs
    real(DP), dimension(12)                 :: fp0, fp1, fp2
    real(DP), dimension(12)                 :: fr0, fr1, fr2
    real(DP), dimension(12)                 :: ft0, ft1, ft2
    real(DP), dimension(12)                 :: fs0, fs1, fs2
    call generalSolution_Coeff_p(arg_t,arg_w,fp0,fp1,fp2)
    call generalSolution_Coeff_r(arg_t,arg_w,fr0,fr1,fr2)
    call generalSolution_Coeff_t(arg_t,arg_w,ft0,ft1,ft2)
    call generalSolution_Coeff_s(arg_t,arg_w,fs0,fs1,fs2)
    arg_fp(0,:) = fp0(5:12)
    arg_fp(1,:) = fp1(5:12)
    arg_fr(0,:) = fr0(5:12)
    arg_fr(1,:) = fr1(5:12)
    arg_ft(0,:) = ft0(5:12)
    arg_ft(1,:) = ft1(5:12)
    arg_fs(0,:) = fs0(5:12)
    arg_fs(1,:) = fs1(5:12)
    end subroutine similarity1_FUNCTIONS
!=============================================================================
end module mod_similarity1
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_similarity1Check
use mod_constants
use mod_similarity1
use mod_stokesEq2d
implicit none
private
public :: similarity1Check
real(DP)               :: sav_w
real(DP), dimension(8) :: sav_d
contains
!=============================================================================
    subroutine similarity1Check
    implicit none
    sav_w = 2.7_DP
    sav_d(1:4) = (/  1.37_DP,  5.76_DP, -6.32_DP, -7.98_DP /)
```

```
    sav_d(5:8) = (/ -7.62_DP,  3.17_DP,  4.51_DP,  2.89_DP /)
    call stokesEq2d_Cartesian(similarity1Check_CARTESIAN,'similarity1_Cartesian.txt')
    call stokesEq2d_Polar(similarity1Check_POLAR,'similarity1_Polar.txt')
    end subroutine similarity1Check
!=============================================================================
    subroutine similarity1Check_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    call similarity1_Cartesian(sav_w,sav_d,arg_p,arg_f)
    end subroutine similarity1Check_CARTESIAN
!=============================================================================
    subroutine similarity1Check_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_pr, arg_vr, arg_vt, arg_st
    call similarity1_Polar(sav_w,sav_d,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end subroutine similarity1Check_POLAR
!=============================================================================
end module mod_similarity1Check
!#############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outerSeries
use mod_constants
use mod_utilities
use mod_similarity1
use mod_outer0
use mod_outer1
implicit none
private
type, public :: typ_outerSeries
    private
    integer                                 :: angle = 90
    integer                                 :: k = 0
    integer                                 :: n = 25
    real(DP)                                :: s = 12.3_DP
    real(DP), dimension(:,:,:), allocatable :: co
    real(DP), dimension(:,:,:), allocatable :: pr
    real(DP), dimension(:,:,:), allocatable :: st
    real(DP), dimension(:,:),   allocatable :: a
    real(DP), dimension(:),     allocatable :: w
contains
    procedure :: SetAngle      => outerSeries_SetAngle
    procedure :: SetSolution   => outerSeries_SetSolution
    procedure :: SetOrder      => outerSeries_SetOrder
    procedure :: SetSlip       => outerSeries_SetSlip
    procedure :: Coefficients  => outerSeries_Coefficients
    procedure :: errorPressure => outerSeries_errorPressure
    procedure :: errorStress   => outerSeries_errorStress
    procedure :: Boundary      => outerSeries_Boundary
    procedure :: Cartesian     => outerSeries_Cartesian
    procedure :: Polar         => outerSeries_Polar
end type typ_outerSeries
type(typ_outer0)               :: sav_outer0
type(typ_outer1), dimension(3) :: sav_outer1
contains
!=============================================================================
    subroutine outerSeries_SetAngle(this,arg_angle)
    implicit none
```

```
    class(typ_outerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_angle
    this%angle = clip(arg_angle,45,180)
    end subroutine outerSeries_Setangle
!================================================================================
    subroutine outerSeries_SetSolution(this,arg_k)
    implicit none
    class(typ_outerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_k
    this%k = clip(arg_k,0,1)
    end subroutine outerSeries_SetSolution
!================================================================================
    subroutine outerSeries_SetOrder(this,arg_n)
    implicit none
    class(typ_outerSeries), intent(inout) :: this
    integer,                intent(in)    :: arg_n
    integer                               :: n
    this%n = max(arg_n,0)
    n = max(this%n,1)
    call outerSeries_DEALLOCATE(this)
    allocate(this%co(0:n,4,8))
    allocate(this%pr(0:n,4,0:1))
    allocate(this%st(0:n,4,0:1))
    allocate(this%a(0:n,4))
    allocate(this%w(0:n))
    call sav_outer0%SetAngle(this%angle)
    call sav_outer1(1)%SetAngle(this%angle)
    call sav_outer1(2)%SetAngle(this%angle)
    call sav_outer1(3)%SetAngle(this%angle)
    call sav_outer1(1)%SetSolution(1)
    call sav_outer1(2)%SetSolution(2)
    call sav_outer1(3)%SetSolution(3)
    end subroutine outerSeries_SetOrder
!================================================================================
    subroutine outerSeries_SetSlip(this,arg_s)
    class(typ_outerSeries), intent(inout) :: this
    real(DP),               intent(in)    :: arg_s
    integer                               :: n
    this%s = abs(arg_s)
    call outerSeries_RECURSION(this)
    end subroutine outerSeries_SetSlip
!================================================================================
    subroutine outerSeries_Coefficients(this,arg_filename)
    class(typ_outerSeries), intent(in) :: this
    character(len=*),       intent(in) :: arg_filename
    integer                            :: n
    open(unit = 20, file = arg_filename)
    do n = 0, this%n
        write(20,100) n, this%a(n,:)
    end do
    close(20)
100 format('n = ', i3, 4x, 4(1x, e17.10))
    end subroutine outerSeries_Coefficients
!================================================================================
    function outerSeries_errorPressure(this,arg_r) result(arg_f)
    implicit none
    class(typ_outerSeries), intent(in) :: this
    real(DP),               intent(in) :: arg_r
    real(DP)                           :: arg_f
    integer                            :: i
```

```
    arg_f = 0.0_DP
    do i = 1, 4
        arg_f = arg_f + this%a(this%n,i) * arg_r**this%w(this%n) / arg_r &
                * (this%pr(this%n,i,1) * log(arg_r) + this%pr(this%n,i,0))
    end do
    end function outerSeries_errorPressure
!==============================================================================
    function outerSeries_errorStress(this,arg_r) result(arg_f)
    implicit none
    class(typ_outerSeries), intent(in) :: this
    real(DP),               intent(in) :: arg_r
    real(DP)                           :: arg_f
    integer                            :: i
    arg_f = 0.0_DP
    do i = 1, 4
        arg_f = arg_f + this%a(this%n,i) * arg_r**this%w(this%n) / arg_r &
                * (this%st(this%n,i,1) * log(arg_r) + this%st(this%n,i,0))
    end do
    end function outerSeries_errorStress
!==============================================================================
    subroutine outerSeries_Boundary(this,arg_filename)
    implicit none
    class(typ_outerSeries), intent(in) :: this
    character(len=*),       intent(in) :: arg_filename
    integer                            :: i, iMax
    real(DP)                           :: r, rMax, rMin
    real(DP)                           :: pr, vr, vt, st
    real(DP)                           :: e1, e2, f1, f2
    real(DP)                           :: error1, error2, distance, cutoff
    cutoff = 1.0e-2_DP
    rMin = 1.0e1_DP
    rMax = 1.0e4_DP
    iMax = 15
    error1 = 0.0_DP
    error2 = 0.0_DP
    distance = 0.0_DP
    open(unit = 20, file = arg_filename)
    do i = 0, iMax
        r = exp(log(rMin) + log(rMax / rMin) * real(i,DP) / real(iMax,DP))
        call outerSeries_Polar(this,r,0.0_DP,pr,vr,vt,st)
        f1 = outerSeries_errorPressure(this,r)
        f2 = outerSeries_errorStress(this,r)
        e1 = pr + vt
        e2 = st - vr / this%s
        write(20,100) r, pr, vt, e1, f1, st, vr / this%s, e2, f2
        if (abs(e1 - f1) > error1) then
            error1 = abs(e1 - f1)
        end if
        if (abs(e2 - f2) > error2) then
            error2 = abs(e2 - f2)
        end if
        if (max(abs(e1),abs(e2)) > cutoff) then
            distance = r
        end if
    end do
    close(20)
    print 200, error1, error2, distance

100 format(e12.5, 2(8x, e15.8, 1x, e15.8, 8x, e12.5, 1x, e12.5))
200 format('outerSeries: Solution' / &
```

```
          5x, 'Error1   = ', e12.5 / 5x, 'Error2   = ', e12.5 / &
          5x, 'Distance = ', e12.5 /)
    end subroutine outerSeries_Boundary
!===============================================================================
    subroutine outerSeries_Cartesian(this,arg_p,arg_f)
    class(typ_outerSeries),   intent(in)  :: this
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    real(DP), dimension(0:3)              :: f
    integer                               :: i, n
    call sav_outer0%Cartesian(arg_p,f)
    arg_f = this%a(0,1) * f
    if (this%n > 0) then
        do i = 1, 3
            call sav_outer1(i)%Cartesian(arg_p,f)
            arg_f = arg_f + this%a(1,i) * f
        end do
    end if
    do n = 2, this%n
        do i = 1, 4
            call similarity1_Cartesian(this%w(n),this%co(n,i,:),arg_p,f)
            arg_f = arg_f + this%a(n,i) * f
        end do
    end do
    end subroutine outerSeries_Cartesian
!===============================================================================
    subroutine outerSeries_Polar(this,arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    class(typ_outerSeries), intent(in)  :: this
    real(DP),               intent(in)  :: arg_r, arg_t
    real(DP),               intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    real(DP)                            :: st, pr, vr, vt, cs, cp
    integer                             :: i, n
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    arg_st = 0.0_DP
    call sav_outer0%Polar(arg_r,arg_t,pr,vr,vt,st)
    arg_pr = this%a(0,1) * pr
    arg_vr = this%a(0,1) * vr
    arg_vt = this%a(0,1) * vt
    arg_st = this%a(0,1) * st
    if (this%n > 0) then
        do i = 1, 3
            call sav_outer1(i)%Polar(arg_r,arg_t,pr,vr,vt,st)
            arg_pr = arg_pr + this%a(1,i) * pr
            arg_vr = arg_vr + this%a(1,i) * vr
            arg_vt = arg_vt + this%a(1,i) * vt
            arg_st = arg_st + this%a(1,i) * st
        end do
    end if
    do n = 2, this%n
        do i = 1, 4
            call similarity1_Polar(this%w(n),this%co(n,i,:),arg_r,arg_t,pr,vr,vt,st)
            arg_pr = arg_pr + this%a(n,i) * pr
            arg_vr = arg_vr + this%a(n,i) * vr
            arg_vt = arg_vt + this%a(n,i) * vt
            arg_st = arg_st + this%a(n,i) * st
        end do
    end do
```

```
    end subroutine outerSeries_Polar
!===============================================================================
    subroutine outerSeries_RECURSION(this)
    implicit none
    class(typ_outerSeries), intent(inout) :: this
    real(DP)                              :: pr0, vr0, vt0, st0
    real(DP), dimension(3,0:1)            :: pr1, vr1, vt1, st1
    real(DP), dimension(0:1)              :: pr, st
    integer                               :: i, j, n
!
    this%w(0) = 0.0_DP
    call sav_outer0%Boundary(pr0,vr0,vt0,st0)
    if (this%k == 0) then
        this%a(0,1) = 1.0_DP
    else
        this%a(0,1) = 0.0_DP
    end if
    this%a(0,2) = 0.0_DP
    this%a(0,3) = 0.0_DP
    this%a(0,4) = 0.0_DP
    this%pr(0,1,:) = (/ pr0, 0.0_DP /)
    this%pr(0,2,:) = 0.0_DP
    this%pr(0,3,:) = 0.0_DP
    this%pr(0,4,:) = 0.0_DP
    this%st(0,1,:) = (/ st0, 0.0_DP /)
    this%st(0,2,:) = 0.0_DP
    this%st(0,3,:) = 0.0_DP
    this%st(0,4,:) = 0.0_DP
!
    this%w(1) = -1.0_DP
    call sav_outer1(1)%Boundary(pr1(1,:),vr1(1,:),vt1(1,:),st1(1,:))
    call sav_outer1(2)%Boundary(pr1(2,:),vr1(2,:),vt1(2,:),st1(2,:))
    call sav_outer1(3)%Boundary(pr1(3,:),vr1(3,:),vt1(3,:),st1(3,:))
    this%a(1,1) = -this%a(0,1) * this%pr(0,1,0)
    this%a(1,2) =  this%a(0,1) * this%st(0,1,0) * this%s
    if (this%k == 0) then
        this%a(1,3) =  0.0_DP
    else
        this%a(1,3) =  1.0_DP
    end if
    this%a(1,4) =  0.0_DP
    this%pr(1,1,:) = pr1(1,:)
    this%pr(1,2,:) = pr1(2,:)
    this%pr(1,3,:) = pr1(3,:)
    this%pr(1,4,:) = 0.0_DP
    this%st(1,1,:) = st1(1,:)
    this%st(1,2,:) = st1(2,:)
    this%st(1,3,:) = st1(3,:)
    this%st(1,4,:) = 0.0_DP
!
    do n = 2, this%n
        this%w(n) = -real(n,DP)
        call similarity1_Coefficients(this%angle,this%w(n),this%co(n,:,:),this%pr(n,:,:),this%st(n,:,:))
        pr = 0.0_DP
        st = 0.0_DP
        do i = 1, 4
            do j = 0, 1
                pr(j) = pr(j) + this%a(n-1,i) * this%pr(n-1,i,j)
                st(j) = st(j) + this%a(n-1,i) * this%st(n-1,i,j)
            end do
```

```
        end do
        this%a(n,1) = -pr(0)
        this%a(n,2) = -pr(1)
        this%a(n,3) =  st(0) * this%s
        this%a(n,4) =  st(1) * this%s
    end do
100 format('Order: ', i3)
    end subroutine outerSeries_RECURSION
!=======================================================================
    subroutine outerSeries_DEALLOCATE(this)
    implicit none
    class(typ_outerSeries), intent(inout) :: this
    if (allocated(this%co)) then
        deallocate(this%co)
    end if
    if (allocated(this%pr)) then
        deallocate(this%pr)
    end if
    if (allocated(this%st)) then
        deallocate(this%st)
    end if
    if (allocated(this%a)) then
        deallocate(this%a)
    end if
    if (allocated(this%w)) then
        deallocate(this%w)
    end if
    end subroutine outerSeries_DEALLOCATE
!=======================================================================
end module mod_outerSeries
!#######################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_outerSeriesCheck
use mod_constants
use mod_outerSeries
use mod_stokesEq2d
implicit none
private
public :: outerSeriesCheck
type(typ_outerSeries) :: sav_a
contains
!=======================================================================
    subroutine outerSeriesCheck
    implicit none
    integer                   :: angle, i, k, m, n
    real(DP), dimension(10)   :: r, e
    real(DP), dimension(7,13) :: c
    real(DP)                  :: s
    print*, 'outerSeriesCheck'
    print*, 'angle ?'
    read *,  angle
    print*, 'k ?'
    read *,  k
    print*, 'n ?'
    read *,  n
    print*, 's ?'
    read *,  s
    call sav_a%SetAngle(angle)
    call sav_a%SetSolution(k)
```

```
    call sav_a%SetOrder(n)
    call sav_a%SetSlip(s)
    call sav_a%Coefficients('outerSeries_Coefficients.txt')
    call sav_a%Boundary('outerSeries_Boundary' // char(48 + k) // '.txt')
    call stokesEq2d_Cartesian(outerSeriesCheck_CARTESIAN,'outerSeries_Cartesian' // char(48 + k) // '.txt')
    call stokesEq2d_Polar    (outerSeriesCheck_POLAR    ,'outerSeries_Polar'     // char(48 + k) // '.txt')
    r( 1) = 1.0e1_DP
    r( 2) = 2.0e1_DP
    r( 3) = 5.0e1_DP
    r( 4) = 1.0e2_DP
    r( 5) = 2.0e2_DP
    r( 6) = 5.0e2_DP
    r( 7) = 1.0e3_DP
    r( 8) = 2.0e3_DP
    r( 9) = 5.0e3_DP
    r(10) = 1.0e4_DP
    open(unit = 20, file = 'outerSeries_Errors' // char(48 + k) // '.txt')
    write(20,100) r
    do m = 0, n
        call sav_a%SetOrder(m)
        call sav_a%SetSlip(s)
        do i = 1, 10
            e(i) = sav_a%errorPressure(r(i))
        end do
        write(20,140) m, e
    end do
    write(20,120) r
    do m = 0, n
        call sav_a%SetOrder(m)
        call sav_a%SetSlip(s)
        do i = 1, 10
            e(i) = sav_a%errorStress(r(i))
        end do
        write(20,140) m, e
    end do
    close(20)
100 format(   'Series errors in pressure' // '  n', 10(1x, e9.2) / '---', 10(1x, 9('-')))
120 format(// 'Series errors in stress'   // '  n', 10(1x, e9.2) / '---', 10(1x, 9('-')))
140 format(i3, 110(1x, e9.2))
    end subroutine outerSeriesCheck
!===============================================================================
    subroutine outerSeriesCheck_CARTESIAN(arg_p,arg_f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: arg_p
    real(DP), dimension(0:3), intent(out) :: arg_f
    call sav_a%Cartesian(arg_p,arg_f)
    end subroutine outerSeriesCheck_CARTESIAN
!===============================================================================
    subroutine outerSeriesCheck_POLAR(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_st, arg_pr, arg_vr, arg_vt
    call sav_a%Polar(arg_r,arg_t,arg_pr,arg_vr,arg_vt,arg_st)
    end subroutine outerSeriesCheck_POLAR
!===============================================================================
end module mod_outerSeriesCheck
!###############################################################################
! Copyright 2013 by Ludwig C. Nitsche
!-----------------------------------
module mod_check
```

```
use mod_outer1Check
use mod_innerBuildCheck
use mod_outerBuildCheck
use mod_innerSeriesCheck
use mod_outerSeriesCheck
use mod_similarityCheck
use mod_similarity1Check
implicit none
private
public :: check
contains
!==============================================================================
    subroutine check
    call outer1Check
    call innerBuildCheck
    call outerBuildCheck
    call innerSeriesCheck
    call outerSeriesCheck
    call similarityCheck
    call similarity1Check
    end subroutine check
!==============================================================================
end module mod_check
!##############################################################################
```