Supplementary Material†

1. Consideration of slip along the porous boundary

We initially prescribed the no-slip condition (1.10) because slip along porous walls has been found to have a weak effect upon similarity solutions (Chellam & Liu 2006) and because arguments have been made to neglect it (Schmitz & Prat 1995; Pak *et al.* 2008). On the other hand, there is some evidence that slip can mitigate the effects of concentration polarization in ultrafiltration (Singh & Laurence 1979*a*,*b*; Yeh & Cheng 1999; Yucel & Turkoglu 2005). Thus, for completeness we shall now revisit the question of slip along the porous wall.

The classic Beavers-Joseph slip condition (Beavers & Joseph 1967; Saffman 1971; Beavers *et al.* 1974; Neale & Nader 1974; Nield 1983; Singh & Laurence 1979*a*,*b*; Chellam *et al.* 1992, 1995; Yeh & Cheng 1999; Nield 2009; Chen *et al.* 2010; Borsi *et al.* 2011) was originally phrased in terms of the normal derivative of the tangential velocity, whereas we shall add the transposed gradient term to make the slip velocity proportional to the wall shear stress — as appears in Jones (1973) and Nield (1983).

$$V_t \;=\; \frac{\sqrt{\kappa}}{\alpha}\left[\frac{\partial V_t}{\partial n}\right] \qquad \text{(Original)} \qquad \text{(I)}$$

$$V_t \;=\; \frac{\sqrt{\kappa}}{\alpha}\left[\frac{\partial V_t}{\partial n} + \frac{\partial V_n}{\partial t}\right] \qquad \text{(Modified)} \qquad \text{(II)}$$

The difference is negligible at macroscopic lengthscales, but becomes significant when the problem is scaled to resolve the (weakly singular) fine structure at the origin. Specifically, smallness of the normal velocity itself need not rule out appreciable magnitude of its tangential derivative. (Indeed, the unexpectedly drastic effect of the original slip condition upon preliminary numerical solutions alerted us to the need for an additional term in the shear stress, for which we then sought out the aforementioned precedents in the literature.)

Recast in dimensionless form, the slip condition (II) becomes

$$\tau_{\theta r}(r,0) \;=\; \sigma^{-1} v_r(r,0), \qquad \tau_{\theta r} \;=\; \frac{1}{r}\left[\frac{\partial v_r}{\partial \theta} - v_\theta\right] + \frac{\partial v_\theta}{\partial r} \qquad \text{(III)}$$

The dimensionless slip length σ is defined by

$$1 \;\ll\; \sigma \;\stackrel{\text{def}}{=}\; \frac{\ell}{\alpha\sqrt{\kappa}} \;\ll\; \frac{\ell L}{\kappa} \qquad \text{(IV)}$$

with L a characteristic macroscopic lengthscale. One can expect σ to be very large for two reasons. First, the dimensionless Beavers-Joseph parameter α is typically on the order of 0.1 (Beavers & Joseph 1967; Beavers *et al.* 1974). Second, $\sqrt{\kappa}$ appearing in the denominator is on the order of the pore size, which is vastly exceeded by the membrane thickness ℓ appearing in the numerator. Thus,

† Here equations are numbered with Roman numerals. Arabic equation numbers refer to the printed paper and literature citations refer to its bibliography.

vanishing shear stress (in the limit as $\sigma \to \infty$)

$$\tau_{\theta r}(r,0) \;=\; 0 \qquad \text{(V)}$$

should be a good approximation for the slip condition (III), at least for moderate r. However, even if σ is very large the slip length will still appear negligibly small on the macroscopic scale; cf. equation (IV). Thus, the no-slip condition (1.10) should become increasingly accurate as $r \to \infty$.

The Stokes flow problem (1.6) – (1.13) is unusual because — as was seen in the paper — it admits of a solution that satisfies both the no-slip condition (1.10) and the no-shear condition (V). This means that the slip condition (III) is also satisfied *a fortiori* for any value of σ. The mathematical degeneracy that allows such an outcome leaves an open question for further investigation in the future. It does seem safe to say that the question of slip has no significant bearing on the flow singularity at the junction of impermeable and porous walls.

2. Inapplicability of the Mellin transform

Although the Mellin transform yields solutions for a variety of Stokes flow problems posed in angular wedges (Moffatt 1964*b*; Moffatt & Duffy 1980), such an attack cannot be mounted on the problem (1.6) – (1.13). The Mellin transform of the stream function is defined by

$$\overline{\psi}(p,\theta) \;=\; \int_0^\infty r^{p-1}\psi(r,\theta)\,\mathrm{d}r \qquad \text{(I)}$$

Suitable integration by parts of relevant terms allows the Stokes equations

$$\left(\frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial^2}{\partial\theta^2}\right)^2 \psi \;=\; 0 \qquad \text{Momentum and continuity}$$

$$\Rightarrow\quad \frac{\mathrm{d}^4\overline{\psi}}{\mathrm{d}\theta^4} + \Big[(p+2)^2 + p^2\Big]\frac{\mathrm{d}^2\overline{\psi}}{\mathrm{d}\theta^2} + p^2(p+2)^2\overline{\psi} \;=\; 0 \qquad \text{(II)}$$

and three of the boundary conditions

$$\frac{\partial\psi}{\partial\theta}(r,\pi) \;=\; 0 \quad\Rightarrow\quad \frac{\mathrm{d}\overline{\psi}}{\mathrm{d}\theta}(p,\pi) \;=\; 0 \qquad \text{No slip along solid wall} \qquad \text{(III)}$$

$$\frac{\partial\psi}{\partial\theta}(r,0) \;=\; 0 \quad\Rightarrow\quad \frac{\mathrm{d}\overline{\psi}}{\mathrm{d}\theta}(p,0) \;=\; 0 \qquad \text{No slip along porous wall} \qquad \text{(IV)}$$

$$\frac{\partial\psi}{\partial r}(r,\pi) \;=\; 0 \quad\Rightarrow\quad \overline{\psi}(p,\pi) \;=\; 0 \qquad \text{No seepage along solid wall} \qquad \text{(V)}$$

to be transformed from (r,θ) space to (p,θ) space in a straightforward fashion.

The difficulty arises because the seepage condition (1.11) equates pressure and velocity, which are quantities of different "orders." [For example, velocity is of order $r^{\lambda-1}$ whereas pressure is of order $r^{\lambda-2}$ in the similarity solution (1.1). More generally, pressure appears at the order of the gradient of velocity in the Stokes equations.] Expressed in terms of the stream function, the left-hand side of equation (1.11) effectively has two powers of r in the denominator while the

right-hand side has three.

$$\frac{\partial^2\psi}{\partial r^2}(r,0) \;=\; -\frac{1}{r}\frac{\partial}{\partial\theta}\left(\frac{\partial^2\psi}{\partial r^2}+\frac{1}{r}\frac{\partial\psi}{\partial r}+\frac{1}{r^2}\frac{\partial^2\psi}{\partial\theta^2}\right) \qquad \text{Seepage along porous wall}$$

$$\Rightarrow\quad p(p+1)\overline{\psi}(p,0) \;=\; -\frac{\mathrm{d}^3\overline{\psi}}{\mathrm{d}\theta^3}(p-1,0) \;-\; (p-1)^2\frac{\mathrm{d}\overline{\psi}}{\mathrm{d}\theta}(p-1,0) \qquad \text{(VI)}$$

Correspondingly, the Mellin-transformed version of the seepage condition has a shift in the transformed variable (from p to $p-1$) between the left-and right-hand sides. This feature spoils prospects for a direct solution in (p,θ) space.

3. A test problem for validating the numerical scheme

As a test problem for the numerical scheme, consider the leading outer solution $p^{\{1\}}$, $v_r^{\{1\}}$, $v_\theta^{\{1\}}$ from equations (1.14) – (1.16) on the finite domain from figure 2.

$$\nabla^2\boldsymbol{v} = \nabla p, \qquad 0<r<30,\quad 0<\theta<\pi \qquad \text{Momentum} \qquad \text{(I)}$$

$$\frac{\partial(rv_r)}{\partial r}+\frac{\partial v_\theta}{\partial\theta} = 0, \qquad 0<r<30,\quad 0<\theta<\pi \qquad \text{Continuity} \qquad \text{(II)}$$

$$v_r(r,\pi) = 0, \qquad 0<r<30 \qquad \text{No slip} \qquad \text{(III)}$$

$$v_\theta(r,\pi) = 0, \qquad 0<r<30 \qquad \text{Impermeability} \qquad \text{(IV)}$$

$$v_r(r,0) = 0 \qquad 0<r<30 \qquad \text{No slip} \qquad \text{(V)}$$

$$v_\theta(r,0) = -1, \qquad 0<r<30 \qquad \text{Uniform seepage} \qquad \text{(VI)}$$

$$p(30,\theta) = p^{\{1\}}(30,\theta), \quad 0<\theta<\pi \qquad \text{Far-field pressure} \qquad \text{(VII)}$$

$$v_\theta(30,\theta) = v_\theta^{\{1\}}(30,\theta), \quad 0<\theta<\pi \qquad \text{Far-field velocity} \qquad \text{(VIII)}$$

We focus on the effectiveness of mesh refinement near the origin. The discretization parameters listed in figure 3 gave 376 boundary segments and 1500 boundary points in all. Along the impermeable wall and permeable membrane we imposed velocity boundary conditions, whereas along the semicircular arc we impose p and v_θ. It is not worthwhile to show graphs, because the numerical results reproduced the analytical velocity and pressure fields easily within the plotted line width down to $r=10^{-2}$. Essentially identical results were obtained with or without point sources augmenting the point forces (Stokeslets) in the basis — with one exception: when using only Stokeslets, a localized sawtooth instability afflicted the calculated pressures near the two corners at $\theta=0,\pi$ ($r=30$). This might have been expected from the linear variation of the lineal density of singularity strength along the boundary segments in equation (2.2); cf. page 162 in the book by Pozrikidis (1992). Not yet having ascertained why the point sources almost entirely eliminate the (localized) numerical instability, we will here be satisfied to benefit from their inclusion in the subsequent calculations.

4. Fortran 2003 library that is referred to in Appendix D

The following Fortran 2003 library should be copied into a text editor window and saved under the filename "stokes2d.f95" in the same directory as the calling program (e.g., the sample program from Appendix D).

```
!###############################################################################
! Copyright 2011 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 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_utilities
use mod_constants
implicit none
private
public :: angle
public :: numChar
public :: PolarToCartesian
public :: CartesianToPolar
public :: LatexSciNotation
public :: FiniteDifference
contains
!===============================================================================
    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 2011 by Ludwig C. Nitsche
!---------------------------------
module mod_tridiag
use mod_constants
implicit none
type, public :: typ_tridiag
    private
    integer                                 :: n
    real(DP), dimension(:,:), allocatable :: a
contains
    procedure :: setMatrix => tridiag_setMatrix
    procedure :: getMatrix => tridiag_getMatrix
    procedure :: solSystem => tridiag_solSystem
end type typ_tridiag
private :: tridiag_setMatrix
private :: tridiag_getMatrix
private :: tridiag_solSystem
public  :: tridiag_mulVector
contains
!================================================================================
    subroutine tridiag_getMatrix(this,arg_a)
    class(typ_tridiag),         intent(in)  :: this
    real(DP), dimension(:,-1:), intent(out) :: arg_a
    integer                                 :: i
    if (size(arg_a,1) /= size(this%a,1) .or. size(arg_a,2) /= 3) then
        print*, 'Error in {tridiag_getMatrix}'
        stop
    end if
    arg_a(1,-1) = 0.0_DP
    arg_a(1, 0) = this%a(1,0)
    arg_a(1, 1) = this%a(1,1)
    do i = 2, this%n - 1
```

```
        arg_a(i,-1) = this%a(i,-1) * this%a(i-1,0)
        arg_a(i, 0) = this%a(i,-1) * this%a(i-1,1) + this%a(i,0)
        arg_a(i, 1) = this%a(i,1)
    end do
    arg_a(this%n,-1) = this%a(this%n,-1) * this%a(this%n-1,0)
    arg_a(this%n, 0) = this%a(this%n,-1) * this%a(this%n-1,1) + this%a(this%n,0)
    arg_a(this%n, 1) = 0.0_DP
    end subroutine tridiag_getMatrix
!================================================================================
    subroutine tridiag_setMatrix(this,arg_a)
    class(typ_tridiag),       intent(inout) :: this
    real(DP), dimension(:,:), intent(in)    :: arg_a
    integer                                 :: i
    if (size(arg_a,2) /= 3) then
        print*, 'Error in {tridiag_setMatrix}'
        stop
    end if
    this%n = size(arg_a,1)
    if (allocated(this%a)) then
        deallocate(this%a)
    end if
    allocate(this%a(this%n,-1:1))
    this%a = arg_a
    do i = 2, this%n
        this%a(i,-1) = this%a(i,-1) / this%a(i-1,0)
        this%a(i,0) = this%a(i,0) - this%a(i,-1) * this%a(i-1,1)
    end do
    end subroutine tridiag_setMatrix
!================================================================================
    subroutine tridiag_solSystem(this,arg_v)
    class(typ_tridiag),     intent(in)    :: this
    real(DP), dimension(:), intent(inout) :: arg_v
    integer                               :: i
    if (size(arg_v) /= this%n) then
        print*, 'Error in {matrix_solSystem}'
        print*, '[arg_v] of incorrect length'
        stop
    end if
    do i = 1, this%n
        arg_v(i) = arg_v(i) - this%a(i,-1) * arg_v(i-1)
    end do
    arg_v(this%n) = arg_v(this%n) / this%a(this%n,0)
    do i = this%n - 1, 1, -1
        arg_v(i) = (arg_v(i) - this%a(i,1) * arg_v(i+1)) / this%a(i,0)
    end do
    end subroutine tridiag_solSystem
!================================================================================
    subroutine tridiag_mulVector(arg_a,arg_x,arg_y)
    real(DP), dimension(:,-1:), intent(in)  :: arg_a
    real(DP), dimension(:),     intent(in)  :: arg_x
    real(DP), dimension(:),     intent(out) :: arg_y
    integer                                 :: i, j, n
    if (size(arg_x) /= size(arg_a,1)) then
        print*, 'Error #1 in {matrix_solSystem}'
        print*, '[arg_x] of incorrect length'
        stop
    end if
    if (size(arg_y) /= size(arg_a,1)) then
        print*, 'Error #2 in {matrix_solSystem}'
```

```
        print*, '[arg_y] of incorrect length'
        stop
    end if
    if (size(arg_a,2) /= 3) then
        print*, 'Error #3 in {matrix_solSystem}'
        print*, '[arg_a] of incorrect width'
        stop
    end if
    n = size(arg_a,1)
    arg_y(1) = arg_a(1, 0) * arg_x(1)   + arg_a(1,1) * arg_x(2)
    arg_y(n) = arg_a(n,-1) * arg_x(n-1) + arg_a(n,0) * arg_x(n)
    do i = 2, n - 1
        arg_y(i) = 0.0_DP
        do j = -1, 1
            arg_y(i) = arg_y(i) + arg_a(i,j) * arg_x(i+j)
        end do
    end do
    end subroutine tridiag_mulVector
!=======================================================================
end module mod_tridiag
!#######################################################################################
! Copyright 2011 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 = 6
    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 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_interp
use mod_constants
implicit none
private
public :: interp_monotonic
public :: interp_interval
public :: interp_coeff
public :: interp_phi1
public :: interp_phi2
public :: interp_psi1
public :: interp_psi2
contains
!=======================================================================
    function interp_monotonic(arg_a) result(arg_answer)
    real(DP), dimension(:), intent(in) :: arg_a
    character(len=1)                   :: arg_answer
    integer                            :: i, n
    n = size(arg_a)
    arg_answer = 'y'
    do i = 3, n
        if ((arg_a(i) - arg_a(i-1)) * (arg_a(i-1) - arg_a(i-2)) <= 0.0_DP) then
            arg_answer = 'n'
            exit
        end if
    end do
```

```
    if (arg_answer == 'y') then
        if (arg_a(n) > arg_a(1)) then
            arg_answer = 'i'
        else
            arg_answer = 'd'
        end if
    end if
    end function interp_monotonic
!===============================================================================
    function interp_interval(arg_a,arg_x) result(arg_i)
    real(DP), dimension(:), intent(in) :: arg_a
    real(DP),               intent(in) :: arg_x
    integer                            :: arg_i
    integer                            :: i, n
    n = size(arg_a)
    if (arg_x <= arg_a(1) .and. arg_a(1) < arg_a(2)) then
        arg_i = 2
    else if (arg_x >= arg_a(n) .and. arg_a(n) > arg_a(n-1)) then
        arg_i = n
    else if (arg_x >= arg_a(1) .and. arg_a(1) > arg_a(2)) then
        arg_i = 2
    else if (arg_x <= arg_a(n) .and. arg_a(n) < arg_a(n-1)) then
        arg_i = n
    else
        do i = 2, n
            if ((arg_a(i-1) <= arg_x .and. arg_x <= arg_a(i)) .or. &
                (arg_a(i-1) >= arg_x .and. arg_x >= arg_a(i))) then
                arg_i = i
                exit
            end if
        end do
    end if
    end function interp_interval
!===============================================================================
    function interp_coeff(arg_x0,arg_x1,arg_x) result(arg_w)
    implicit none
    real(DP), intent(in)     :: arg_x, arg_x0, arg_x1
    real(DP), dimension(0:1) :: arg_w
    arg_w(0) = (arg_x0 - arg_x) / (arg_x0 - arg_x1)
    arg_w(1) = (arg_x1 - arg_x) / (arg_x1 - arg_x0)
    end function interp_coeff
!===============================================================================
    function interp_phi1(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = 1.0_DP - 3.0_DP * arg_u**2 + 2.0_DP * arg_u**3
        case(1)
            arg_f = -6.0_DP * arg_u + 6.0_DP * arg_u**2
        case(2)
            arg_f = -6.0_DP + 12.0_DP * arg_u
        case default
            arg_f = 0.0_DP
    end select
    end function interp_phi1
!===============================================================================
```

```
    function interp_phi2(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = 3.0_DP * arg_u**2 - 2.0_DP * arg_u**3
        case(1)
            arg_f = 6.0_DP * arg_u - 6.0_DP * arg_u**2
        case(2)
            arg_f = 6.0_DP - 12.0_DP * arg_u
        case default
            arg_f = 0.0_DP
    end select
    end function interp_phi2
!=============================================================================
    function interp_psi1(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = arg_u - 2.0_DP * arg_u**2 + arg_u**3
        case(1)
            arg_f = 1.0_DP - 4.0_DP * arg_u + 3.0_DP * arg_u**2
        case(2)
            arg_f = -4.0_DP + 6.0_DP * arg_u
        case default
            arg_f = 0.0_DP
    end select
    end function interp_psi1
!=============================================================================
    function interp_psi2(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = -arg_u**2 + arg_u**3
        case(1)
            arg_f = -2.0_DP * arg_u + 3.0_DP * arg_u**2
        case(2)
            arg_f = -2.0_DP + 6.0_DP * arg_u
        case default
            arg_f = 0.0_DP
    end select
    end function interp_psi2
!=============================================================================
end module mod_interp
!#############################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_interp1
use mod_constants
use mod_utilities
use mod_interp
```

```
use mod_tridiag
implicit none
type, public :: typ_interp1
    private
    integer                              :: n  ! number of points
    real(DP), dimension(:), allocatable :: x  ! array  of abcissas
    real(DP), dimension(:), allocatable :: f  ! array  of ordinates
    real(DP), dimension(:), allocatable :: s1 ! array  of slopes
    real(DP), dimension(:), allocatable :: s2 ! array  of slopes
contains
    procedure :: setData  => interp1_setData
    procedure :: tabulate => interp1_tabulate
    procedure :: linear   => interp1_linear
    procedure :: spline1  => interp1_spline1
    procedure :: spline2  => interp1_spline2
end type typ_interp1
private :: interp1_setData
private :: interp1_tabulate
private :: interp1_linear
private :: interp1_spline1
private :: interp1_spline2
contains
!=======================================================================
    subroutine interp1_setData(this,arg_x,arg_f)
    implicit none
    type (typ_tridiag)                      :: z
    class(typ_interp1),       intent(inout) :: this
    real(DP), dimension(:),   intent(in)    :: arg_x, arg_f
    if (allocated(this%x)) then
        deallocate(this%x)
    end if
    if (allocated(this%f)) then
        deallocate(this%f)
    end if
    if (allocated(this%s1)) then
        deallocate(this%s1)
    end if
    if (allocated(this%s2)) then
        deallocate(this%s2)
    end if
    this%n = min(size(arg_x),size(arg_f))
    allocate(this%x(this%n))
    allocate(this%f(this%n))
    allocate(this%s1(this%n))
    allocate(this%s2(this%n))
    this%x  = arg_x(1:this%n)
    this%f  = arg_f(1:this%n)
    this%s1 = 0.0_DP
    this%s2 = 0.0_DP
    if (interp_monotonic(this%x) == 'n') then
        print*, 'Error in {interp1_setData}'
        print*, 'Non-monotonic abcissa'
        stop
    end if
    call interp1_splineSlopes(this)
    end subroutine interp1_setData
!=======================================================================
    subroutine interp1_splineSlopes(this)
    implicit none
```

```
    class(typ_interp1),       intent(inout) :: this
    type (typ_tridiag)                      :: z
    real(DP), dimension(:,:), allocatable   :: a
    real(DP), dimension(-2:3)               :: c, x
    real(DP)                                :: h, h0, h1
    integer                                 :: i
    c = 0.0_DP
    x = 0.0_DP
    x(1) = this%x(1)
    x(2) = this%x(2)
    x(3) = this%x(3)
    c(1) = (2.0_DP * x(1) - x(2) - x(3)) / (x(2) - x(1)) / (x(3) - x(1))
    c(2) = (x(3) - x(1))                 / (x(2) - x(1)) / (x(3) - x(2))
    c(3) = (x(1) - x(2))                 / (x(3) - x(2)) / (x(3) - x(1))
    this%s1(1) = c(1) * this%f(1) + c(2) * this%f(2) + c(3) * this%f(3)
    x(-2) = this%x(this%n-2)
    x(-1) = this%x(this%n-1)
    x( 0) = this%x(this%n)
    c(-2) = (x( 0) - x(-1))                 / (x(-2) - x(-1)) / (x(-2) - x( 0))
    c(-1) = (x(-2) - x( 0))                 / (x(-1) - x( 0)) / (x(-2) - x(-1))
    c( 0) = (2.0_DP * x(0) - x(-1) - x(-2)) / (x(-1) - x( 0)) / (x(-2) - x( 0))
    this%s1(this%n) = c(-2) * this%f(this%n-2) + c(-1) * this%f(this%n-1) + c(0) * this%f(this%n)
    do i = 2, this%n - 1
        x(-1) = this%x(i-1)
        x( 0) = this%x(i)
        x( 1) = this%x(i+1)
        c(-1) = (x(0) - x( 1))                  / (x(1) - x(-1)) / (x(0) - x(-1))
        c( 0) = (2.0_DP * x(0) - x(-1) - x(1)) / (x(0) - x(-1)) / (x(0) - x( 1))
        c( 1) = (x(0) - x(-1))                  / (x(1) - x(-1)) / (x(1) - x( 0))
        this%s1(i) = c(-1) * this%f(i-1) + c(0) * this%f(i) + c(1) * this%f(i+1)
    end do
    allocate(a(this%n,-1:1))
    h = this%x(2) - this%x(1)
    a(1,-1) =  0.0_DP
    a(1, 0) = -4.0_DP / h
    a(1, 1) = -2.0_DP / h
    this%s2(1) = 6.0_DP / h**2 * (this%f(1) - this%f(2))
    h = this%x(this%n) - this%x(this%n-1)
    a(this%n,-1) =  2.0_DP / h
    a(this%n, 0) =  4.0_DP / h
    a(this%n, 1) =  0.0_DP
    this%s2(this%n) = 6.0_DP / h**2 * (this%f(this%n) - this%f(this%n-1))
    do i = 2, this%n - 1
        h0 = this%x(i) - this%x(i-1)
        h1 = this%x(i+1) - this%x(i)
        a(i,-1) = 2.0_DP / h0
        a(i, 0) = 4.0_DP / h0 + 4.0_DP / h1
        a(i, 1) = 2.0_DP / h1
        this%s2(i) = 6.0_DP * (  (this%f(i+1) - this%f(i)) / h1**2 &
                               - (this%f(i-1) - this%f(i)) / h0**2)
    end do
    call z%setMatrix(a)
    call z%solSystem(this%s2)
    deallocate(a)
    end subroutine interp1_splineSlopes
!=======================================================================
    subroutine interp1_tabulate(this,arg_filename)
    implicit none
    class(typ_interp1), intent(in) :: this
```

```
    character(len=*),   intent(in) :: arg_filename
    real(DP)                       :: x, x1, x2
    integer                        :: i, j, m
    integer,            parameter  :: k = 20
    open(unit = 30, file = arg_filename // '_points.txt')
    do j = 1, this%n
        write(30,100) this%x(j), this%f(j)
    end do
    close(30)
    x1 = this%x(1)
    x2 = this%x(this%n)
    m = k * this%n
    open(unit = 30, file = arg_filename // '_curves.txt')
    do i = 0, m
        x = x1 + (x2 - x1) * dble(i) / dble(m)
        write(30,200) x, interp1_linear(this,x),    &
                         interp1_spline1(this,x,0), &
                         interp1_spline2(this,x,0)
    end do
    close(30)
100 format (f12.7, 5x, f12.7)
200 format (f12.7, 4x, 3(1x, f12.7))
    end subroutine interp1_tabulate
!===============================================================================
    function interp1_linear(this,arg_x) result(arg_f)
    class(typ_interp1), intent(in) :: this
    real(DP),           intent(in) :: arg_x
    real(DP)                       :: arg_f
    real(DP), dimension(0:1)       :: w
    integer                        :: i, l
    i = interp_interval(this%x,arg_x)
    if (i == 0) then
        print*, 'Error in {interp1_linear}'
        print*, '[arg_x] out of range'
        stop
    else
        w = interp_coeff(this%x(i-1),this%x(i),arg_x)
        arg_f = 0.0_DP
        do l = 0, 1
            arg_f = arg_f + w(l) * this%f(i-l)
        end do
    end if
    end function interp1_linear
!===============================================================================
    function interp1_spline1(this,arg_x,arg_k) result(arg_f)
    class(typ_interp1), intent(in) :: this
    integer,            intent(in) :: arg_k
    real(DP),           intent(in) :: arg_x
    real(DP)                       :: arg_f
    real(DP)                       :: h, u
    integer                        :: i
    i = interp_interval(this%x,arg_x)
    if (i == 0) then
        print*, 'Error in {interp1_spline1}'
        print*, '[arg_x] out of range'
        stop
    else
        h = this%x(i) - this%x(i-1)
        u = (arg_x - this%x(i-1)) / (this%x(i) - this%x(i-1))
```

```
        arg_f = (this%f(i-1)  * interp_phi1(u,arg_k)              + &
                 this%f(i)    * interp_phi2(u,arg_k)) / h**arg_k   + &
                (this%s1(i-1) * interp_psi1(u,arg_k)              + &
                 this%s1(i)   * interp_psi2(u,arg_k)) / h**(arg_k - 1)
    end if
    end function interp1_spline1
!=======================================================================================
    function interp1_spline2(this,arg_x,arg_k) result(arg_f)
    class(typ_interp1), intent(in) :: this
    integer,            intent(in) :: arg_k
    real(DP),           intent(in) :: arg_x
    real(DP)                       :: arg_f
    real(DP)                       :: h, u
    integer                        :: i
    i = interp_interval(this%x,arg_x)
    if (i == 0) then
        print*, 'Error in {interp1_spline2}'
        print*, '[arg_x] out of range'
        stop
    else
        h = this%x(i) - this%x(i-1)
        u = (arg_x - this%x(i-1)) / (this%x(i) - this%x(i-1))
        arg_f = (this%f(i-1)  * interp_phi1(u,arg_k)              + &
                 this%f(i)    * interp_phi2(u,arg_k)) / h**arg_k   + &
                (this%s2(i-1) * interp_psi1(u,arg_k)              + &
                 this%s2(i)   * interp_psi2(u,arg_k)) / h**(arg_k - 1)
    end if
    end function interp1_spline2
!=======================================================================================
end module mod_interp1
!#######################################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_hermiteInterp
use mod_constants
use mod_utilities
implicit none
type, public :: typ_hermiteInterp
    private
    real(DP)                 :: a, b
    real(DP), dimension(0:2) :: f, g
contains
    procedure :: setEndPoints => hermiteInterp_setEndPoints
    procedure :: getEndPoints => hermiteInterp_getEndPoints
    procedure :: setEndValues => hermiteInterp_setEndValues
    procedure :: evaluateFunc => hermiteInterp_evaluateFunc
end type typ_hermiteInterp
private :: hermiteInterp_setEndPoints
private :: hermiteInterp_getEndPoints
private :: hermiteInterp_setEndValues
private :: hermiteInterp_evaluateFunc
private :: hermiteInterp_chi1
private :: hermiteInterp_chi2
private :: hermiteInterp_phi1
private :: hermiteInterp_phi2
private :: hermiteInterp_psi1
private :: hermiteInterp_psi2
public  :: hermiteInterp_check
type(typ_hermiteInterp), private :: sav_hermite
```

```
contains
!==============================================================================
    subroutine hermiteInterp_setEndPoints(this,arg_a,arg_b)
    implicit none
    class(typ_hermiteInterp), intent(inout) :: this
    real(DP),                 intent(in)    :: arg_a, arg_b
    this%a = arg_a
    this%b = arg_b
    end subroutine hermiteInterp_setEndPoints
!==============================================================================
    subroutine hermiteInterp_getEndPoints(this,arg_a,arg_b)
    implicit none
    class(typ_hermiteInterp), intent(in)  :: this
    real(DP),                 intent(out) :: arg_a, arg_b
    arg_a = this%a
    arg_b = this%b
    end subroutine hermiteInterp_getEndPoints
!==============================================================================
    subroutine hermiteInterp_setEndValues(this,arg_f,arg_g)
    implicit none
    class(typ_hermiteInterp), intent(inout) :: this
    real(DP), dimension(0:2)                :: arg_f, arg_g
    this%f = arg_f
    this%g = arg_g
    end subroutine hermiteInterp_setEndValues
!==============================================================================
    function hermiteInterp_evaluateFunc(this,arg_x,arg_k) result(arg_f)
    implicit none
    class(typ_hermiteInterp), intent(in) :: this
    integer,                  intent(in) :: arg_k
    real(DP),                 intent(in) :: arg_x
    real(DP)                             :: arg_f
    real(DP)                             :: u, c
    c = (this%b - this%a)
    u = (arg_x  - this%a) / c
    arg_f = (         this%f(0) * hermiteInterp_chi1(u,arg_k) &
             +        this%g(0) * hermiteInterp_chi2(u,arg_k) &
             + c    * this%f(1) * hermiteInterp_phi1(u,arg_k) &
             + c    * this%g(1) * hermiteInterp_phi2(u,arg_k) &
             + c**2 * this%f(2) * hermiteInterp_psi1(u,arg_k) &
             + c**2 * this%g(2) * hermiteInterp_psi2(u,arg_k)) / c**arg_k
    end function hermiteInterp_evaluateFunc
!==============================================================================
    function hermiteInterp_chi1(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = 1.0_DP - 10.0_DP * arg_u**3 +  15.0_DP * arg_u**4 -   6.0_DP * arg_u**5
        case(1)
            arg_f =        - 30.0_DP * arg_u**2 +  60.0_DP * arg_u**3 -  30.0_DP * arg_u**4
        case(2)
            arg_f =        - 60.0_DP * arg_u    + 180.0_DP * arg_u**2 - 120.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_chi1
```

```
!================================================================================
    function hermiteInterp_chi2(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = 10.0_DP * arg_u**3 -  15.0_DP * arg_u**4 +   6.0_DP * arg_u**5
        case(1)
            arg_f = 30.0_DP * arg_u**2 -  60.0_DP * arg_u**3 +  30.0_DP * arg_u**4
        case(2)
            arg_f = 60.0_DP * arg_u    - 180.0_DP * arg_u**2 + 120.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_chi2
!================================================================================
    function hermiteInterp_phi1(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = arg_u   - 6.0_DP * arg_u**3 +  8.0_DP * arg_u**4 -  3.0_DP * arg_u**5
        case(1)
            arg_f = 1.0_DP - 18.0_DP * arg_u**2 + 32.0_DP * arg_u**3 - 15.0_DP * arg_u**4
        case(2)
            arg_f =        - 36.0_DP * arg_u    + 96.0_DP * arg_u**2 - 60.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_phi1
!================================================================================
    function hermiteInterp_phi2(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = - 4.0_DP * arg_u**3 +  7.0_DP * arg_u**4 -  3.0_DP * arg_u**5
        case(1)
            arg_f = -12.0_DP * arg_u**2 + 28.0_DP * arg_u**3 - 15.0_DP * arg_u**4
        case(2)
            arg_f = -24.0_DP * arg_u    + 84.0_DP * arg_u**2 - 60.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_phi2
!================================================================================
    function hermiteInterp_psi1(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
```

```
            arg_f = 0.5_DP * arg_u**2 - 1.5_DP * arg_u**3 +  1.5_DP * arg_u**4 -  0.5_DP * arg_u**5
        case(1)
            arg_f =          arg_u    - 4.5_DP * arg_u**2 +  6.0_DP * arg_u**3 -  2.5_DP * arg_u**4
        case(2)
            arg_f = 1.0_DP           - 9.0_DP * arg_u    + 18.0_DP * arg_u**2 - 10.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_psi1
!=========================================================================
    function hermiteInterp_psi2(arg_u,arg_k) result(arg_f)
    implicit none
    integer,  intent(in) :: arg_k
    real(DP), intent(in) :: arg_u
    real(DP)             :: arg_f
    select case(arg_k)
        case(0)
            arg_f = 0.5_DP * arg_u**3 -           arg_u**4 +  0.5_DP * arg_u**5
        case(1)
            arg_f = 1.5_DP * arg_u**2 -  4.0_DP * arg_u**3 +  2.5_DP * arg_u**4
        case(2)
            arg_f = 3.0_DP * arg_u    - 12.0_DP * arg_u**2 + 10.0_DP * arg_u**3
        case default
            arg_f = 0.0_DP
    end select
    end function hermiteInterp_psi2
!=========================================================================
    subroutine hermiteInterp_check
    real(DP)                 :: a, b, h, x
    real(DP), dimension(0:2) :: f, g
    integer                  :: i, iMax
    h = 1.0e-4_DP
    a = -5.2_DP
    b =  7.8_DP
    f = (/  3.7_DP, 1.4_DP, -5.3_DP /)
    g = (/ -7.4_DP, 2.3_DP,  8.5_DP /)
    call sav_hermite%setEndPoints(a,b)
    call sav_hermite%setEndValues(f,g)
    iMax = 20
    open(unit = 30, file = 'testHermite.txt')
    do i = 0, iMax
        x = a + (b - a) * dble(i) / dble(iMax)
        write(30,100) i, x, sav_hermite%evaluateFunc(x,0), FiniteDifference(x,0,h,hermiteInterp_func), &
                            sav_hermite%evaluateFunc(x,1), FiniteDifference(x,1,h,hermiteInterp_func), &
                            sav_hermite%evaluateFunc(x,2), FiniteDifference(x,2,h,hermiteInterp_func)
    end do
    close(30)
100 format(i3, 1x, f12.7, 3(8x, f12.7, 1x, f12.7))
    end subroutine hermiteInterp_check
!=========================================================================
    function hermiteInterp_func(arg_x) result(arg_f)
    real(DP), intent(in) :: arg_x
    real(DP)             :: arg_f
    arg_f = sav_hermite%evaluateFunc(arg_x,0)
    end function hermiteInterp_func
!=========================================================================
end module mod_hermiteInterp
!#########################################################################
! Copyright 2011 by Ludwig C. Nitsche
```

```
!-----------------------------------
module mod_polyFit
use mod_constants
use mod_utilities
use mod_hermiteInterp
implicit none
type, extends(typ_hermiteInterp), public :: typ_polyFit
    private
    integer                              :: n
    real(DP), dimension(:), allocatable :: x, y
contains
    procedure :: setData  => polyFit_setData
    procedure :: evaluate => polyFit_evaluate
end type typ_polyFit
private :: polyFit_setData
private :: polyFit_evaluate
public  :: polyFit_check
private :: polyFit_func
private :: polyFit_beta
type(typ_polyFit), private :: sav_poly
contains
!=============================================================================
    subroutine polyFit_setData(this,arg_x,arg_y)
    implicit none
    class(typ_polyFit),     intent(inout) :: this
    real(DP), dimension(:), intent(in)    :: arg_x, arg_y
    integer                               :: i
    this%n = min(size(arg_x),size(arg_y))
    if (allocated(this%x)) then
        deallocate(this%x)
    end if
    if (allocated(this%y)) then
        deallocate(this%y)
    end if
    allocate(this%x(this%n))
    allocate(this%y(this%n))
    this%x = arg_x(1:this%n)
    do i = 1, this%n
        this%y(i) = arg_y(i) - this%evaluateFunc(this%x(i),0)
    end do
    end subroutine polyFit_setData
!=============================================================================
    function polyFit_evaluate(this,arg_x) result(arg_f)
    implicit none
    class(typ_polyFit), intent(in) :: this
    real(DP),           intent(in) :: arg_x
    real(DP)                       :: arg_f
    integer                        :: i
    arg_f = this%evaluateFunc(arg_x,0)
    do i = 1, this%n
        arg_f = arg_f + this%y(i) * polyFit_beta(this,arg_x,i)
    end do
    end function polyFit_evaluate
!=============================================================================
    subroutine polyFit_check
    real(DP)                 :: a, b, h, x
    real(DP), dimension(0:2) :: f, g
    real(DP), dimension(3)   :: xMid, yMid
    integer                  :: i, iMax
```

```
    h = 1.0e-4_DP
    a = -5.2_DP
    b =  7.8_DP
    f = (/  3.7_DP, 1.4_DP, -5.3_DP /)
    g = (/ -7.4_DP, 2.3_DP,  8.5_DP /)
    xMid = (/ -0.5_DP, 1.5_DP, 4.5_DP /)
    yMid = (/  3.0_DP, 5.0_DP, 9.0_DP /)
    call sav_poly%setEndPoints(a,b)
    call sav_poly%setEndValues(f,g)
    call sav_poly%setData(xMid,yMid)
    iMax = nint((b - a) / 0.1_DP)
    open(unit = 30, file = 'testPolyFit.txt')
    do i = 0, iMax
        x = a + (b - a) * dble(i) / dble(iMax)
        write(30,100) i, x, FiniteDifference(x,0,h,polyFit_func), &
                            FiniteDifference(x,1,h,polyFit_func), &
                            FiniteDifference(x,2,h,polyFit_func)
    end do
    close(30)
100 format(i3, 1x, f12.7, 8x, 3(1x, f12.7))
    end subroutine polyFit_check
!=============================================================================
    function polyFit_func(arg_x) result(arg_f)
    real(DP), intent(in) :: arg_x
    real(DP)             :: arg_f
    arg_f = sav_poly%evaluate(arg_x)
    end function polyFit_func
!=============================================================================
    function polyFit_beta(this,arg_x,arg_i) result(arg_f)
    implicit none
    class(typ_polyFit), intent(in) :: this
    integer,            intent(in) :: arg_i
    real(DP),           intent(in) :: arg_x
    real(DP)                       :: arg_f
    real(DP)                       :: a, b
    integer                        :: k
    call this%getEndPoints(a,b)
    arg_f = (arg_x - a)**3 / (this%x(arg_i) - a)**3 * (arg_x - b)**3 / (this%x(arg_i) - b)**3
    do k = 1, this%n
        if (k == arg_i) then
            cycle
        end if
        arg_f = arg_f * (arg_x - this%x(k)) / (this%x(arg_i) - this%x(k))
    end do
    end function polyFit_beta
!=============================================================================
end module mod_polyFit
!#############################################################################
! Copyright 2011 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
    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
    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)
    end do
    xInc = 0.25_DP
    xMin = 0.25_DP
    xMax = 2.0_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
    end do
    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))
    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), 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
    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)
    end do
    rInc = 0.25_DP
    rMin = 0.25_DP
    rMax = 2.0_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
    end do
    end do
    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))
    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_r)
    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(-1:1,0:2) :: 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 =  0, 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 = (-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 2011 by Ludwig C. Nitsche
!----------------------------------
```

```
module mod_generalSolutionPlus1
use mod_constants
use mod_utilities
implicit none
private
public :: generalSolutionPlus1_Polar
public :: generalSolutionPlus1_Cartesian
contains
!================================================================================
    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)                             :: lnr1, lnr2
    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
    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
!=======================================================================
end module mod_generalSolutionPlus1
!#######################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_generalSolutionMinus1
use mod_constants
use mod_utilities
implicit none
private
public :: generalSolutionMinus1_Polar
public :: generalSolutionMinus1_Cartesian
contains
!=======================================================================
    subroutine generalSolutionMinus1_Coeff_p(t,fpa0,fpa1,fpa2,fpb0,fpb1,fpc0)
    implicit none
    real(DP)                             :: s, c
    real(DP),               intent(in)   :: t
    real(DP), dimension(4), intent(out)  :: 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
    end subroutine generalSolutionMinus1_Coeff_p
!================================================================================
    subroutine generalSolutionMinus1_Coeff_r(t,fra0,fra1,fra2,frb0,frb1,frc0)
    implicit none
    real(DP)                              :: s, c
    real(DP),               intent(in)  :: t
    real(DP), dimension(4), intent(out) :: 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
    end subroutine generalSolutionMinus1_Coeff_r
!================================================================================
    subroutine generalSolutionMinus1_Coeff_t(t,fta0,fta1,fta2,ftb0,ftb1,ftc0)
    implicit none
    real(DP)                              :: s, c
    real(DP),               intent(in)  :: t
    real(DP), dimension(4), intent(out) :: 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
    end subroutine generalSolutionMinus1_Coeff_t
!===============================================================================
    subroutine generalSolutionMinus1_Coeff_s(t,fsa0,fsa1,fsa2,fsb0,fsb1,fsc0)
    implicit none
    real(DP)                             :: s, c
    real(DP),               intent(in)   :: t
    real(DP), dimension(4), intent(out)  :: 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
    end subroutine generalSolutionMinus1_Coeff_s
!===============================================================================
    subroutine generalSolutionMinus1_Polar(a,b,c,r,t,pr,vr,vt,st)
    implicit none
    real(DP), dimension(4), intent(in)  :: a, b, c
    real(DP),               intent(in)  :: r, t
    real(DP),               intent(out) :: pr, vr, vt, st
    real(DP)                            :: lnr1, lnr2
    real(DP)                            :: fs0, fs1, fs2
    real(DP)                            :: fp0, fp1, fp2
    real(DP)                            :: fr0, fr1, fr2
    real(DP)                            :: ft0, ft1, ft2
    real(DP), dimension(4)              :: fsa0, fsa1, fsa2, fsb0, fsb1, fsc0
    real(DP), dimension(4)              :: fpa0, fpa1, fpa2, fpb0, fpb1, fpc0
    real(DP), dimension(4)              :: fra0, fra1, fra2, frb0, frb1, frc0
    real(DP), dimension(4)              :: fta0, fta1, fta2, ftb0, ftb1, ftc0
    lnr2 = log(r)**2
    lnr1 = log(r)
```

```
!
    call generalSolutionMinus1_Coeff_s(t,fsa0,fsa1,fsa2,fsb0,fsb1,fsc0)
    call generalSolutionMinus1_Coeff_p(t,fpa0,fpa1,fpa2,fpb0,fpb1,fpc0)
    call generalSolutionMinus1_Coeff_r(t,fra0,fra1,fra2,frb0,frb1,frc0)
    call generalSolutionMinus1_Coeff_t(t,fta0,fta1,fta2,ftb0,ftb1,ftc0)
!
    fs2 = dot_product(a,fsa2)
    fs1 = dot_product(a,fsa1) + dot_product(b,fsb1)
    fs0 = dot_product(a,fsa0) + dot_product(b,fsb0) + dot_product(c,fsc0)
!
    fp2 = dot_product(a,fpa2)
    fp1 = dot_product(a,fpa1) + dot_product(b,fpb1)
    fp0 = dot_product(a,fpa0) + dot_product(b,fpb0) + dot_product(c,fpc0)
!
    fr2 = dot_product(a,fra2)
    fr1 = dot_product(a,fra1) + dot_product(b,frb1)
    fr0 = dot_product(a,fra0) + dot_product(b,frb0) + dot_product(c,frc0)
!
    ft2 = dot_product(a,fta2)
    ft1 = dot_product(a,fta1) + dot_product(b,ftb1)
    ft0 = dot_product(a,fta0) + dot_product(b,ftb0) + dot_product(c,ftc0)
!
    st = (fs2 * lnr2 + fs1 * lnr1 + fs0) / r**2
    pr = (fp2 * lnr2 + fp1 * lnr1 + fp0) / r**2
    vr = (fr2 * lnr2 + fr1 * lnr1 + fr0) / r
    vt = (ft2 * lnr2 + ft1 * lnr1 + ft0) / r
    end subroutine generalSolutionMinus1_Polar
!=============================================================================
    subroutine generalSolutionMinus1_Cartesian(a,b,c,p,f)
    real(DP), dimension(4),   intent(in)  :: a, b, c
    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(a,b,c,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine generalSolutionMinus1_Cartesian
!=============================================================================
end module mod_generalSolutionMinus1
!#############################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_generalSolution
use mod_constants
use mod_utilities
implicit none
private
public :: generalSolution_Polar
public :: generalSolution_Cartesian
public :: generalSolution_BoundaryCoefficients
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_Coeff_p(t,w,fpa0,fpa1,fpa2,fpb0,fpb1,fpc0)
    implicit none
    real(DP),               intent(in)  :: t, w
    real(DP), dimension(4), intent(out) :: 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)
    end subroutine generalSolution_Coeff_p
!=============================================================================
```

```
    subroutine generalSolution_Coeff_r(t,w,fra0,fra1,fra2,frb0,frb1,frc0)
    implicit none
    real(DP),               intent(in)  :: t, w
    real(DP), dimension(4), intent(out) :: 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)
    end subroutine generalSolution_Coeff_r
!=============================================================================
    subroutine generalSolution_Coeff_t(t,w,fta0,fta1,fta2,ftb0,ftb1,ftc0)
    implicit none
    real(DP),               intent(in)  :: t, w
    real(DP), dimension(4), intent(out) :: 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)
    end subroutine generalSolution_Coeff_t
!=============================================================================
    subroutine generalSolution_Coeff_s(t,w,fsa0,fsa1,fsa2,fsb0,fsb1,fsc0)
    implicit none
    real(DP),               intent(in)  :: t, w
    real(DP), dimension(4), intent(out) :: 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)
    end subroutine generalSolution_Coeff_s
!=============================================================================
    subroutine generalSolution_Cartesian(w,a,b,c,p,f)
    real(DP),                 intent(in)  :: w
    real(DP), dimension(4),   intent(in)  :: a, b, c
    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,a,b,c,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,a,b,c,r,t,pr,vr,vt,st)
    implicit none
    real(DP),               intent(in)  :: r, t, w
    real(DP), dimension(4), intent(in)  :: a, b, c
    real(DP),               intent(out) :: pr, vr, vt, st
    real(DP)                            :: lnr1, lnr2
    real(DP)                            :: fs0, fs1, fs2
```

```
    real(DP)                              :: fp0, fp1, fp2
    real(DP)                              :: fr0, fr1, fr2
    real(DP)                              :: ft0, ft1, ft2
    real(DP), dimension(4)                :: fsa0, fsa1, fsa2, fsb0, fsb1, fsc0
    real(DP), dimension(4)                :: fpa0, fpa1, fpa2, fpb0, fpb1, fpc0
    real(DP), dimension(4)                :: fra0, fra1, fra2, frb0, frb1, frc0
    real(DP), dimension(4)                :: fta0, fta1, fta2, ftb0, ftb1, ftc0
    real(DP),                 parameter   :: tol = 1.0e-8_DP
    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,fsa0,fsa1,fsa2,fsb0,fsb1,fsc0)
        call generalSolution_Coeff_p(t,w,fpa0,fpa1,fpa2,fpb0,fpb1,fpc0)
        call generalSolution_Coeff_r(t,w,fra0,fra1,fra2,frb0,frb1,frc0)
        call generalSolution_Coeff_t(t,w,fta0,fta1,fta2,ftb0,ftb1,ftc0)
!
        fs2 = dot_product(a,fsa2)
        fs1 = dot_product(a,fsa1) + dot_product(b,fsb1)
        fs0 = dot_product(a,fsa0) + dot_product(b,fsb0) + dot_product(c,fsc0)
!
        fp2 = dot_product(a,fpa2)
        fp1 = dot_product(a,fpa1) + dot_product(b,fpb1)
        fp0 = dot_product(a,fpa0) + dot_product(b,fpb0) + dot_product(c,fpc0)
!
        fr2 = dot_product(a,fra2)
        fr1 = dot_product(a,fra1) + dot_product(b,frb1)
        fr0 = dot_product(a,fra0) + dot_product(b,frb0) + dot_product(c,frc0)
!
        ft2 = dot_product(a,fta2)
        ft1 = dot_product(a,fta1) + dot_product(b,ftb1)
        ft0 = dot_product(a,fta0) + dot_product(b,ftb0) + dot_product(c,ftc0)
!
        st = (fs2 * lnr2 + fs1 * lnr1 + fs0) * r**(w - 1.0_DP)
        pr = (fp2 * lnr2 + fp1 * lnr1 + fp0) * r**(w - 1.0_DP)
        vr = (fr2 * lnr2 + fr1 * lnr1 + fr0) * r**w
        vt = (ft2 * lnr2 + ft1 * lnr1 + ft0) * r**w
    end if
    end subroutine generalSolution_Polar
!===============================================================================
    subroutine generalSolution_BoundaryCoefficients(w,a,b,c)
    implicit none
    real(DP),                intent(in) :: w
    real(DP), dimension(4), intent(in) :: a, b, c
    real(DP)                            :: fs0, fs1, fs2
    real(DP)                            :: fp0, fp1, fp2
    real(DP)                            :: fr0, fr1, fr2
    real(DP)                            :: ft0, ft1, ft2
    real(DP)                            :: gp0, gp1, gp2
    real(DP)                            :: gr0, gr1, gr2
    real(DP)                            :: gt0, gt1, gt2
    real(DP), dimension(4)              :: fsa0, fsa1, fsa2, fsb0, fsb1, fsc0
    real(DP), dimension(4)              :: fpa0, fpa1, fpa2, fpb0, fpb1, fpc0
    real(DP), dimension(4)              :: fra0, fra1, fra2, frb0, frb1, frc0
```

```
    real(DP), dimension(4)             :: fta0, fta1, fta2, ftb0, ftb1, ftc0
    real(DP), dimension(4)             :: gpa0, gpa1, gpa2, gpb0, gpb1, gpc0
    real(DP), dimension(4)             :: gra0, gra1, gra2, grb0, grb1, grc0
    real(DP), dimension(4)             :: gta0, gta1, gta2, gtb0, gtb1, gtc0
!
    call generalSolution_Coeff_s(0.0_DP,w,fsa0,fsa1,fsa2,fsb0,fsb1,fsc0)
    call generalSolution_Coeff_p(0.0_DP,w,fpa0,fpa1,fpa2,fpb0,fpb1,fpc0)
    call generalSolution_Coeff_r(0.0_DP,w,fra0,fra1,fra2,frb0,frb1,frc0)
    call generalSolution_Coeff_t(0.0_DP,w,fta0,fta1,fta2,ftb0,ftb1,ftc0)
!
    call generalSolution_Coeff_p(    pi,w,gpa0,gpa1,gpa2,gpb0,gpb1,gpc0)
    call generalSolution_Coeff_r(    pi,w,gra0,gra1,gra2,grb0,grb1,grc0)
    call generalSolution_Coeff_t(    pi,w,gta0,gta1,gta2,gtb0,gtb1,gtc0)
!
    fs2 = dot_product(a,fsa2)
    fs1 = dot_product(a,fsa1) + dot_product(b,fsb1)
    fs0 = dot_product(a,fsa0) + dot_product(b,fsb0) + dot_product(c,fsc0)
!
    fp2 = dot_product(a,fpa2)
    fp1 = dot_product(a,fpa1) + dot_product(b,fpb1)
    fp0 = dot_product(a,fpa0) + dot_product(b,fpb0) + dot_product(c,fpc0)
!
    fr2 = dot_product(a,fra2)
    fr1 = dot_product(a,fra1) + dot_product(b,frb1)
    fr0 = dot_product(a,fra0) + dot_product(b,frb0) + dot_product(c,frc0)
!
    ft2 = dot_product(a,fta2)
    ft1 = dot_product(a,fta1) + dot_product(b,ftb1)
    ft0 = dot_product(a,fta0) + dot_product(b,ftb0) + dot_product(c,ftc0)
!
    gp2 = dot_product(a,gpa2)
    gp1 = dot_product(a,gpa1) + dot_product(b,gpb1)
    gp0 = dot_product(a,gpa0) + dot_product(b,gpb0) + dot_product(c,gpc0)
!
    gr2 = dot_product(a,gra2)
    gr1 = dot_product(a,gra1) + dot_product(b,grb1)
    gr0 = dot_product(a,gra0) + dot_product(b,grb0) + dot_product(c,grc0)
!
    gt2 = dot_product(a,gta2)
    gt1 = dot_product(a,gta1) + dot_product(b,gtb1)
    gt0 = dot_product(a,gta0) + dot_product(b,gtb0) + dot_product(c,gtc0)
!
    print 300
    print 100, ' s', ' 0', fs2, fs1, fs0, w - 1.0_DP
    print 100, ' p', ' 0', fp2, fp1, fp0, w - 1.0_DP
    print 100, 'vr', ' 0', fr2, fr1, fr0, w
    print 100, 'vt', ' 0', ft2, ft1, ft0, w
    print 200
    print 100, ' p', 'pi', gp2, gp1, gp0, w - 1.0_DP
    print 100, 'vr', 'pi', gr2, gr1, gr0, w
    print 100, 'vt', 'pi', gt2, gt1, gt0, w
    print 300
100 format (a, '(r,', a, ') = [', f18.14, 1x, f18.14, 1x, f18.14, ' ] r^(', f3.1, ')')
200 format ()
300 format (79('-'))
!
    end subroutine generalSolution_BoundaryCoefficients
!=======================================================================
end module mod_generalSolution
```

```
!###############################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module outer1
use mod_constants
use mod_utilities
use mod_generalSolutionMinus1
use mod_stokesEq2d
implicit none
private
public :: outer1_Setup
public :: outer1_Polar
public :: outer1_Cartesian
real(DP), dimension(4) :: sav_a, sav_b, sav_c
contains
!===============================================================================
    subroutine outer1_Const
    implicit none
    sav_a(1) =  1.37_DP
    sav_a(2) =  2.45_DP
    sav_a(3) = -1.87_DP
    sav_a(4) =  3.28_DP
    sav_b(1) =  5.14_DP
    sav_b(2) =  0.74_DP
    sav_b(3) = -2.72_DP
    sav_b(4) =  0.58_DP
    sav_c(1) =  1.83_DP
    sav_c(2) =  0.43_DP
    sav_c(3) = -2.17_DP
    sav_c(4) =  1.27_DP
    end subroutine outer1_Const
!===============================================================================
    subroutine outer1_Cartesian(p,f)
    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 outer1_Polar(r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine outer1_Cartesian
!===============================================================================
    subroutine outer1_Polar(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_a,sav_b,sav_c,r,t,pr,vr,vt,st)
    end subroutine outer1_Polar
!===============================================================================
    subroutine outer1_Setup
    implicit none
    call outer1_Const
    call stokesEq2d_Cartesian(outer1_Cartesian,'outer0c')
    call stokesEq2d_Polar(outer1_Polar,'outer0p')
    end subroutine outer1_Setup
!===============================================================================
end module outer1
!###############################################################################
```

```
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_inner1
use mod_constants
use mod_utilities
use mod_generalSolutionPlus1
use mod_stokesEq2d
implicit none
private
public :: inner1_Setup
public :: inner1_Polar
public :: inner1_Cartesian
real(DP), dimension(10) :: sav_d
contains
!================================================================================
    subroutine inner1_Const
    implicit none
    sav_d( 1) =  1.37_DP
    sav_d( 2) =  2.45_DP
    sav_d( 3) = -1.87_DP
    sav_d( 4) =  3.28_DP
    sav_d( 5) =  5.14_DP
    sav_d( 6) =  0.74_DP
    sav_d( 7) = -2.72_DP
    sav_d( 8) =  0.58_DP
    sav_d( 9) =  1.83_DP
    sav_d(10) =  0.43_DP
    end subroutine inner1_Const
!================================================================================
    subroutine inner1_Cartesian(p,f)
    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 inner1_Polar(r,t,pr,vr,vt,st)
    call PolarToCartesian(t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine inner1_Cartesian
!================================================================================
    subroutine inner1_Polar(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 inner1_Polar
!================================================================================
    subroutine inner1_Setup
    implicit none
    call inner1_Const
    call stokesEq2d_Cartesian(inner1_Cartesian,'inner0c')
    call stokesEq2d_Polar(inner1_Polar,'inner0p')
    end subroutine inner1_Setup
!================================================================================
end module mod_inner1
!################################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_inner
```

```
use mod_constants
use mod_utilities
use mod_generalSolution
use mod_stokesEq2d
implicit none
private
public :: inner_Setup
public :: inner_Polar
public :: inner_Cartesian
real(DP), dimension(4) :: a1, a2, a3, a4, a5, a6, a7, a8, a9
real(DP), dimension(4) :: b1, b2, b3, b4, b5, b6, b7, b8, b9
real(DP), dimension(4) :: c1, c2, c3, c4, c5, c6, c7, c8, c9
real(DP)               :: w1, w2, w3, w4, w5, w6, w7, w8, w9
integer                :: kSelect
contains
!==============================================================================
    subroutine inner_Latex
    implicit none
    integer :: i
    open(unit = 30, file = 'table.tex')
    write(30,110)
    write(30,200) '1', (LatexSciNotation(a1(i)), i = 1, 4)
    write(30,200) '2', (LatexSciNotation(a2(i)), i = 1, 4)
    write(30,200) '3', (LatexSciNotation(a3(i)), i = 1, 4)
    write(30,200) '4', (LatexSciNotation(a4(i)), i = 1, 4)
    write(30,300)
    write(30,120)
    write(30,200) '1', (LatexSciNotation(b1(i)), i = 1, 4)
    write(30,200) '2', (LatexSciNotation(b2(i)), i = 1, 4)
    write(30,200) '3', (LatexSciNotation(b3(i)), i = 1, 4)
    write(30,200) '4', (LatexSciNotation(b4(i)), i = 1, 4)
    write(30,300)
    write(30,130)
    write(30,200) '1', (LatexSciNotation(c1(i)), i = 1, 4)
    write(30,200) '2', (LatexSciNotation(c2(i)), i = 1, 4)
    write(30,200) '3', (LatexSciNotation(c3(i)), i = 1, 4)
    write(30,200) '4', (LatexSciNotation(c4(i)), i = 1, 4)
    write(30,300)
    close(30)
110 format('\begin{tabular}{l|l|l|l|l|}' / &
           '$n$ & $a_{1}^{\{n\}}$ & $a_{2}^{\{n\}}$ & $a_{3}^{\{n\}}$ & $a_{4}^{\{n\}}$' / &
           '\\ \hline')
120 format('\begin{tabular}{l|l|l|l|l|}' / &
           '$n$ & $b_{1}^{\{n\}}$ & $b_{2}^{\{n\}}$ & $b_{3}^{\{n\}}$ & $b_{4}^{\{n\}}$' / &
           '\\ \hline')
130 format('\begin{tabular}{l|l|l|l|l|}' / &
           '$n$ & $c_{1}^{\{n\}}$ & $c_{2}^{\{n\}}$ & $c_{3}^{\{n\}}$ & $c_{4}^{\{n\}}$' / &
           '\\ \hline')
200 format(a, ' & ', a / 2('  & ', a /), '  & ', a, ' \\ \hline')
300 format('\end{tabular}' /)
    end subroutine inner_Latex
!==============================================================================
    subroutine inner_Const
    implicit none
    print*
    print*, '1st inner solution'
    call inner_Const1
    call inner_Const2
    call inner_Const3
```

```
    call inner_Const4
    print*
    print*, '2nd inner solution'
    call inner_Const5
    call inner_Const6
    call inner_Const7
    print*
    print*, '3rd inner solution'
    call inner_Const8
    call inner_Const9
    end subroutine inner_Const
!===============================================================================
    subroutine inner_Const1
    implicit none
    w1 = 0.5_DP
!
    a1 = 0.0_DP
    b1 = 0.0_DP
    c1 = 0.0_DP
!
    c1(1) = -1.0_DP / 6.0_DP
    c1(3) = -0.5_DP
!
    call generalSolution_BoundaryCoefficients(w1,a1,b1,c1)
    end subroutine inner_Const1
!===============================================================================
    subroutine inner_Const2
    implicit none
    w2 = 1.5_DP
!
    a2 = 0.0_DP
    b2 = 0.0_DP
    c2 = 0.0_DP
!
    b2(1) =  0.1_DP / ( 3.0_DP * pi)
    b2(3) = -1.0_DP / ( 6.0_DP * pi)
    c2(1) =  4.0_DP / (75.0_DP * pi)
    c2(2) =  1.0_DP /  30.0_DP
    c2(4) = -1.0_DP /   6.0_DP
!
    call generalSolution_BoundaryCoefficients(w2,a2,b2,c2)
    end subroutine inner_Const2
!===============================================================================
    subroutine inner_Const3
    implicit none
    w3 = 2.5_DP
!
    a3 = 0.0_DP
    b3 = 0.0_DP
    c3 = 0.0_DP
!
    a3(1) =  0.72372274030215e-03_DP
    a3(3) = -0.16886863940387e-02_DP
!
    b3(1) =                           - 0.27570390106645e-04_DP
    b3(2) =  1.0_DP / (30.0_DP * pi) - 0.60630454511198e-02_DP
    b3(3) =                             0.13509491152311e-02_DP
    b3(4) = -1.0_DP / (30.0_DP * pi)
!
```

```
    c3(1) = -1.0_DP /  30.0_DP       + 0.32955225126152e-01_DP
    c3(2) = -1.0_DP / (75.0_DP * pi) + 0.41575168807679e-02_DP
    c3(3) =  0.0_DP
    c3(4) =  1.0_DP / (75.0_DP * pi)
!
!    b3(2) =  21.0_DP / (1470.0_DP * pi)
!    c3(2) =  -1.0_DP / (3675.0_DP * pi)
!
    call generalSolution_BoundaryCoefficients(w3,a3,b3,c3)
    end subroutine inner_Const3
!================================================================================
    subroutine inner_Const4
    implicit none
    w4 = 3.5_DP
!
    a4 = 0.0_DP
    b4 = 0.0_DP
    c4 = 0.0_DP
!
    a4(2) =  0.24124091343414D-03
    a4(4) = -0.24124091343414D-03
!
    b4(1) = -0.15157613627800D-02
    b4(2) = -0.33084468128110D-03
    b4(3) =  0.15157613627800D-02
    b4(4) =  0.33084468128110D-03
!
    c4(1) =  0.10393792201920D-02
    c4(2) =  0.94527051794601D-04
    c4(3) = -0.10393792201920D-02
    c4(4) = -0.94527051794601D-04
!
    call generalSolution_BoundaryCoefficients(w4,a4,b4,c4)
    end subroutine inner_Const4
!================================================================================
    subroutine inner_Const5
    implicit none
    w5 = 1.5_DP
!
    a5 = 0.0_DP
    b5 = 0.0_DP
    c5 = 0.0_DP
!
    c5(1) =  0.1_DP
    c5(3) = -0.5_DP
!
    call generalSolution_BoundaryCoefficients(w5,a5,b5,c5)
    end subroutine inner_Const5
!================================================================================
    subroutine inner_Const6
    implicit none
    w6 = 2.5_DP
!
    a6 = 0.0_DP
    b6 = 0.0_DP
    c6 = 0.0_DP
!
    b6(1) =  3.0_DP / ( 70.0_DP * pi)
    b6(3) = -1.0_DP / ( 10.0_DP * pi)
```

```
!
    c6(1) =  4.0_DP / (245.0_DP * pi)
    c6(2) =  3.0_DP /   70.0_DP
    c6(4) = -0.1_DP
!
    call generalSolution_BoundaryCoefficients(w6,a6,b6,c6)
    end subroutine inner_Const6
!=============================================================================
    subroutine inner_Const7
    implicit none
    w7 = 3.5_DP
!
    a7 = 0.0_DP
    b7 = 0.0_DP
    c7 = 0.0_DP
!
    b7(2) =  1.0_DP / ( 70.0_DP * pi)
    b7(4) = -1.0_DP / ( 70.0_DP * pi)
!
    c7(1) = -1.0_DP / ( 70.0_DP)
    c7(2) = -1.0_DP / (245.0_DP * pi)
    c7(3) =  1.0_DP / ( 70.0_DP)
    c7(4) =  1.0_DP / (245.0_DP * pi)
!
    call generalSolution_BoundaryCoefficients(w7,a7,b7,c7)
    end subroutine inner_Const7
!=============================================================================
    subroutine inner_Const8
    implicit none
    w8 = 2.5_DP
!
    a8 = 0.0_DP
    b8 = 0.0_DP
    c8 = 0.0_DP
!
    c8(1) =  3.0_DP / 14.0_DP
    c8(3) = -0.5_DP
!
    call generalSolution_BoundaryCoefficients(w8,a8,b8,c8)
    end subroutine inner_Const8
!=============================================================================
    subroutine inner_Const9
    implicit none
    w9 = 3.5_DP
!
    a9 = 0.0_DP
    b9 = 0.0_DP
    c9 = 0.0_DP
!
    c9(2) =  1.0_DP / 14.0_DP
    c9(4) = -1.0_DP / 14.0_DP
!
    call generalSolution_BoundaryCoefficients(w9,a9,b9,c9)
    end subroutine inner_Const9
!=============================================================================
    subroutine inner_CartesianSelect(p,f)
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    call inner_Cartesian(kSelect,p,f)
```

```
    end subroutine inner_CartesianSelect
!================================================================================
    subroutine inner_Cartesian(k,p,f)
    integer,                  intent(in)  :: k
    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 inner_Polar(k,r,t,pr,vr,vt,st)
    call PolarToCartesian(t,vr,vt,vx,vy)
    f = (/ pr, vx, vy, st /)
    end subroutine inner_Cartesian
!================================================================================
    subroutine inner_PolarSelect(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    call inner_Polar(kSelect,r,t,pr,vr,vt,st)
    end subroutine inner_PolarSelect
!================================================================================
    subroutine inner_Polar(k,r,t,pr,vr,vt,st)
    implicit none
    integer,  intent(in)  :: k
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    select case (k)
        case(1)      ; call inner_Polar1(r,t,pr,vr,vt,st)
        case(2)      ; call inner_Polar2(r,t,pr,vr,vt,st)
        case(3)      ; call inner_Polar3(r,t,pr,vr,vt,st)
        case default ; pr = 0.0_DP ; vr = 0.0_DP ; vt = 0.0_DP ; st = 0.0_DP
    end select
    end subroutine inner_Polar
!================================================================================
    subroutine inner_Polar1(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    real(DP)              :: pr1, vr1, vt1, st1
    real(DP)              :: pr2, vr2, vt2, st2
    real(DP)              :: pr3, vr3, vt3, st3
    real(DP)              :: pr4, vr4, vt4, st4
    call generalSolution_Polar(w1,a1,b1,c1,r,t,pr1,vr1,vt1,st1)
    call generalSolution_Polar(w2,a2,b2,c2,r,t,pr2,vr2,vt2,st2)
    call generalSolution_Polar(w3,a3,b3,c3,r,t,pr3,vr3,vt3,st3)
    call generalSolution_Polar(w4,a4,b4,c4,r,t,pr4,vr4,vt4,st4)
    st = st1 + st2 + st3 + st4
    pr = pr1 + pr2 + pr3 + pr4
    vr = vr1 + vr2 + vr3 + vr4
    vt = vt1 + vt2 + vt3 + vt4
    end subroutine inner_Polar1
!================================================================================
    subroutine inner_Polar2(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    real(DP)              :: pr5, vr5, vt5, st5
    real(DP)              :: pr6, vr6, vt6, st6
    real(DP)              :: pr7, vr7, vt7, st7
```

```
    call generalSolution_Polar(w5,a5,b5,c5,r,t,pr5,vr5,vt5,st5)
    call generalSolution_Polar(w6,a6,b6,c6,r,t,pr6,vr6,vt6,st6)
    call generalSolution_Polar(w7,a7,b7,c7,r,t,pr7,vr7,vt7,st7)
    st = st5 + st6 + st7
    pr = pr5 + pr6 + pr7
    vr = vr5 + vr6 + vr7
    vt = vt5 + vt6 + vt7
    end subroutine inner_Polar2
!=============================================================================
    subroutine inner_Polar3(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    real(DP)              :: pr8, vr8, vt8, st8
    real(DP)              :: pr9, vr9, vt9, st9
    call generalSolution_Polar(w8,a8,b8,c8,r,t,pr8,vr8,vt8,st8)
    call generalSolution_Polar(w9,a9,b9,c9,r,t,pr9,vr9,vt9,st9)
    st = st8 + st9
    pr = pr8 + pr9
    vr = vr8 + vr9
    vt = vt8 + vt9
    end subroutine inner_Polar3
!=============================================================================
    subroutine inner_Error(k,r,vr,vt)
    implicit none
    integer,  intent(in)  :: k
    real(DP), intent(in)  :: r
    real(DP), intent(out) :: vr, vt
    select case (k)
        case(1)      ; call inner_Error1(r,vr,vt)
        case(2)      ; call inner_Error2(r,vr,vt)
        case(3)      ; call inner_Error3(r,vr,vt)
        case default ; vr = 0.0_DP ; vt = 0.0_DP
    end select
    end subroutine inner_Error
!=============================================================================
    subroutine inner_Error1(r,vr,vt)
    implicit none
    real(DP), intent(in)  :: r
    real(DP), intent(out) :: vr, vt
    real(DP)              :: c2, c1, c0
    c2 = -0.00048248182687_DP
    c1 =  0.00066168936256_DP
    c0 = -0.00018905410359_DP
    vr = (c2 * log(r)**2 + c1 * log(r) + c0) * r**3.5_DP
    vt = 0.0_DP
    end subroutine inner_Error1
!=============================================================================
    subroutine inner_Error2(r,vr,vt)
    implicit none
    real(DP), intent(in)  :: r
    real(DP), intent(out) :: vr, vt
    real(DP)              :: c1, c0
    c1 = -0.00909456817668_DP
    c0 =  0.00259844805048_DP
    vr = (c1 * log(r) + c0) * r**3.5_DP
    vt = 0.0_DP
    end subroutine inner_Error2
!=============================================================================
```

```
    subroutine inner_Error3(r,vr,vt)
    implicit none
    real(DP), intent(in)  :: r
    real(DP), intent(out) :: vr, vt
    real(DP)              :: c0
    c0 = -1.0_DP / 7.0_DP
    vr = c0 * r**3.5_DP
    vt = 0.0_DP
    end subroutine inner_Error3
!==========================================================================
    subroutine inner_Check(k)
    implicit none
    character(len=1)         :: kChar
    integer, intent(in)      :: k
    integer                  :: i, iMax
    real(DP)                 :: r, rMin, rMax
    real(DP)                 :: t1, pr1, vr1, vt1, st1
    real(DP)                 :: t2, pr2, vr2, vt2, st2
    real(DP)                 :: er1, er2, et1, et2
    rMin = 1.0e-03_DP
    rMax = 1.0e+01_DP
    iMax = 100
    t1 = 0.0_DP
    t2 = pi
    call numChar(k,kChar)
    kSelect = k
    call stokesEq2d_Cartesian(inner_CartesianSelect,'inner' // kChar // 'c.txt')
    call stokesEq2d_Polar(inner_PolarSelect,'inner' // kChar // 'p.txt')
    open (unit = 30, file = 'inner' // kChar // '.txt')
    do i = 0, iMax
        r = exp(log(rMin) + dble(i) / dble(iMax) * log(rMax / rMin))
        call inner_Polar(k,r,t1,pr1,vr1,vt1,st1)
        call inner_Polar(k,r,t2,pr2,vr2,vt2,st2)
        er1 = vr1
        et1 = vt1 + pr1
        call inner_Error(k,r,er2,et2)
        write (30,100) r, st1, pr1, vt1, vr1, pr2, vr2, vt2, er1, er2, et1, et2
    end do
    close (30)
100 format (e11.4, 6x, e11.4, 2(5x, 3(1x, e11.4)), 2(5x, d11.4, 1x, d11.4))
    end subroutine inner_Check
!==========================================================================
    subroutine inner_Setup
    implicit none
    call inner_Const
    call inner_Check(1)
    call inner_Check(2)
    call inner_Check(3)
    call inner_Latex
    end subroutine inner_Setup
!==========================================================================
end module mod_inner
!##########################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_outer
use mod_constants
use mod_utilities
use mod_stokesEq2d
```

```
implicit none
private
public :: outer_Setup
public :: outer_Polar
public :: outer_Cartesian
integer :: kSelect
contains
!=============================================================================
    subroutine outer_Build1_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = 0.0_DP
    end subroutine outer_Build1_shear
!=============================================================================
    subroutine outer_Build2_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = -2.0_DP / x**2
    end subroutine outer_Build2_shear
!=============================================================================
    subroutine outer_Build3_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = 0.0_DP
    end subroutine outer_Build3_shear
!=============================================================================
    subroutine outer_Build4_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = -4.0_DP / (pi * x**2)
    end subroutine outer_Build4_shear
!=============================================================================
    subroutine outer_Build5_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = -4.0_DP / x**3
    end subroutine outer_Build5_shear
!=============================================================================
    subroutine outer_Build6_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = (11.0_DP / 3.0_DP) / x**3
    end subroutine outer_Build6_shear
!=============================================================================
    subroutine outer_Build7_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = 2.0_DP / x**3
    end subroutine outer_Build7_shear
!=============================================================================
    subroutine outer_Build8_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
    s = 0.0_DP
    end subroutine outer_Build8_shear
!=============================================================================
    subroutine outer_Build9_shear(x,s)
    real(DP), intent(in)  :: x
    real(DP), intent(out) :: s
```

```
    s = (4.0_DP * log(x) + 5.0_DP / 3.0_DP) / (pi * x**3)
    end subroutine outer_Build9_shear
!==============================================================================
        subroutine outer_Build1(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: th, rh, x, y
        real(DP)                              :: pr, vx, vy, st
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build1_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            th = angle(r)
            pr = 1.0d0 - 2.0d0 * x / (pi * rh**2)
            vx = y**2 / (pi * rh**2)
            vy = th / pi - 1.0d0 - x * y / (pi * rh**2)
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build1
!==============================================================================
        subroutine outer_Build2(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: th, rh, x, y
        real(DP)                              :: pr, pr1, pr2
        real(DP)                              :: vx, vx1, vx2
        real(DP)                              :: vy, vy1, vy2
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build2_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            th = angle(r)
!
            pr1 = (2.0d0 * y**2 - 2.0d0 * x**2) / pi
            pr2 = (2.0d0 * x**2 - 4.0d0 * x * y * th - 2.0d0 * y**2) / pi
!
            vx1 = 2.0d0 * x * y**2 / pi
            vx2 = -x**2 * y * (1.0d0 + th / pi) - 2.0d0 * x * y**2 / pi &
                  - y**3 * (1.0d0 - th / pi)
!
            vy1 = 2.0d0 * y**3 / pi
            vy2 = x**3 * (1.0d0 - th / pi) + x**2 * y / pi &
                  + x * y**2 * (1.0d0 - 3.0d0 * th / pi) - y**3 / pi
!
            pr = (pr1 * log(rh) + pr2) / rh**4
            vx = (vx1 * log(rh) + vx2) / rh**4
            vy = (vy1 * log(rh) + vy2) / rh**4
            f = (/ pr, vx, vy, st /)
```

```
        end if
        end subroutine outer_Build2
!=============================================================================
        subroutine outer_Build3(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: rh, x, y
        real(DP)                              :: pr, vx, vy
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build3_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            pr = (2.0d0 * x**2 - 2.0d0 * y**2) / (pi * rh**4)
            vx = -2.0d0 * x * y**2 / (pi * rh**4)
            vy = -2.0d0     * y**3 / (pi * rh**4)
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build3
!=============================================================================
        subroutine outer_Build4(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: rh, x, y
        real(DP)                              :: pr, vx, vy
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build4_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            pr = -8.0d0 * x * y    / (pi * rh**4)
            vx = -4.0d0 * x**2 * y / (pi * rh**4)
            vy = -4.0d0 * y**2 * x / (pi * rh**4)
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build4
!=============================================================================
        subroutine outer_Build5(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: th, rh, x, y
        real(DP)                              :: pr, pr1, pr2
        real(DP)                              :: vx, vx1, vx2
        real(DP)                              :: vy, vy1, vy2
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build5_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
```

```
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            th = angle(r)
!
            pr1 = 4.0d0 * x * (3.0d0 * y**2 - x**2) / pi
            pr2 = 5.0d0 * x * (3.0d0 * y**2 - x**2) / (3.0d0 * pi) &
                  + 4.0d0 * y * th * (y**2 - 3.0d0 * x**2) / pi
!
            vx1 = 2.0d0 * y**2 * (3.0d0 * x**2 - y**2) / pi
            vx2 = -2.0d0 * x**3 * y * (1.0d0 + th / pi) + 5.0d0 * x**2 * y**2 / (2.0d0 * pi) &
                  - 2.0d0 * x * y**3 * (1.0d0 - 3.0d0 * th / pi) - 5.0d0 * y**4 / (6.0d0 * pi)
!
            vy1 = 8.0d0 * x * y**3 / pi
            vy2 = x**4 * (1.0d0 - th / pi) + x**3 * y / pi - 6.0d0 * x**2 * y**2 * th / pi &
                  + 13.0d0 * x * y**3 / (3.0d0 * pi) - y**4 * (1.0d0 - 3.0d0 * th / pi)
!
            pr = (pr1 * log(rh) + pr2) / rh**6
            vx = (vx1 * log(rh) + vx2) / rh**6
            vy = (vy1 * log(rh) + vy2) / rh**6
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build5
!===============================================================================
        subroutine outer_Build6(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: th, rh, x, y
        real(DP)                              :: pr, pr1, pr2, pr3
        real(DP)                              :: vx, vx1, vx2, vx3
        real(DP)                              :: vy, vy1, vy2, vy3
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build6_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            th = angle(r)
!
            pr1 = x * (6.0d0 * y**2 - 2.0d0 * x**2) / pi
            pr2 = y * (12.0d0 * x**2 - 4.0d0 * y**2) * (1.0d0 - th / pi)       &
                  + x * (2.0d0 * x**2 - 6.0d0 * y**2) / pi
            pr3 = x**3 * (1.0d0 / pi - 4.0d0 * th + 2.0d0 * th**2 / pi)        &
                  + x**2 * y * (5.0d0 + 6.0d0 * th / pi)                       &
                  - x * y**2 * (3.0d0 / pi - 12.0d0 * th + 6.0d0 * th**2 / pi) &
                  - y**3 * (5.0d0 / 3.0d0 + 2.0d0 * th / pi)
!
            vx1 = y**2 * (3.0d0 * x**2 - y**2) / pi
            vx2 = x * y * (2.0d0 * x**2 - 6.0d0 * y**2) * (1.0d0 - th / pi)                &
                  + y**2 * (y**2 - 3.0d0 * x**2) / pi
            vx3 = x**3 * y * (8.0d0 / 3.0d0 + th / pi)                                     &
                  - x**2 * y**2 * (3.0d0 / (2.0d0 * pi) - 6.0d0 * th + 3.0d0 * th**2 / pi) &
                  - x * y**3 * (2.0d0 / 3.0d0 + 3.0d0 * th / pi)                           &
```

```
                + y**4 * (1.0d0 / (2.0d0 * pi) - 2.0d0 * th + th**2 / pi)
!
            vy1 = 4.0d0 * x * y**3 / pi
            vy2 =  (x**4 + 6.0d0 * x**2 * y**2 - 3.0d0 * y**4) * (1.0d0 - th / pi) &
                   + x * y * (x**2 - 3.0d0 * y**2) / pi
            vy3 = x**2 * y**2 * (5.0d0 / 2.0d0 + 3.0d0 * th / pi)                  &
                  - x * y**3 * (2.0d0 / pi - 8.0d0 * th + 4.0d0 * th**2 / pi)      &
                  - y**4 * (5.0d0 / 6.0d0 + th / pi)
!
            pr = (pr1 * log(rh)**2 + pr2 * log(rh) + pr3) / rh**6
            vx = (vx1 * log(rh)**2 + vx2 * log(rh) + vx3) / rh**6
            vy = (vy1 * log(rh)**2 + vy2 * log(rh) + vy3) / rh**6
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build6
!=======================================================================
        subroutine outer_Build7(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: rh, x, y
        real(DP)                              :: pr, vx, vy
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build7_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            pr = y * (6.0d0 * x**2 - 2.0d0 * y**2) / rh**6
            vx = 2.0d0 * x * y * (x**2 - y**2) / rh**6
            vy = y**2 * (3.0d0 * x**2 - y**2) / rh**6
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build7
!=======================================================================
        subroutine outer_Build8(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: rh, x, y
        real(DP)                              :: pr, vx, vy
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build8_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            pr = x * (2.0d0 * x**2 - 6.0d0 * y**2) / rh**6
            vx = y**2 * (y**2 - 3.0d0 * x**2) / rh**6
            vy = -4.0d0 * x * y**3 / rh**6
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build8
```

```
!========================================================================
        subroutine outer_Build9(r,f)
        implicit none
        real(DP), dimension(2),   intent(in)  :: r
        real(DP), dimension(0:3), intent(out) :: f
        real(DP)                              :: st
        real(DP)                              :: th, rh, x, y
        real(DP)                              :: pr, pr1, pr2
        real(DP)                              :: vx, vx1, vx2
        real(DP)                              :: vy, vy1, vy2
        real(DP),                 parameter   :: tol = 1.0d-9
        call outer_Build8_shear(abs(r(1)),st)
        rh = sqrt(dot_product(r,r))
        if (rh < tol) then
            f = 0.0d0
        else
            x = r(1)
            y = r(2)
            th = angle(r)
!
            pr1 = 4.0d0 * y * (3.0d0 * x**2 - y**2) / pi
            pr2 = 5.0d0 * y * (3.0d0 * x**2 - y**2) / (3.0d0 * pi) &
                  + 4.0d0 * (th / pi) * x * (3.0d0 * y**2 - x**2)
!
            vx1 = 4.0d0 * x * y * (x**2 - y**2) / pi
            vx2 = 2.0d0 * x * y * (4.0d0 * x**2 -y**2) / (3.0d0 * pi) &
                  - (th / pi) * (x**4 - 6.0d0 * x**2 * y**2 + y**4)   &
                  + x**4 - y**4
!
            vy1 = 2.0d0 * y**2 * (3.0d0 * x**2 - y**2) / pi
            vy2 =   x**2 * y * (5.0d0 * y / (2.0d0 * pi) + 2.0d0 * x) &
                  - y**3     * (5.0d0 * y / (6.0d0 * pi) - 2.0d0 * x) &
                  + (th / pi) * 2.0d0 * x * y * (3.0d0 * y**2 - x**2)
!
            pr = (pr1 * log(rh) + pr2) / rh**6
            vx = (vx1 * log(rh) + vx2) / rh**6
            vy = (vy1 * log(rh) + vy2) / rh**6
            f = (/ pr, vx, vy, st /)
        end if
        end subroutine outer_Build9
!========================================================================
    subroutine outer_Cartesian1(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f1, f2, f4, f5, f6, f7
    real(DP)                              :: c, a4, a7
    call outer_Build1(r,f1)
    call outer_Build2(r,f2)
    call outer_Build4(r,f4)
    call outer_Build5(r,f5)
    call outer_Build6(r,f6)
    call outer_Build7(r,f7)
    a4 =      pi / 2.0_DP
    a7 = 23.0_DP / 6.0_DP
    c = 2.0d0 / pi
    f = f1 + c * (f2 - a4 * f4) + c**2 * (f6 - f5 - a7 * f7)
    end subroutine outer_Cartesian1
!========================================================================
```

```
    subroutine outer_Cartesian1b(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f1, f2, f4
    real(DP)                              :: c, a4
    call outer_Build1(r,f1)
    call outer_Build2(r,f2)
    call outer_Build4(r,f4)
    a4 = pi / 2.0_DP
    c = 2.0d0 / pi
    f = f1 + c * (f2 - a4 * f4)
    end subroutine outer_Cartesian1b
!=============================================================================
    subroutine outer_Cartesian2(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f3, f5, f7
    real(DP)                              :: c, a7
    call outer_Build3(r,f3)
    call outer_Build5(r,f5)
    call outer_Build7(r,f7)
    a7 = 2.0_DP
    c = 2.0d0 / pi
    f = f3 - c * (f5 + a7 * f7)
    end subroutine outer_Cartesian2
!=============================================================================
    subroutine outer_Cartesian2b(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f3
    call outer_Build3(r,f3)
    f = f3
    end subroutine outer_Cartesian2b
!=============================================================================
    subroutine outer_Cartesian3(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f8
    call outer_Build8(r,f8)
    f = f8
    end subroutine outer_Cartesian3
!=============================================================================
    subroutine outer_shear(k,x,s,t)
    implicit none
    integer                             :: i, j
    integer,  intent(in)                :: k
    real(DP), intent(in)                :: x
    real(DP), intent(out)               :: s, t
    real(DP), dimension(2)              :: p
    real(DP), dimension(0:3,-1:1,0:2) :: f
    real(DP)                            :: h, vx_y, vy_x
    h = x * 1.0e-6_DP
    do i = -1, 1
    do j =  0, 2
        p(1) = dble(i) * h + x
```

```
        p(2) = dble(j) * h
        call outer_Cartesian(k,p,f(:,i,j))
    end do
    end do
    vx_y = (-3.0d0 * f(1,0,0) + 4.0d0 * f(1,0,1) - f(1,0,2)) / (2.0_DP * h)
    vy_x = (f(2,1,0) - f(2,-1,0)) / (2.0_DP * h)
    s = f(3,0,0)
    t = vx_y + vy_x
    end subroutine outer_shear
!=======================================================================
    subroutine outer_CartesianSelect(p,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    call outer_Cartesian(kSelect,p,f)
    end subroutine outer_CartesianSelect
!=======================================================================
    subroutine outer_Cartesian(k,p,f)
    implicit none
    integer,                  intent(in)  :: k
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    if (k == 1) then
        call outer_Cartesian1(p,f)
    else if (k == -1) then
        call outer_Cartesian1b(p,f)
    else if (k == 2) then
        call outer_Cartesian2(p,f)
    else if (k == -2) then
        call outer_Cartesian2b(p,f)
    else if (k == 3) then
        call outer_Cartesian3(p,f)
    else
        print*, '[k] out of range in OUTER_Cartesian'
        stop
    end if
    end subroutine outer_Cartesian
!=======================================================================
    subroutine outer_PolarSelect(r,t,pr,vr,vt,st)
    implicit none
    real(DP), intent(in)  :: r, t
    real(DP), intent(out) :: pr, vr, vt, st
    call outer_Polar(kSelect,r,t,pr,vr,vt,st)
    end subroutine outer_PolarSelect
!=======================================================================
    subroutine outer_Polar(k,r,t,pr,vr,vt,st)
    implicit none
    integer,  intent(in)     :: k
    real(DP), intent(in)     :: r, t
    real(DP), intent(out)    :: pr, vr, vt, st
    real(DP), dimension(2)   :: p
    real(DP), dimension(0:3) :: f
    p = r * (/ cos(t), sin(t) /)
    call outer_Cartesian(k,p,f)
    pr = f(0)
    st = f(3)
    call CartesianToPolar(t,f(1),f(2),vr,vt)
    end subroutine outer_Polar
!=======================================================================
```

```
    subroutine outer_Check(k)
    implicit none
    integer, intent(in) :: k
    character(len=1)    :: kChar
    kSelect = k
    call numChar(k,kChar)
    call stokesEq2d_Cartesian(outer_CartesianSelect,'outer' // kChar // 'c.txt')
    call stokesEq2d_Polar(outer_PolarSelect,'outer' // kChar // 'p.txt')
    end subroutine outer_Check
!=============================================================================
    subroutine outer_BoundaryCondition
    implicit none
    integer                  :: i, iMax
    real(DP)                 :: x, xMin, xMax, s1, s2, s3, t1, t2, t3
    real(DP), dimension(2)   :: p, q
    real(DP), dimension(0:3) :: f1, f1b, f2, f2b, f3, g1, g2, g3
    open (unit = 21, file = 'outer.txt')
    open (unit = 22, file = 'shear.txt')
    xMin = 1.0e-03_DP
    xMax = 1.0e+02_DP
    iMax = 25
    write (21,100)
    write (22,200)
    do i = 0, iMax
        x = exp(log(xMin) + dble(i) / dble(iMax) * log(xMax / xMin))
        p = (/  x, 0.0_DP /)
        q = (/ -x, 0.0_DP /)
        call outer_Cartesian( 1,p,f1)
        call outer_Cartesian(-1,p,f1b)
        call outer_Cartesian( 2,p,f2)
        call outer_Cartesian(-2,p,f2b)
        call outer_Cartesian( 3,p,f3)
        call outer_Cartesian( 1,q,g1)
        call outer_Cartesian( 2,q,g2)
        call outer_Cartesian( 3,q,g3)
        call outer_shear(1,x,s1,t1)
        call outer_shear(2,x,s2,t2)
        call outer_shear(3,x,s3,t3)
        write(21,300) x, f1b(0), f1(2), f1(1), s1, g1(2), g1(1), &
                         f2b(0), f2(2), f2(1), s2, g2(2), g2(1), &
                         f3(0),  f3(2), f3(1), s3, g3(2), g3(1)
        write(22,400) x, s1, s2, s3, t1, t2, t3
    end do
    close(21)
    close(22)
100 format('Check outer asymptotic solutions for boundary conditions' /)
200 format('Check outer asymptotic solutions for shear stress' /)
300 format('x = ', d12.5, 5x, 'pr1 = ', d12.5, 3x, 'vt1 = ', d12.5, 3x,  &
                              'vr1 = ', d12.5, 3x, 'sh1 = ', d12.5, 3x,  &
                              'wt1 = ', d12.5, 3x, 'wr1 = ', d12.5, 5x / &
                         21x, 'pr2 = ', d12.5, 3x, 'vt2 = ', d12.5, 3x,  &
                              'vr2 = ', d12.5, 3x, 'sh2 = ', d12.5, 3x,  &
                              'wt2 = ', d12.5, 3x, 'wr2 = ', d12.5, 5x / &
                         21x, 'pr3 = ', d12.5, 3x, 'vt3 = ', d12.5, 3x,  &
                              'vr3 = ', d12.5, 3x, 'sh3 = ', d12.5, 3x,  &
                              'wt3 = ', d12.5, 3x, 'wr3 = ', d12.5, 5x /)
400 format('x = ', d12.5, 5x, &
           's1 = ', d12.5, 3x, 's2 = ', d12.5, 3x, 's3 = ', d12.5 / 21x, &
           't1 = ', d12.5, 3x, 't2 = ', d12.5, 3x, 't3 = ', d12.5 /)
```

```
    end subroutine outer_BoundaryCondition
!==========================================================================
    subroutine outer_Setup
    implicit none
    call outer_Check(1)
    call outer_check(2)
    call outer_Check(3)
    call outer_BoundaryCondition
    end subroutine outer_Setup
!==========================================================================
    subroutine outer_NoSlipCartesian1(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f1, f2, f5, f6
    real(DP)                              :: c
    call outer_Build1(r,f1)
    call outer_Build2(r,f2)
    call outer_Build5(r,f5)
    call outer_Build6(r,f6)
    c = 2.0d0 / pi
    f = f1 + c * f2 + c**2 * (f6 - f5)
    end subroutine outer_NoSlipCartesian1
!==========================================================================
    subroutine outer_NoSlipCartesian1b(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f1, f2
    real(DP)                              :: c
    call outer_Build1(r,f1)
    call outer_Build2(r,f2)
    c = 2.0d0 / pi
    f = f1 + c * f2
    end subroutine outer_NoSlipCartesian1b
!==========================================================================
    subroutine outer_NoSlipCartesian2(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f3, f5
    real(DP)                              :: c
    call outer_Build3(r,f3)
    call outer_Build5(r,f5)
    c = 2.0d0 / pi
    f = f3 - c * f5
    end subroutine outer_NoSlipCartesian2
!==========================================================================
    subroutine outer_NoSlipCartesian2b(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f3
    call outer_Build3(r,f3)
    f = f3
    end subroutine outer_NoSlipCartesian2b
!==========================================================================
    subroutine outer_NoSlipCartesian3(r,f)
    implicit none
```

```
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f4
    call outer_Build4(r,f4)
    f = f4
    end subroutine outer_NoSlipCartesian3
!=============================================================================
    subroutine outer_NoSlipCartesian4(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f7
    call outer_Build7(r,f7)
    f = f7
    end subroutine outer_NoSlipCartesian4
!=============================================================================
    subroutine outer_NoSlipCartesian5(r,f)
    implicit none
    real(DP), dimension(2),   intent(in)  :: r
    real(DP), dimension(0:3), intent(out) :: f
    real(DP), dimension(0:3)              :: f8
    call outer_Build8(r,f8)
    f = f8
    end subroutine outer_NoSlipCartesian5
!=============================================================================
end module mod_outer
!#############################################################################
! Copyright 2011 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_Polar
public :: genSol_Cartesian
contains
!=============================================================================
    subroutine genSol_Cartesian(w,a,b,c,p,f)
    real(DP),                 intent(in)  :: w
    real(DP), dimension(4),   intent(in)  :: a, b, c
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:2), intent(out) :: f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(p,p))
    t = angle(p)
    call genSol_Polar(w,a,b,c,r,t,pr,vr,vt,st)
    call PolarToCartesian (t,vr,vt,vx,vy)
    f = (/ pr, vx, vy /)
    end subroutine genSol_Cartesian
!=============================================================================
    subroutine gensol_Polar(w,a,b,c,r,t,pr,vr,vt,st)
    implicit none
    real(DP),               intent(in)  :: r, t, w
    real(DP), dimension(4), intent(in)  :: a, b, c
    real(DP),               intent(out) :: pr, vr, vt, st
```

```
    real(DP), dimension(10)             :: d
    real(DP),               parameter   :: tol = 1.0e-6_DP
    if (abs(w - 1.0_DP) < tol) then
        d(1: 4) = a
        d(5: 6) = b(1:2)
        d(7:10) = c
        call generalSolutionPlus1_Polar(d,r,t,pr,vr,vt,st)
    else if(abs(w + 1.0_DP) < tol) then
        call generalSolutionMinus1_Polar(a,b,c,r,t,pr,vr,vt,st)
    else
        call generalSolution_Polar(w,a,b,c,r,t,pr,vr,vt,st)
    end if
    end subroutine gensol_Polar
!=========================================================================
end module mod_genSol
!#########################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_asymptotic
use mod_constants
use mod_inner
use mod_outer
implicit none
private
public :: asymptotic_Setup
public :: asymptotic_Polar
public :: asymptotic_Cartesian
contains
!=========================================================================
    subroutine asymptotic_Setup
    implicit none
    call inner_Setup
    call outer_Setup
    end subroutine asymptotic_Setup
!=========================================================================
    subroutine asymptotic_Polar(region,k,r,t,pr,vr,vt)
    implicit none
    character(len=*), intent(in)  :: region
    integer,          intent(in)  :: k
    real(DP),         intent(in)  :: r, t
    real(DP),         intent(out) :: pr, vr, vt
    real(DP)                      :: st
    if (region(1:1) == 'i' .or. region(1:1) == 'I') then
        call inner_Polar(k,r,t,pr,vr,vt,st)
    else
        call outer_Polar(k,r,t,pr,vr,vt,st)
    end if
    end subroutine asymptotic_Polar
!=========================================================================
    subroutine asymptotic_Cartesian(region,k,p,f)
    character(len=*),         intent(in)  :: region
    integer,                  intent(in)  :: k
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:2), intent(out) :: f
    real(DP), dimension(0:3)              :: g
    if (region(1:1) == 'i' .or. region(1:1) == 'I') then
        call inner_Cartesian(k,p,g)
    else
        call outer_Cartesian(k,p,g)
```

```
    end if
    f = g(0:2)
    end subroutine asymptotic_Cartesian
!=======================================================================================
end module mod_asymptotic
!#######################################################################################
! Copyright 2011 by Ludwig C. Nitsche
!-----------------------------------
module mod_membrane
use mod_constants
use mod_utilities
use mod_stokesEq2d
use mod_interp1
use mod_polyFit
use mod_asymptotic
implicit none
private
public :: membrane_Setup
public :: membrane_Plot1
public :: membrane_Plot2
public :: membrane_Polar
public :: membrane_Cartesian
real(DP)                          :: sav_h
real(DP)                          :: sav_rInner
real(DP)                          :: sav_rOuter
real(DP)                          :: sav_cInner
real(DP),          dimension(3) :: sav_cOuter
real(DP),          dimension(3) :: sav_r
type(typ_interp1), dimension(3) :: sav_pr
type(typ_interp1), dimension(3) :: sav_vr
type(typ_interp1), dimension(3) :: sav_vt
contains
!=======================================================================================
    subroutine membrane_Setup
    implicit none
    real(DP), parameter :: factor = 1.0e-3_DP
    sav_rInner = 1.0e-2_DP
    sav_rOuter = 30.0_DP
    sav_cInner = 0.8218_DP
    sav_cOuter = (/ 1.0_DP, -0.57190_DP, 1.68952_DP /)
    sav_h = factor * log(sav_rOuter / sav_rInner)
    call asymptotic_Setup
    call membrane_Splines
    call stokesEq2d_Cartesian(membrane_CartesianCheck,'membrane_c.txt')
    call stokesEq2d_Polar(membrane_PolarCheck,'membrane_p.txt')
    end subroutine membrane_Setup
!=======================================================================================
    subroutine membrane_Splines
    implicit none
    character(len=1)            :: iChar
    integer                     :: i
    real(DP), dimension(0:12)   :: t
    real(DP), dimension(3,0:12) :: pr, vr, vt
    sav_r(1) = 0.10014066E+00_DP
    sav_r(2) = 0.10028151E+01_DP
    sav_r(3) = 0.10042256E+02_DP
    pr(1,:) = (/  0.24034749E+00,  0.58623974E+00,  0.92406968E+00, &
                  0.12488095E+01,  0.15555038E+01,  0.18393588E+01, &
                  0.20958283E+01,  0.23207067E+01,  0.25102263E+01, &
```

```
                0.26611537E+01,  0.27708766E+01,  0.28374789E+01, &
                0.28600447E+01 /)
    pr(2,:) = (/  0.60502392E+00,  0.70503303E+00,  0.80357168E+00, &
                0.89893280E+00,  0.98946048E+00,  0.10735641E+01, &
                0.11497556E+01,  0.12166815E+01,  0.12731492E+01, &
                0.13181511E+01,  0.13508829E+01,  0.13707591E+01, &
                0.13773558E+01 /)
    pr(3,:) = (/  0.93080993E+00,  0.93935786E+00,  0.95087752E+00, &
                0.96436617E+00,  0.97890646E+00,  0.99367854E+00, &
                0.10079582E+01,  0.10211159E+01,  0.10326164E+01, &
                0.10420201E+01,  0.10489833E+01,  0.10532598E+01, &
                0.10547132E+01 /)
    vr(1,:) = (/  0.33544433E-05, -0.33007402E-01, -0.61352465E-01, &
               -0.81937206E-01, -0.92901029E-01, -0.93808159E-01, &
               -0.85628940E-01, -0.70530966E-01, -0.51520503E-01, &
               -0.31988899E-01, -0.15226010E-01, -0.39629126E-02, &
               -0.28118735E-05 /)
    vr(2,:) = (/  0.28612046E-05, -0.10054043E+00, -0.17944847E+00, &
               -0.23133699E+00, -0.25441836E+00, -0.25031182E+00, &
               -0.22355164E+00, -0.18085486E+00, -0.13022453E+00, &
               -0.79975640E-01, -0.37775617E-01, -0.97883306E-02, &
               -0.46553093E-05 /)
    vr(3,:) = (/ -0.74566398E-05, -0.20078959E+00, -0.34698663E+00, &
               -0.43564806E+00, -0.46883407E+00, -0.45314847E+00, &
               -0.39890680E+00, -0.31901222E+00, -0.22764296E+00, &
               -0.13886548E+00, -0.65289259E-01, -0.16875334E-01, &
                0.61606197E-05 /)
    vt(1,:) = (/ -0.24036105E+00, -0.23376524E+00, -0.21507045E+00, &
               -0.18688005E+00, -0.15270768E+00, -0.11644026E+00, &
               -0.81790545E-01, -0.51811064E-01, -0.28526801E-01, &
               -0.12725610E-01, -0.39240008E-02, -0.50473488E-03, &
                0.20902193E-05 /)
    vt(2,:) = (/ -0.60498888E+00, -0.58543833E+00, -0.53260197E+00, &
               -0.45646421E+00, -0.36770762E+00, -0.27657360E+00, &
               -0.19187681E+00, -0.12024160E+00, -0.65611478E-01, &
               -0.29060659E-01, -0.89117605E-02, -0.11375961E-02, &
                0.38910505E-05 /)
    vt(3,:) = (/ -0.93083934E+00, -0.89930845E+00, -0.81536025E+00, &
               -0.69598401E+00, -0.55837010E+00, -0.41838072E+00, &
               -0.28927487E+00, -0.18075634E+00, -0.98403526E-01, &
               -0.43508946E-01, -0.13327032E-01, -0.16995165E-02, &
               -0.14709556E-04 /)
    vr(:, 0) = 0.0_DP
    vt(:,12) = 0.0_DP
    do i = 0, 12
        t(i) = pi * dble(i) / 12.0_DP
    end do
    do i = 1, 3
        call sav_pr(i)%setData(t,pr(i,:))
        call sav_vr(i)%setData(t,vr(i,:))
        call sav_vt(i)%setData(t,vt(i,:))
        call numChar(i,iChar)
        call sav_pr(i)%tabulate('spline_pr' // iChar)
        call sav_vr(i)%tabulate('spline_vr' // iChar)
        call sav_vt(i)%tabulate('spline_vt' // iChar)
    end do
    end subroutine membrane_Splines
!=======================================================================
    subroutine membrane_Plot1
```

```
    implicit none
    integer                              :: m, mm, n, nn
    real(dp)                             :: rh, th, d1, d2, factor
    real(dp), dimension(:), allocatable :: pr, vr, vt
    d1 = 0.01_DP
    d2 = 30.0_DP
    factor = 5.0d0
    mm = nint(factor * log10(d2 / d1))
    nn = 12
    open (unit = 21, file = 'pr_vs_r.txt')
    open (unit = 22, file = 'vr_vs_r.txt')
    open (unit = 23, file = 'vt_vs_r.txt')
    allocate (pr(0:nn))
    allocate (vr(0:nn))
    allocate (vt(0:nn))
    do m = 0, mm
        rh = exp(log(d1) + dble(m) / dble(mm) * log(d2 / d1))
        do n = 0, nn
            th = pi * dble(n) / dble(nn)
            call membrane_Polar(rh,th,pr(n),vr(n),vt(n))
        end do
        write (21,100) m, rh, pr
        write (22,100) m, rh, vr
        write (23,100) m, rh, vt
    end do
    deallocate (pr)
    deallocate (vr)
    deallocate (vt)
    close (21)
    close (22)
    close (23)
100 format (i3, 1x, e15.8, 4x, 19(1x, e15.8))
    end subroutine membrane_Plot1
!=======================================================================
    subroutine membrane_Plot2(arg_r,arg_filename)
    implicit none
    character(len=*), intent(in) :: arg_filename
    real(DP),         intent(in) :: arg_r
    integer                      :: i, j, k
    real(DP), dimension(2)       :: p
    real(DP), dimension(0:2)     :: f
    open(unit = 30, file = arg_filename)
    k = 40
    do j =  0, k
    do i = -k, k
        p(1) = arg_r * dble(i) / dble(k)
        p(2) = arg_r * dble(j) / dble(k)
        call membrane_Cartesian(p,f)
        write(30,100) p,f
    end do
    end do
100 format(2(1x, f12.7), 5x, 3(1x, f12.7))
    end subroutine membrane_Plot2
!=======================================================================
    subroutine membrane_Polar(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    implicit none
    real(DP), intent(in)      :: arg_r, arg_t
    real(DP), intent(out)     :: arg_pr, arg_vr, arg_vt
    real(DP)                  :: u
```

```
    real(DP), dimension(-1:1) :: pr, vr, vt
    real(DP), dimension( 0:2) :: fpri, fvri, fvti
    real(DP), dimension( 0:2) :: fpro, fvro, fvto
    real(DP), dimension(3)    :: prData, vrData, vtData
    type(typ_polyfit)         :: poly
    integer                   :: i
    if (arg_r < sav_rInner) then
        call membrane_Inner(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    else if (arg_r > sav_rOuter) then
        call membrane_Outer(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    else
        do i = -1, 1
            u = log(sav_rInner) + sav_h * dble(i)
            call membrane_Inner(exp(u),arg_t,pr(i),vr(i),vt(i))
        end do
        call membrane_Deriv(log(pr),fpri)
        call membrane_Deriv(vr,fvri)
        call membrane_Deriv(vt,fvti)
        do i = -1, 1
            u = log(sav_rOuter) + sav_h * dble(i)
            call membrane_Outer(exp(u),arg_t,pr(i),vr(i),vt(i))
        end do
        call membrane_Deriv(log(pr),fpro)
        call membrane_Deriv(vr,fvro)
        call membrane_Deriv(vt,fvto)
        do i = 1, 3
            prData(i) = sav_pr(i)%spline2(arg_t,0)
            vrData(i) = sav_vr(i)%spline2(arg_t,0)
            vtData(i) = sav_vt(i)%spline2(arg_t,0)
        end do
        call poly%setEndPoints(log(sav_rInner),log(sav_rOuter))
        u = log(arg_r)
        call poly%setEndValues(fpri,fpro)
        call poly%setData(log(sav_r),log(prData))
        arg_pr = exp(poly%evaluate(u))
        call poly%setEndValues(fvri,fvro)
        call poly%setData(log(sav_r),vrData)
        arg_vr = poly%evaluate(u)
        call poly%setEndValues(fvti,fvto)
        call poly%setData(log(sav_r),vtData)
        arg_vt = poly%evaluate(u)
    end if
    end subroutine membrane_Polar
!=======================================================================
    subroutine membrane_Cartesian(p,f)
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:2), intent(out) :: f
    real(DP)                              :: r, t, pr, vr, vt, st, vx, vy
    r = sqrt(dot_product(p,p))
    t = angle(p)
    call membrane_Polar(r,t,pr,vr,vt)
    call PolarToCartesian(t,vr,vt,vx,vy)
    f = (/ pr, vx, vy /)
    end subroutine membrane_Cartesian
!=======================================================================
    subroutine membrane_PolarCheck(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 membrane_Polar(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    arg_st = 0.0_DP
    end subroutine membrane_PolarCheck
!=============================================================================
    subroutine membrane_CartesianCheck(p,f)
    real(DP), dimension(2),   intent(in)  :: p
    real(DP), dimension(0:3), intent(out) :: f
    f = 0.0_DP
    call membrane_Cartesian(p,f(0:2))
    end subroutine membrane_CartesianCheck
!=============================================================================
    subroutine membrane_Deriv(arg_f,arg_c)
    implicit none
    real(DP), dimension(-1:1), intent(in)  :: arg_f
    real(DP), dimension( 0:2), intent(out) :: arg_c
    arg_c(0) = arg_f(0)
    arg_c(1) = (arg_f(1) - arg_f(-1)) / (2.0_DP * sav_h)
    arg_c(2) = (arg_f(1) - 2.0_DP * arg_f(0) + arg_f(-1)) / sav_h**2
    end subroutine membrane_Deriv
!=============================================================================
    subroutine membrane_Inner(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_pr, arg_vr, arg_vt
    call asymptotic_Polar('inner',1,arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    arg_pr = sav_cInner * arg_pr
    arg_vr = sav_cInner * arg_vr
    arg_vt = sav_cInner * arg_vt
    end subroutine membrane_Inner
!=============================================================================
    subroutine membrane_Outer(arg_r,arg_t,arg_pr,arg_vr,arg_vt)
    implicit none
    real(DP), intent(in)  :: arg_r, arg_t
    real(DP), intent(out) :: arg_pr, arg_vr, arg_vt
    real(DP)              :: pr, vr, vt
    integer               :: k
    arg_pr = 0.0_DP
    arg_vr = 0.0_DP
    arg_vt = 0.0_DP
    do k = 1, 3
        call asymptotic_Polar('outer',k,arg_r,arg_t,pr,vr,vt)
        arg_pr = arg_pr + sav_cOuter(k) * pr
        arg_vr = arg_vr + sav_cOuter(k) * vr
        arg_vt = arg_vt + sav_cOuter(k) * vt
    end do
    end subroutine membrane_Outer
!=============================================================================
end module mod_membrane
!#############################################################################
```