E V E R H A R T − I N T E G R A T O R

by Angelo Graziosi

I N T R O D U C T I O N

This document contains a few examples of fortran programs "using" the Everhart's integrator method for numerical solution of systems of ordinary differential equations.

This method has proven very efficient in the calculation of the orbits of comets and, more generally, in the simulation of the gravitational interactions of a system of n−bodies.

Here we present an implementation in modern Fortran, changing some notation and conventions. For example, the GAUSS−RADAU spacings are represented not as array HS(1:8) but as array HS(0:7), which seems a more natural way, given the meaning of spacings within a sequence. Another change is to use H, HP, H2, HVAL for time steps and not T, TP, T2, TVAL.

We have broken the original routine in three routines: radau_on() (to initialize the integrator), ra15() (the integrator itself) and radau_off() (to close opened files and to recover allocated memory).

This implementation does not use old Fortran statements like GOTOs etc., but the most recent "allocatable", named loops and so on. It adds, also, the capability to save the solution in a binary file.

All this is in the Fortran module 'everhart_integrator.f90'.

We have tested this implementation writing the programs described in section 3 of the original Everhart's paper:

> E. Everhart, An Efficient Integrator That Use Gauss−Radau Spacings, in A. Carusi and G. B. Valsecchi − Dynamics of Comets: Their Origin and Evolution, 185−202. 1985 by D. Reidel Publishing Company.

One of the following examples, test_jsunp.f90, is an attempt to re−write the JSUNP.FOR program cited at the end of section 4 of the above paper. For the initial positions, we have used those found in

> Eckert, Brouwer, Clemence (1951),
> Coordinates of the Five Outer Planets 1653−2060,
> Astronom. Papers American Ephem. XII

while for initial velocities, we have computed them as numerical derivative. For details, see the source code of test_jsunp.f90 below.

Beside this program, we have added also the source code of close_encounters.f90. This program can simulate the gravitational interactions of n−bodies using the Everhart's integrator. It finds also the close encounters of two of them. The data are read from a cards file, close_encounters.cards, which is added too. A screen shot showing the Apophis orbit (in red), calculated up to 2070 with this program, can be found on this same web page . The orbits are drawn in perspective using the Fortran interface to BGI described elsewhere on this WEB site. As for test_jsunp.f90, all the details are found in the source code below.

A special thanks goes to G. Matarazzo who provided the original paper of Everhart.

This document has been created using EMACS (and some "friends" tools like ps2pdf, pdftk etc..).

```
!
! Fortran Interface to the Everhart Integrator Library
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! This is the 'everhart_integrator' module.
!
!
! A simple module which tries to re−implement in modern Fortran the
! Everhart's RADAU integrator.
!
! Ref. :
!
!   E. Everhart, An Efficient Integrator That Use Gauss−Radau Spacings,
!   in A. Carusi and G. B. Valsecchi − Dynamics of Comets: Their Origin and
!   Evolution, 185−202. 1985 by D. Reidel Publishing Company.
!

module everhart_integrator
  use kind_consts, only: DP
  implicit none
  private

  integer, parameter :: NOR = 15
  integer, parameter :: NSTEPS = 7, NCOEF = (NSTEPS*(NSTEPS−1))/2
  real(DP), parameter :: ZERO = 0, ONE = 1, HALF = ONE/2
  !
  ! These HS(:) values are the Gauss−Radau spacings, scaled to the
  ! range 0 to 1, for integrating to order 15. HS(0) == ZERO always.
  ! The sum of these H−values should be 3.7(3) = 3.733333... = 56/15
  ! (Viete formulas for the polynomial of degree 7 whose root are
  ! HS(1:NSTEPS)−values)
  !
  real(DP), parameter :: HS(0:NSTEPS) = [ ZERO, 0.05626256053692215_DP, &
       0.18024069173689236_DP, 0.35262471711316964_DP, &
       0.54715362633055538_DP, 0.73421017721541053_DP, &
       0.88532094683909577_DP, 0.97752061356128750_DP ]

  abstract interface
     subroutine ode_field(t,y,yp,f)
       use kind_consts, only: DP
       real(DP), intent(in) :: t, y(:), yp(:)
       real(DP), intent(out) :: f(:)
     end subroutine ode_field
  end interface

  integer :: nv, ll, nclass, log_unit, data_unit
  logical :: npq, ncl, nes, debug_flag, save_data_flag
  ! WC, UC, WC0, SS, C, D, R are, really, CONSTANTS
  real(DP) :: WC(NSTEPS), UC(NSTEPS), WC0, SS
  real(DP) :: C(NCOEF), D(NCOEF), R(NCOEF)
  !
  ! The workspace would be NV x 3*NSTEPS+4 = NV x 3*7+4 −−> w(NV,25)
  ! (BSG uses w(NEQ,36), being NEQ the number of equations of 1st order)
  !
  real(DP), allocatable :: f0(:), fj(:), y(:), yp(:)
  real(DP), allocatable :: b(:,:), g(:,:), e(:,:)

  public :: radau_on, ra15, radau_off

contains
  subroutine radau_on(nv0,ll0,nclass0,debug_flag0,save_data_flag0)
    integer, intent(in) :: nv0, ll0, nclass0
    logical, intent(in), optional :: debug_flag0, save_data_flag0

    integer, parameter :: NW(0:NSTEPS)= [ 0, 0, 1, 3, 6, 10, 15, 21 ]
    real(DP) :: temp
    integer :: l, la, lb, lc, ld, le, k, ierr

    ! Work space allocation
    allocate(b(NSTEPS,nv0),stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Allocation failure for B(:,:). Exiting...'
```

```
      stop
   end if
   allocate(g(NSTEPS,nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for G(:,:). Exiting...'
      stop
   end if
   allocate(e(NSTEPS,nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for E(:,:). Exiting...'
      stop
   end if
   allocate(f0(nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for F0(:). Exiting...'
      stop
   end if
   allocate(fj(nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for FJ(:). Exiting...'
      stop
   end if
   allocate(y(nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for Y(:). Exiting...'
      stop
   end if
   allocate(yp(nv0),stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Allocation failure for YP(:). Exiting...'
      stop
   end if

   ! Input data initialization
   nv = nv0
   ll = ll0
   nclass = nclass0
   !
   ! Global logical data initialization
   !
   ! NCL is a flag which says if the equations are of 1st order (.TRUE.) or
   ! of 2nd order (.FALSE.) :
   !
   !   y' = F(t,y),    NCL == .TRUE.
   !   y" = F(t,y),    NCL == .FALSE.
   !   y" = F(t,y,y'), NCL == .FALSE.
   !
   ! NPQ is a flag which says if the equations are of 2nd order general
   ! (.FALSE.) or NOT 2nd order general (.TRUE.), i.e. of 1st order or
   ! 2nd order Special (without y') :
   !
   ! NCLASS ==  1, NPQ == .TRUE.
   ! NCLASS == -2, NPQ == .TRUE.
   ! NCLASS ==  2, NPQ == .FALSE.
   ! NES  is .TRUE.  only if LL is negative. Then the sequence size is H0.
   !
   ncl = (nclass == 1)
   npq = (nclass < 2)
   nes = (ll < 0)
   debug_flag = .false.
   if (present(debug_flag0)) debug_flag = debug_flag0

   save_data_flag = .false.
   if (present(save_data_flag0)) save_data_flag = save_data_flag0

   ! CONSTANT coefficients setup
   k = 2
   do l = 1, NSTEPS
      temp = k+k*k
      if (ncl) temp = k
      WC(l) = ONE/temp
      temp = k
```

```
      UC(l) = ONE/temp
      k = k+1
   end do

   WC0 = HALF
   if (ncl) WC0 = ONE

   C(1) = −HS(1)
   D(1) = HS(1)
   R(1) = ONE/(HS(2)−HS(1))
   la = 1
   lc = 1
   do k = 3, NSTEPS
      lb = la
      la = lc+1
      lc = NW(k)
      C(la) = −HS(k−1)*C(lb)
      C(lc) = C(la−1)−HS(k−1)

      D(la) = HS(1)*D(lb)
      D(lc) = −C(lc)

      R(la) = ONE/(HS(k)−HS(1))
      R(lc) = ONE/(HS(k)−HS(k−1))

      if (k == 3) cycle

      do l = 4, k
         ld = la+l−3
         le = lb+l−4
         C(ld) = C(le)−HS(k−1)*C(le+1)
         D(ld) = D(le)+HS(l−2)*D(le+1)
         R(ld) = ONE/(HS(k)−HS(l−2))
      end do
   end do

   ! SS is, really, a CONSTANT (like WC, UC, and WC0)
   SS = 10.0_DP ** (−ll)
   !
   ! The statements above are used only once in an integration to set up
   ! the constants. They uses less than a second of execution time.
   !
   if (debug_flag) then
      ! Opening LOG file
      open(newunit = log_unit, file = 'ra15.log', status = 'replace')
   end if
   if (save_data_flag) then
      ! Opening DATA file
      open(newunit = data_unit, file = 'ra15.data', access = 'stream', &
           form = 'unformatted', status = 'replace')
      write(data_unit) nv, ll, nclass
   end if
end subroutine radau_on
subroutine radau_off()
   integer :: ierr

   if (save_data_flag) then
      ! Closing DATA file
      close(data_unit)
   end if

   if (debug_flag) then
      ! Closing LOG file
      close(log_unit)
   end if

   ! Freeing work space
   if (allocated(yp)) deallocate(yp,stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Deallocation failure for YP(:). Exiting...'
      stop
   end if
   if (allocated(y)) deallocate(y,stat=ierr)
   if (ierr /= 0) then
      write(*,*) '*** FATAL ERROR ***'
      write(*,*) 'Deallocation failure for Y(:). Exiting...'
      stop
```

```
    end if
    if (allocated(fj)) deallocate(fj,stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Deallocation failure for FJ(:). Exiting...'
       stop
    end if
    if (allocated(f0)) deallocate(f0,stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Deallocation failure for F0(:). Exiting...'
       stop
    end if
    if (allocated(e)) deallocate(e,stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Deallocation failure for E(:,:). Exiting...'
       stop
    end if
    if (allocated(g)) deallocate(g,stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Deallocation failure for G(:,:). Exiting...'
       stop
    end if
    if (allocated(b)) deallocate(b,stat=ierr)
    if (ierr /= 0) then
       write(*,*) '*** FATAL ERROR ***'
       write(*,*) 'Deallocation failure for B(:,:). Exiting...'
       stop
    end if
  end subroutine radau_off
  subroutine ra15(ta,tz,x,v,h0,force)
    real(DP), intent(in) :: ta, tz
    real(DP), intent(inout) :: x(:), v(:), h0
    procedure(ode_field) :: force
    !
    ! Integrator by E. Everhart, Physics Department, University of Denver
    ! Revision :  Angelo Graziosi (Sept. 12, 2014)
    !
    ! This 15th−order version is called RA15. Order NOR is 15.
    !
    ! y' = F(t,y)    is NCLASS == 1,   y" = F(t,y) is NCLASS == −2,
    ! y" = F(t,y,y') is NCLASS == 2
    !
    ! TF is t(final)−t(initial). (Negative when integrating backward.)
    ! NV = the number of simultaneous differential equations.
    !
    ! LL controls accuracy. Thus SS = 10.**(−LL) controls the size of
    ! the last term in a series. Try LL = 8 and work up or down from there.
    ! However, if LL < 0, then H0 is the constant sequence size used.
    ! A non−zero H0 sets the size of the first sequence regardless of
    ! LL sign. Zero's and Oh's could look alike here. Use care!
    !
    ! X and V enter as the starting position−velocity vector (values of y
    ! and y' at t = ta) and they output as the final position−velocity vector.
    !
    integer, parameter :: MAX_NCOUNT = 10
    real(DP), parameter :: SR = 1.4_DP, PW = ONE/9, EPS_TF_MATCH = 1.0E−08_DP, &
         Z2 = 2, Z3 = 3, Z4 = 4, Z5 = 5, Z6 = 6, Z7 = 7, &
         Z10 = 10, Z15 = 15, Z20 = 20, Z21 = 21, Z35 = 35
    integer, save :: i, k, ncount, ns, nf, ni, j
    logical, save :: nsf, nper
    real(DP), save :: tf, hval, hp, tm, tmf, h, h2, s, q, temp, hv, &
         bd(NSTEPS), q2, q3, q4, q5, q6, q7
    !
    ! NSF  is .FALSE. on starting sequence, otherwise .TRUE.
    ! NPER is .TRUE.  only on last sequence of integration.
    !
    nsf = .false.
    nper = .false.

    ! Initialize the working space. We need to initialize only B and BD.
    if (ncl) v(:) = ZERO
    b(:,:) = ZERO
    bd(:) = ZERO

    tf = tz−ta
```

```
h0 = sign(abs(h0),tf)
!
! Now set in an estimate to HP based on experience. Same sign as TF.
!
hp = sign(0.1_DP,tf)
if (h0 /= ZERO) hp = h0
if (hp/tf > HALF) hp = HALF*tf

! NCOUNT is the number of attempts to find the optimal sequence size.
! If NCOUNT > MAX_NCOUNT it returns to the caller: integration failed.
ncount = 0

if (debug_flag) then
   ! An * is the symbol for writing on the monitor. The file is unit
   ! LOG_UNIT.
   write(*,*) ' No. of calls, Every 10th seq.X(1),X(2),H,TM,TF'
end if
!
! Now the loop regarding the first sequence, aka THE MAIN LOOP, or
! if you prefer, the main sequence loop.
!
! NS is the number of sequences done
! NF is the number of calls to FORCE subroutine
! NI is the number of iterations to predict the B−values. NI is 6 for
! the first sequence, 2 after it.
!
main_loop: do
   ns = 0
   if (debug_flag) nf = 0
   ni = 6
   tm = ZERO
   tmf = ta
   call force(tmf,x,v,f0)
   if (debug_flag) nf = nf+1

   ! Now begins every sequence after the first. First find new
   ! G−values from the predicted B−values, following Eqs. (7) in text.
   every_sequence_loop: do
      do k = 1, nv
         g(1,k) = b(1,k)+D(1)*b(2,k)+D(2)*b(3,k)+D(4)*b(4,k)+D(7)*b(5,k) &
              +D(11)*b(6,k)+D(16)*b(7,k)
         g(2,k) = b(2,k)+D(3)*b(3,k)+D(5)*b(4,k)+D(8)*b(5,k)+D(12)*b(6,k) &
              +D(17)*b(7,k)
         g(3,k) = b(3,k)+D(6)*b(4,k)+D(9)*b(5,k)+D(13)*b(6,k)+D(18)*b(7,k)
         g(4,k) = b(4,k)+D(10)*b(5,k)+D(14)*b(6,k)+D(19)*b(7,k)
         g(5,k) = b(5,k)+D(15)*b(6,k)+D(20)*b(7,k)
         g(6,k) = b(6,k)+D(21)*b(7,k)
         g(7,k) = b(7,k)
      end do
      ! H    is the sequence size
      ! HP   is the guessed sequence size
      ! HVAL is the absolute value of sequence size
      ! TM   is the current time relative to TA
      ! TMF  is the current time (time to be passed to the force/FCN)
      h = hp
      h2 = h*h
      if (ncl) h2 = h
      hval = abs(h)

      if (debug_flag) then
         ! Writing to the screen during the integration lets one monitor
         ! the progress. Values are shown at every 10th sequence.
         if (ns/10*10 == ns) then
            temp = ZERO
            if (nv > 1) temp = x(2)
            write(*,'(1X,2I6,5F12.5)') nf, ns, x(1), temp, h, tm, tf
         end if
      end if

      ! better_B_loop is 6 iterations on first sequence and
      ! 2 iterations therafter
      better_B_loop: do i = 1, ni
         ! This loop is for each substep within a sequence.
         substep_loop: do j = 1, NSTEPS
            s = HS(j)
            q = s
            if (ncl) q = ONE
```

```
! Here Y is used for the value of y at substep n.
! We use Eq. (9). The collapsed series are broken in two part
! because an otherwise excellent compiler could not handle the
! complicated expression.
do k = 1, nv
   temp = WC(3)*b(3,k)+s*(WC(4)*b(4,k)+s*(WC(5)*b(5,k) &
        +s*(WC(6)*b(6,k)+s*WC(7)*b(7,k))))
   y(k) = x(k)+q*(h*v(k)+h2*s*(f0(k)*WC0+s*(WC(1)*b(1,k) &
        +s*(WC(2)*b(2,k)+s*temp))))

   ! If equations are 1st order or 2nd order special (i.e.
   ! without y', continue oops.. cycle..
   if (npq) cycle

   ! Next are calculated the velocity predictors if need for
   ! general Class II. Here YP is used as the value of y' at
   ! substep n (Eq. (10)).
   temp = UC(3)*b(3,k)+s*(UC(4)*b(4,k)+s*(UC(5)*b(5,k) &
        +s*(UC(6)*b(6,k)+s*UC(7)*b(7,k))))
   yp(k) = v(k)+s*h*(f0(k)+s*(UC(1)*b(1,k) &
        +s*(UC(2)*b(2,k)+s*temp)))
end do

! Find forces at each substep.
call force(tmf+s*h,y,yp,fj)
if (debug_flag) nf = nf+1
!
! (A)
! Find G−values from the force FJ found at current substep.
! This section uses Eqs. (4) of text.
! Before save in TEMP the current value.
!
! (B)
! TEMP is now the improvement on G(J,K) over its former
! value. Now we upgrade the B−value using this difference
! in the one term.
! This section is based on Eqs. (5).
!
select case (j)
case (1)
   do k = 1, nv
      q = (fj(k)−f0(k))/s

      ! See comment (A) above...
      temp = g(1,k)
      g(1,k) = q

      ! See comment (B) above...
      temp = g(1,k)−temp
      b(1,k) = b(1,k)+temp
   end do
case (2)
   do k = 1, nv
      q = (fj(k)−f0(k))/s

      ! See comment (A) above...
      temp = g(2,k)
      g(2,k) = (q−g(1,k))*R(1)

      ! See comment (B) above...
      temp = g(2,k)−temp
      b(1,k) = b(1,k)+C(1)*temp
      b(2,k) = b(2,k)+temp
   end do
case (3)
   do k = 1, nv
      q = (fj(k)−f0(k))/s

      ! See comment (A) above...
      temp = g(3,k)
      g(3,k) = ((q−g(1,k))*R(2)−g(2,k))*R(3)

      ! See comment (B) above...
      temp = g(3,k)−temp
      b(1,k) = b(1,k)+C(2)*temp
      b(2,k) = b(2,k)+C(3)*temp
      b(3,k) = b(3,k)+temp
   end do
```

```
      case (4)
         do k = 1, nv
            q = (fj(k)−f0(k))/s

            ! See comment (A) above...
            temp = g(4,k)
            g(4,k) = (((q−g(1,k))*R(4)−g(2,k))*R(5)−g(3,k))*R(6)

            ! See comment (B) above...
            temp = g(4,k)−temp
            b(1,k) = b(1,k)+C(4)*temp
            b(2,k) = b(2,k)+C(5)*temp
            b(3,k) = b(3,k)+C(6)*temp
            b(4,k) = b(4,k)+temp
         end do
      case (5)
         do k = 1, nv
            q = (fj(k)−f0(k))/s

            ! See comment (A) above...
            temp = g(5,k)
            g(5,k) = ((((q−g(1,k))*R(7)−g(2,k))*R(8)−g(3,k))*R(9) &
                 −g(4,k))*R(10)

            ! See comment (B) above...
            temp = g(5,k)−temp
            b(1,k) = b(1,k)+C(7)*temp
            b(2,k) = b(2,k)+C(8)*temp
            b(3,k) = b(3,k)+C(9)*temp
            b(4,k) = b(4,k)+C(10)*temp
            b(5,k) = b(5,k)+temp
         end do
      case (6)
         do k = 1, nv
            q = (fj(k)−f0(k))/s

            ! See comment (A) above...
            temp = g(6,k)
            g(6,k) = (((((q−g(1,k))*R(11)−g(2,k))*R(12) &
                 −g(3,k))*R(13)−g(4,k))*R(14)−g(5,k))*R(15)

            ! See comment (B) above...
            temp = g(6,k)−temp
            b(1,k) = b(1,k)+C(11)*temp
            b(2,k) = b(2,k)+C(12)*temp
            b(3,k) = b(3,k)+C(13)*temp
            b(4,k) = b(4,k)+C(14)*temp
            b(5,k) = b(5,k)+C(15)*temp
            b(6,k) = b(6,k)+temp
         end do
      case (7)
         do k = 1, nv
            q = (fj(k)−f0(k))/s

            ! See comment (A) above...
            temp = g(7,k)
            g(7,k) = ((((((q−g(1,k))*R(16)−g(2,k))*R(17) &
                 −g(3,k))*R(18)−g(4,k))*R(19)−g(5,k))*R(20) &
                 −g(6,k))*R(21)

            ! See comment (B) above...
            temp = g(7,k)−temp
            b(1,k) = b(1,k)+C(16)*temp
            b(2,k) = b(2,k)+C(17)*temp
            b(3,k) = b(3,k)+C(18)*temp
            b(4,k) = b(4,k)+C(19)*temp
            b(5,k) = b(5,k)+C(20)*temp
            b(6,k) = b(6,k)+C(21)*temp
            b(7,k) = b(7,k)+temp
         end do
      end select
   end do substep_loop

   if (nes .or. i < ni) cycle better_B_loop

   ! Integration of sequence is over. Next is sequence size control.
   hv = ZERO
   do k = 1, nv
```

```
      hv = max(hv,abs(b(7,k)))
   end do
   hv = hv*WC(7)/hval**7
end do better_B_loop

! If this is the 1st sequence... we still have to adjust the
! time step
if (.not. nsf) then
   if (.not. nes) hp = sign((SS/hv)**PW,tf)
   if (nes .or. hp/h > ONE) then
      if (nes) hp = h0
      nsf = .true.
      if (save_data_flag) then
         write(data_unit) ns, h, tmf, x, v
      end if
   else
      hp = 0.8_DP*hp
      ncount = ncount+1
      if (ncount > MAX_NCOUNT) then
         write(*,*)
         write(*,*) '************************************'
         write(*,*) 'NCOUNT > ', MAX_NCOUNT
         write(*,*) 'Cannot find an optimal sequence size.'
         write(*,*) 'RA15 returns to the caller.'
         write(*,*) '************************************'
         write(*,*)
         ! Exiting the main loop should be the same as RETURN.
         ! (Doing so one could close also an LOG_UNIT file if
         ! it were opened at the beginning of this routine...)
         exit main_loop
         !return
      end if
      if (debug_flag) then
         if (ncount > 1) &
              write(log_unit,'(2X,2I2,2ES18.10)') NOR, ncount, h, hp
      end if

      ! Restart with HP = 0.8*H if new HP is smaller than original
      ! H on 1st sequence.
      cycle main_loop
   end if
end if

! This loop finds new X and V values at end of sequence.
! Eqs. (11), (12).
do k = 1, nv
   x(k) = x(k)+v(k)*h+h2*(f0(k)*WC0+b(1,k)*WC(1)+b(2,k)*WC(2) &
        +b(3,k)*WC(3)+b(4,k)*WC(4)+b(5,k)*WC(5)+b(6,k)*WC(6) &
        +b(7,k)*WC(7))

   ! If equations are 1st order, skip to compute y' (aka V)
   ! at end of sequence, i.e. cycle..
   if (ncl) cycle

   v(k) = v(k)+h*(f0(k)+b(1,k)*UC(1)+b(2,k)*UC(2)+b(3,k)*UC(3) &
        +b(4,k)*UC(4)+b(5,k)*UC(5)+b(6,k)*UC(6)+b(7,k)*UC(7))
end do
! We have done a sequence and can update current time and
! sequence counter.
tm = tm+h
tmf = tmf+h
ns = ns+1

if (save_data_flag .and. .not.nper) then
   write(data_unit) ns, h, tmf, x, v
end if

! Return if done.
if (nper) then
   if (debug_flag) then
      temp = ZERO
      if (nv > 1) temp = x(2)
      write(*,'(1X,2I6,5F12.5)') nf, ns, x(1), temp, h, tm, tf
      write(log_unit,'(1X,2I6)') nf, ns
   end if

   if (save_data_flag) then
      write(data_unit) ns, h, tmf, x, v
```

```
            end if

            ! On exit, H0 contains the last computed (signed) sequence size
            h0 = h
            ! Exiting the main loop should be the same as RETURN.
            ! (Doing so one could close also an LOG_UNIT file if
            ! it were opened at the beginning of this routine...)
            exit main_loop
            !return
         end if

         ! Control on size of next sequence and adjust last sequence to exactly
         ! cover the integration span. NPER = .TRUE. set on last sequence.
         call force(tmf,x,v,f0)
         if (debug_flag) nf = nf+1

         if (nes) then
            hp = h0
         else
            hp = sign((SS/hv)**PW,tf)
            if (hp/h > SR) hp = h*SR
         end if
         if (abs(tm+hp) >= abs(tf)−EPS_TF_MATCH) then
            hp = tf−tm
            nper = .true.
         end if

         ! Now predict B−values for next step using Eqs. (13). Values from the
         ! preceding sequence were saved in the E−matrix. The correction BD
         ! is applied in the following loop as described in Sec. 2.5.
         q = hp/h

         ! To avoid re−computation of the same expession (q**2, q**3,...)
         ! for each K...
         q2 = q*q!q**2
         q3 = q*q2!q**3
         q4 = q2*q2!q**4
         q5 = q2*q3!q**5
         q6 = q3*q3!q**6
         q7 = q3*q4!q**7
         do k = 1, nv
            ! If we have done at least TWO sequences..
            if (ns /= 1) then
               do j = 1, NSTEPS
                  bd(j) = b(j,k)−e(j,k)
               end do
            end if

            e(1,k) = q*(b(1,k)+Z2*b(2,k)+Z3*b(3,k)+Z4*b(4,k)+Z5*b(5,k) &
                 +Z6*b(6,k)+Z7*b(7,k))
            e(2,k) = q2*(b(2,k)+Z3*b(3,k)+Z6*b(4,k)+Z10*b(5,k)+Z15*b(6,k) &
                 +Z21*b(7,k))
            e(3,k) = q3*(b(3,k)+Z4*b(4,k)+Z10*b(5,k)+Z20*b(6,k)+Z35*b(7,k))
            e(4,k) = q4*(b(4,k)+Z5*b(5,k)+Z15*b(6,k)+Z35*b(7,k))
            e(5,k) = q5*(b(5,k)+Z6*b(6,k)+Z21*b(7,k))
            e(6,k) = q6*(b(6,k)+Z7*b(7,k))
            e(7,k) = q7*b(7,k)

            ! Apply the correction.. Notice that when we have done ONLY
            ! one sequence (NS == 1), BD == 0 from its initialization, i.e.
            ! we are doing B = E. It is only when NS > 1 that we are applying
            ! the correction BD.
            do j = 1, NSTEPS
               b(j,k) = e(j,k)+bd(j)
            end do
         end do

         ! Two iterations for every sequence. (Use 3 for 23rd and 27th order.)
         ni = 2
      end do every_sequence_loop
   end do main_loop
  end subroutine ra15
end module everhart_integrator
```

```
!
! Fortran testing of Everhart integrator
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! BUILDING
!
!   cd ~/programming/nbody.apps
!
!   rm -rf {*.mod,~/programming/modules/*} && \
!   gfortran [-Warray-temporaries] -O3 -Wall $BLD_OPTS \
!     -J ~/programming/modules \
!     ../basic-modules/basic_mods.f90 \
!     ../ode-modules/{everhart_integrator.f90,ode_integrators.f90} \
!     test_jsunp.f90 -o test_jsunp$EXE && \
!     rm -rf {*.mod,~/programming/modules/*}
!
!   ./test_jsunp$EXE
!
!   where:
!
!     BLD_OPTS =
!     EXE = .out
!
!   for the build on GNU/Linux
!
!     BLD_OPTS = -static
!     EXE =
!
!   for the build on MSYS2/MINGW32/MINGW64 shells
!
!
! NOTES
!
! The date range used in the work
!
!   Eckert, Brouwer, Clemence (1951),
!     Coordinates of the Five Outer Planets 1653-2060,
!     Astronom. Papers American Ephem. XII
!
!
! is : [ JD = 2325000.5 = 1653.07.13.5, JD = 2473800.5 = 2060.12.07.0 ]
!
! On the WEB, that work is at the URL
!
!   http://hdl.handle.net/2027/mdp.39015017142152
!
! Notice that before 1925 Jan. 01, the astronomical dates began at the
! Greenwich Mean Noon (i.e. at the 12.0 hours): this explain why the day of
! the Gregorian date which corresponds to the JD 2325000.5 is 13.5 and not
! 14.0 as one, normally, would expect. For the same reason, the date
! 1653.05.02.0 has as JD, 2324928.0 and not 2324927.5.
!
! How initial conditions are computed
!
!   The positions are those reported by the above work for the date
!   1941.01.06.0 (JD 2430000.5). For example, for Jupiter they are (in AU):
!
!     (+3.4294 74152, +3.3538 69597, +1.3549 49017)
!
!   The velocity is computed numerically as numerical differentiation.
!   The 11-point formula is used (see below).
!   Initially, the 5-point formula was used,
!
!     f'(0) = [f(-2)-8*f(-1)+8*f(+1)-f(+2)]/(12*h)  +  O(h**4)
!
!   being,
!
!     F(i) = F(i*H), i = -2, -1, 0, +1, +2 and F = f' or F = f.
!
!   (See Koonin-Meredith, Computational Physics, Fortran)
!
!   In the work of Eckert and friends, the positions are given at
!   intervals of 40 days, so the natural time unit is 40D-unit.
```

```
!
! Formulas for Numerical Differentiation
!
!   The following formulas are adaped from (up to 11-point)
!
!     http://www.trentfguidry.net/post/2010/09/04/
!     Numerical-differentiation-formulas.aspx
!
!   and (table 1, up to 17-point) from
!
!     http://www.ias.ac.in/chemsci/Pdf-Sep2009/935.pdf
!
! seven-points formula:
!
!   f'(0) = (-f(-3)+9*f(-2)-45*f(-1)+45*f(+1)-9*f(+2)+f(+3))/(60*h)
!
! nine-points formula:
!
!   f'(0) = (+3*f(-4)-32*f(-3)+168*f(-2)-672*f(-1) \
!     +672*f(+1)-168*f(+2)+32*f(+3)-3*f(+4))/(840*h)
!
! eleven-points formula:
!
!   f'(0) = (-2*f(-5)+25*f(-4)-150*f(-3)+600*f(-2)-2100*f(-1) \
!     +2100*f(+1)-600*f(+2)+150*f(+3)-25*f(+4)+2*f(+5))/(2520*h)
!
! thirteen-points formula:
!
!   f'(0) = (+5*f(-6)-72*f(-5)+495*f(-4)-2200*f(-3)+7425*f(-2)-23760*f(-1) \
!     +23760*f(+1)-7425*f(+2)+2200*f(+3)-495*f(+4)+72*f(+5)-5*f(+6))/(27720*h)
!
! fifteen-points formula:
!
!   f'(0) = (-15*f(-7)+245*f(-6)-1911*f(-5)+9555*f(-4)-35035*f(-3) \
!     +105105*f(-2)-315315*f(-1)+315315*f(+1)-105105*f(+2)+35035*f(+3) \
!     -9555*f(+4)+1911*f(+5)-245*f(+6)+15*f(+7))/(360360*h)
!
! seventeen-points formula:
!
!   f'(0) = (7*f(-8)-128*f(-7)+1120*f(-6)-6272*f(-5)+25480*f(-4)-81536*f(-3) \
!     +224224*f(-2)-640640*f(-1)+640640*f(+1)-224224*f(+2)+81536*f(+3) \
!     -25480*f(+4)+6272*f(+5)-1120*f(+6)+128*f(+7)-7*f(+8))/(720720*h)
!
!
! For the date TZ = 2469600.5_DP = 2049.06.08.0, the best result is obtained
! with the 9-points formula. In the Eckert and friends work, one can find that
! Jupiter position (X, Y, Z) at TZ = 2469600.5_DP is
!
!   -0.832708494, +4.678840367, +2.027156624
!
! With this program and 9-points formula we find
!
!   -0.832709005, +4.678840309, +2.027156609
!
! For the date TZ = 2325000.5_DP = 1653.07.13.5, the best result is obtained
! with the 9-points formula. In the Eckert and friends work, one can find that
! Jupiter position (X, Y, Z) at TZ = 2325000.5_DP is
!
!   +3.548739356, -3.280988352, -1.495613025
!
! With this program and 9-points formula we find
!
!   +3.548738385, -3.280989304, -1.495613406
!

program test_jsunp
  use kind_consts, only: DP
  use general_routines, only: system_time
  use ode_integrators, only: deqgbs
  use everhart_integrator, only: radau_on, ra15, radau_off
  implicit none
  integer, parameter :: NDIM = 3, NB = 5, NV = NDIM*NB, NEQ = 2*NV
  integer, parameter :: NCLASS = -2, LL = -14
  integer, parameter :: MAX_DERIV_PTS = 17, ND = MAX_DERIV_PTS/2
  real(DP), parameter :: Z0 = 0, T_FACT = 800, V_FACT = 20
  integer :: i, ip, np = MAX_DERIV_PTS, np2
  real(DP) :: tf, h, &
       ta = 2430000.5_DP, &   ! 1941.01.06.0
```

```
     tz = 2325000.5_DP, &   ! 1653.07.13.5
     !tz = 2469600.5_DP, &   ! 2049.06.08.0
     h0 = 320   ! The time step used in Everhart paper, sec. 3.3
real(DP) :: x0(NV), v0(NV), xx(-ND:ND,NB), yy(-ND:ND,NB), zz(-ND:ND,NB)
real(DP) :: x(NV), v(NV), y(NEQ), w(NEQ,36), eps = 1.0E-12_DP, &
     dx(3), dv(3)
real(DP) :: t0, t1
character(len = 32) :: np_arg

i = 0
do
   call get_command_argument(i,np_arg)
   if (len_trim(np_arg) == 0) exit

   if (i == 1) read(np_arg,*) np

   i = i+1
end do

! The numer of arguments that have been read is stored in 'i'
!
! In this command line,
!
!   ./test_jsunp.out 17
!
! we have 2 arguments: the program name and the number '17'
!
if (i /= 2) then
   write(*,*) 'USAGE: ./test_jsunp.out <N_POINTS>'
   stop
end if

if (np < 5 .or. MAX_DERIV_PTS < np) then
   write(*,*) np, '-points formula not implemented.'
   write(*,*) 'Program stops...'
   stop
end if

write(*,*) 'Computing initial conditions using ', np, &
     '-points formula for derivatives...'
write(*,*)
!
! Planets positions around the starting date, 2430000.5 JD = 1941.01.06.0.
! The rule, for the planets n. I, is:
!
!   2429680.5 JD = 1940.02.21.0  ==>  XX(-8,I), YY(-8,I), ZZ(-8,I)
!   2429720.5 JD = 1940.04.01.0  ==>  XX(-7,I), YY(-7,I), ZZ(-7,I)
!   2429760.5 JD = 1940.05.11.0  ==>  XX(-6,I), YY(-6,I), ZZ(-6,I)
!   2429800.5 JD = 1940.06.20.0  ==>  XX(-5,I), YY(-5,I), ZZ(-5,I)
!   2429840.5 JD = 1940.07.30.0  ==>  XX(-4,I), YY(-4,I), ZZ(-4,I)
!   2429880.5 JD = 1940.09.08.0  ==>  XX(-3,I), YY(-3,I), ZZ(-3,I)
!   2429920.5 JD = 1940.10.18.0  ==>  XX(-2,I), YY(-2,I), ZZ(-2,I)
!   2429960.5 JD = 1940.11.27.0  ==>  XX(-1,I), YY(-1,I), ZZ(-1,I)
!
!   2430000.5 JD = 1941.01.06.0  ==>  XX(0,I), YY(0,I), ZZ(0,I)
!
!   2430040.5 JD = 1941.02.15.0  ==>  XX(+1,I), YY(+1,I), ZZ(+1,I)
!   2430080.5 JD = 1941.03.27.0  ==>  XX(+2,I), YY(+2,I), ZZ(+2,I)
!   2430120.5 JD = 1941.05.06.0  ==>  XX(+3,I), YY(+3,I), ZZ(+3,I)
!   2430160.5 JD = 1941.06.15.0  ==>  XX(+4,I), YY(+4,I), ZZ(+4,I)
!   2430200.5 JD = 1941.07.25.0  ==>  XX(+5,I), YY(+5,I), ZZ(+5,I)
!   2430240.5 JD = 1941.09.03.0  ==>  XX(+6,I), YY(+6,I), ZZ(+6,I)
!   2430280.5 JD = 1941.10.13.0  ==>  XX(+7,I), YY(+7,I), ZZ(+7,I)
!   2430320.5 JD = 1941.11.22.0  ==>  XX(+8,I), YY(+8,I), ZZ(+8,I)
!
! (I = 1 for jupiter, 2 for Saturn,..., 5 for Pluto)
!
xx(:,1) = [ 4.724184873_DP, 4.621378591_DP, 4.500533544_DP, &
     4.362143991_DP, 4.206780873_DP, 4.035088112_DP, &
     3.847778387_DP, 3.645628468_DP, &
     3.429474152_DP, &
     3.200204874_DP, 2.958758073_DP, &
     2.706113369_DP, 2.443286632_DP, 2.171324008_DP, &
     1.891295968_DP, 1.604291433_DP, 1.311412040_DP ]
xx(:,2) = [ 7.833676700_DP, 7.700327328_DP, 7.562355230_DP, &
     7.419827637_DP, 7.272815279_DP, 7.121392386_DP, &
     6.965636697_DP, 6.805629460_DP, &
     6.641455425_DP, &
```

```
     6.473202831_DP, 6.300963375_DP, &
     6.124832183_DP, 5.944907752_DP, 5.761291895_DP, &
     5.574089663_DP, 5.383409269_DP, 5.189362000_DP ]
xx(:,3) = [ 12.280065983_DP, 12.155630171_DP, 12.030409732_DP, &
     11.904412250_DP, 11.777645443_DP, 11.650117157_DP, &
     11.521835361_DP, 11.392808146_DP, &
     11.263043721_DP, &
     11.132550402_DP, 11.001336615_DP, &
     10.869410886_DP, 10.736781836_DP, 10.603458178_DP, &
     10.469448708_DP, 10.334762302_DP, 10.199407912_DP ]
xx(:,4) = [ -30.062244610_DP, -30.075613890_DP, -30.088485654_DP, &
     -30.100859239_DP, -30.112733915_DP, -30.124108891_DP, &
     -30.134983319_DP, -30.145356298_DP, &
     -30.155226876_DP, &
     -30.164594060_DP, -30.173456819_DP, &
     -30.181814088_DP, -30.189664777_DP, -30.197007772_DP, &
     -30.203841948_DP, -30.210166165_DP, -30.215979282_DP ]
xx(:,5) = [ -20.552905937_DP, -20.624802965_DP, -20.696551366_DP, &
     -20.768149710_DP, -20.839596488_DP, -20.910890122_DP, &
     -20.982028960_DP, -21.053011290_DP, &
     -21.123835338_DP, &
     -21.194499275_DP, -21.265001224_DP, &
     -21.335339263_DP, -21.405511434_DP, -21.475515746_DP, &
     -21.545350180_DP, -21.615012699_DP, -21.684501247_DP ]

yy(:,1) = [ 1.395636674_DP, 1.670118716_DP, 1.938084558_DP, &
     2.198497852_DP, 2.450359098_DP, 2.692710961_DP, &
     2.924643186_DP, 3.145297096_DP, &
     3.353869597_DP, &
     3.549616687_DP, 3.731856438_DP, &
     3.899971436_DP, 4.053410685_DP, 4.191690969_DP, &
     4.314397699_DP, 4.421185252_DP, 4.511776837_DP ]
yy(:,2) = [ 4.692917397_DP, 4.863738797_DP, 5.031639065_DP, &
     5.196506906_DP, 5.358232433_DP, 5.516707288_DP, &
     5.671824759_DP, 5.823479888_DP, &
     5.971569579_DP, &
     6.115992687_DP, 6.256650116_DP, &
     6.393444896_DP, 6.526282269_DP, 6.655069769_DP, &
     6.779717298_DP, 6.900137218_DP, 7.016244426_DP ]
yy(:,3) = [ 14.058466481_DP, 14.141261359_DP, 14.223157071_DP, &
     14.304146527_DP, 14.384222716_DP, 14.463378712_DP, &
     14.541607680_DP, 14.618902882_DP, &
     14.695257679_DP, &
     14.770665540_DP, 14.845120038_DP, &
     14.918614862_DP, 14.991143811_DP, 15.062700802_DP, &
     15.133279868_DP, 15.202875158_DP, 15.271480939_DP ]
yy(:,4) = [ 2.576077682_DP, 2.461358369_DP, 2.346589777_DP, &
     2.231772859_DP, 2.116908609_DP, 2.001998061_DP, &
     1.887042292_DP, 1.772042432_DP, &
     1.656999664_DP, &
     1.541915229_DP, 1.426790427_DP, &
     1.311626621_DP, 1.196425238_DP, 1.081187771_DP, &
     0.965915778_DP, 0.850610887_DP, 0.735274788_DP ]
yy(:,5) = [ 29.131108835_DP, 29.046387877_DP, 28.961424498_DP, &
     28.876217865_DP, 28.790767187_DP, 28.705071716_DP, &
     28.619130752_DP, 28.532943648_DP, &
     28.446509814_DP, &
     28.359828720_DP, 28.272899897_DP, &
     28.185722946_DP, 28.098297530_DP, 28.010623384_DP, &
     27.922700310_DP, 27.834528183_DP, 27.746106944_DP ]

zz(:,1) = [ 0.483165204_DP, 0.603437945_DP, 0.721356205_DP, &
     0.836463264_DP, 0.948316335_DP, 1.056488937_DP, &
     1.160573114_DP, 1.260181470_DP, &
     1.354949017_DP, &
     1.444534814_DP, 1.528623379_DP, &
     1.606925883_DP, 1.679181106_DP, 1.745156173_DP, &
     1.804647057_DP, 1.857478868_DP, 1.903505939_DP ]
zz(:,2) = [ 1.602157778_DP, 1.678543833_DP, 1.753921800_DP, &
     1.828242798_DP, 1.901458371_DP, 1.973520543_DP, &
     2.044381861_DP, 2.113995438_DP, &
     2.182314997_DP, &
     2.249294912_DP, 2.314890241_DP, &
     2.379056771_DP, 2.441751048_DP, 2.502930418_DP, &
     2.562553065_DP, 2.620578047_DP, 2.676965343_DP ]
zz(:,3) = [ 5.986200890_DP, 6.024237821_DP, 6.061892122_DP, &
     6.099160581_DP, 6.136040020_DP, 6.172527294_DP, &
     6.208619295_DP, 6.244312956_DP, &
```

```
     6.279605251_DP, &
     6.314493195_DP, 6.348973852_DP, &
     6.383044329_DP, 6.416701783_DP, 6.449943418_DP, &
     6.482766489_DP, 6.515168298_DP, 6.547146198_DP ]
zz(:,4) = [ 1.811963696_DP, 1.765312683_DP, 1.718628845_DP, &
     1.671912540_DP, 1.625164136_DP, 1.578384018_DP, &
     1.531572591_DP, 1.484730279_DP, &
     1.437857527_DP, &
     1.390954805_DP, 1.344022607_DP, &
     1.297061453_DP, 1.250071890_DP, 1.203054493_DP, &
     1.156009865_DP, 1.108938636_DP, 1.061841466_DP ]
zz(:,5) = [ 15.431663080_DP, 15.426689424_DP, 15.421588226_DP, &
     15.416358811_DP, 15.411000518_DP, 15.405512704_DP, &
     15.399894742_DP, 15.394146025_DP, &
     15.388265968_DP, &
     15.382254010_DP, 15.376109615_DP, &
     15.369832271_DP, 15.363421496_DP, 15.356876831_DP, &
     15.350197850_DP, 15.343384151_DP, 15.336435365_DP ]

! Compute the initial conditions needed for the integration.
! Equatorial Rectangular Coordinates, B1950.0 Epoch
np2 = np/2
do i = 1, NB
   ip = 1+NDIM*(i-1)

   ! Initial position in AU
   x0(ip:ip+2) = [ xx(0,i), yy(0,i), zz(0,i) ]

   ! Initial velocity in AU/40D-unit, i.e. HSTEP = 1 40D-unit
   v0(ip:ip+2) = [ deriv(np,xx(-np2:np2,i)), deriv(np,yy(-np2:np2,i)), &
        deriv(np,zz(-np2:np2,i)) ]
end do

write(*,*) 'Testing CLASS IIS differential equations:'
write(*,*) '       The Outer Planets Problem'
write(*,*) '      (sec. 3.3 of Everhart paper)'
write(*,*)

! Conversion to use time unit 800 days. Being the velocity given in
! AU/40D-unit, the conversion factor is 800/40 = 20
x = x0
v = v0*V_FACT
tf = (tz-ta)/T_FACT
h = h0/T_FACT

!call radau_on(NV,LL,NCLASS,.true.,.true.)
!call radau_on(NV,LL,NCLASS,save_data_flag0=.true.)
call radau_on(NV,LL,NCLASS,.true.)
t0 = system_time()
call ra15(Z0,tf,x,v,h,force)
t1 = system_time()
call radau_off()

! Conversion to AU/D
v(1:3) = v(1:3)/T_FACT

write(*,*)
write(*,'(a,f12.2,a)') 'At t = ', tz, ' the result is (JUPITER):'
write(*,'(a,3f15.9)') 'RA15   :   X = ', x(1:3)
write(*,'(a,3f15.9)') 'RA15   :   V = ', v(1:3)
write(*,'(A,F8.3,A)') 'Run time ',t1-t0,' seconds!'

y(1:NV) = x0
y(NV+1:NEQ) = v0*V_FACT
tf = (tz-ta)/T_FACT
h = h0/T_FACT

t0 = system_time()
call deqgbs(NEQ,Z0,tf,y,h,eps,w,sub)
t1 = system_time()

! Conversion to AU/D
y(NV+1:NV+3) = y(NV+1:NV+3)/T_FACT

dx = x(1:3)-y(1:3)
dv = v(1:3)-y(NV+1:NV+3)

write(*,*)
```

```
  write(*,'(a,f12.2,a)') 'At t = ', tz, ' the result is (JUPITER):'
  write(*,'(a,3f15.9)') 'DEQGBS :   X = ', y(1:3)
  write(*,'(a,3f15.9)') 'DEQGBS :   V = ', y(NV+1:NV+3)
  write(*,'(A,F8.3,A)') 'Run time ',t1-t0,' seconds!'
  write(*,*)
  write(*,'(a,3es10.2)') 'DX = ', dx
  write(*,'(a,3es10.2)') 'DV = ', dv
  write(*,*)
  write(*,'(a,es10.2)') 'ABS(DX) = ', norm2(dx)
  write(*,'(a,es10.2)') 'ABS(DV) = ', norm2(dv)

contains

  function deriv(n_points,f) result (df)
    real(DP) :: df
    ! Declaring F as F(:) would mean that its lower bound is 1 and not
    ! -n_points/2 as in the caller.
    integer, intent(in) :: n_points
    real(DP), intent(in) :: f(-n_points/2:)

    select case (n_points)
    case (5)
       ! 5-points formula, HSTEP = 1
       df = (f(-2)-8.0_DP*f(-1)+8.0_DP*f(+1)-f(+2))/12.0_DP
    case (7)
       ! 7-points formula, HSTEP = 1
       df = (-f(-3)+9.0_DP*f(-2)-45.0_DP*f(-1)+45.0_DP*f(+1)-9.0_DP*f(+2) &
            +f(+3))/60.0_DP
    case (9)
       ! 9-points formula, HSTEP = 1
       df = (+3.0_DP*f(-4)-32.0_DP*f(-3)+168.0_DP*f(-2)-672.0_DP*f(-1) &
            +672.0_DP*f(+1)-168.0_DP*f(+2)+32.0_DP*f(+3) &
            -3.0_DP*f(+4))/840.0_DP
    case (11)
       ! 11-points formula, HSTEP = 1
       ! If F were declared as F(:) then
       !
       ! df = (-2.0_DP*f(1)+25.0_DP*f(2)-150.0_DP*f(3)+600.0_DP*f(4) &
       !   -2100.0_DP*f(5)+2100.0_DP*f(7)-600.0_DP*f(8)+150.0_DP*f(9) &
       !   -25.0_DP*f(10)+2.0_DP*f(11))/2520.0_DP
       !
       df = (-2.0_DP*f(-5)+25.0_DP*f(-4)-150.0_DP*f(-3)+600.0_DP*f(-2) &
            -2100.0_DP*f(-1)+2100.0_DP*f(+1)-600.0_DP*f(+2)+150.0_DP*f(+3) &
            -25.0_DP*f(+4)+2.0_DP*f(+5))/2520.0_DP
    case (13)
       ! 13-points formula, HSTEP = 1
       df = (+5.0_DP*f(-6)-72.0_DP*f(-5)+495.0_DP*f(-4)-2200.0_DP*f(-3) &
            +7425.0_DP*f(-2)-23760.0_DP*f(-1)+23760.0_DP*f(+1) &
            -7425.0_DP*f(+2)+2200.0_DP*f(+3)-495.0_DP*f(+4)+72.0_DP*f(+5) &
            -5.0_DP*f(+6))/27720.0_DP
    case (15)
       ! 15-points formula, HSTEP = 1
       df = (-15.0_DP*f(-7)+245.0_DP*f(-6)-1911.0_DP*f(-5)+9555.0_DP*f(-4) &
            -35035*f(-3)+105105.0_DP*f(-2)-315315.0_DP*f(-1) &
            +315315.0_DP*f(+1)-105105.0_DP*f(+2)+35035.0_DP*f(+3) &
            -9555.0_DP*f(+4)+1911.0_DP*f(+5)-245.0_DP*f(+6) &
            +15.0_DP*f(+7))/360360.0_DP
    case (17)
       ! 17-points formula, HSTEP = 1
       df = (7.0_DP*f(-8)-128.0_DP*f(-7)+1120.0_DP*f(-6)-6272.0_DP*f(-5) &
            +25480.0_DP*f(-4)-81536.0_DP*f(-3)+224224.0_DP*f(-2) &
            -640640.0_DP*f(-1)+640640.0_DP*f(+1)-224224.0_DP*f(+2) &
            +81536.0_DP*f(+3)-25480.0_DP*f(+4)+6272.0_DP*f(+5)-1120.0_DP*f(+6) &
            +128.0_DP*f(+7)-7.0_DP*f(+8))/720720.0_DP
    case default
       df = 0
       write(*,*) n_points, '-points formula not implemented.'
       write(*,*) 'Program stops...'
       stop
    end select

  end function deriv

  subroutine force(t,x,v,f)
    ! The FORCE subroutine for the 5 outer planet integration.
    real(DP), intent(in) :: t, x(:), v(:)
    real(DP), intent(out) :: f(:)
    ! The above statement assumes an 8-byte doubel word (64 bits).
```

```
    ! X, V, and F are dimensioned assumed-shape because they appear
    ! in the call.
    !
    ! SCZ is the Gaussian constant for an 800-day time unit, and SC is the
    ! same except the mass of sun is augmented by masses of inner
    ! planets, Mercury through Mars.
    ! X, V, and F are dimensioned for 15 in the calling programs. Indices 1,2,3
    ! are for x,y,z for Jupiter, 4,5,6 are for x,y,z Saturn, 7,8,9 are for
    ! x,y,z Uranus, 10,11,12 for x,y,z Neptune, and 13,14,15 for Pluto.
    real(DP), parameter :: SCZ = -((1.720209895E-2_DP)**2)*((800._DP)**2), &
         SC = -1.8938494521574133E2_DP, Z1 = 1
    ! The reciprocal masses of the 5 planets, units of reciprocal sun.
    real(DP), parameter :: RM(NB) = [ 1047.355_DP, 3501.6_DP, 22869._DP, &
         19314._DP, 360000._DP]
    real(DP), save :: pm(NB), r(NB), rh(NB,NB), scm
    integer, save :: j, k, l, n, na
    logical, save :: first = .true.

    if (first) then
       first = .false.
       pm(:) = -SCZ/RM(:)
    end if

    do  n = 1, NB
       j = (n-1)*3+1
       r(n) = Z1/norm2(x(j:j+2))**3
       if (n == NB) cycle
       na = n+1
       do  l = na, NB
          k = (l-1)*3+1
          rh(n,l) = Z1/norm2(x(j:j+2)-x(k:k+2))**3
          rh(l,n) = rh(n,l)
       end do
       ! Indices K and J run 1-15, indices N and L for the planets run 1-5.
       ! The mass factors are in PM, the distance from the sun of each planet
       ! contribute to R, and the planet-to-planet distances contribute to RH.
    end do

    do n = 1, NB
       j = (n-1)*3+1
       scm = (SC-pm(n))*r(n)
       f(j:j+2) = scm*x(j:j+2)
       ! Th F-values above are for the sun-planet forces/unit mass.
       do l = 1, NB
          if (l == n) cycle
          k = (l-1)*3+1
          f(j:j+2) = f(j:j+2)+pm(l)*((x(k:k+2)-x(j:j+2))*rh(n,l)-x(k:k+2)*r(l))
          ! The mutual planetary perturbation forces/unit mass are added on. The
          ! first part of the second term is due to the planet-to-planet force,
          ! and the second part is the indirect term because the sun at the
          ! origin is not at the center of mass of the system.
       end do
    end do

  end subroutine force

  subroutine sub(t,y,f)
    real(DP), intent(in) :: t, y(:)
    real(DP), intent(out) :: f(:)
    f(1:NV) = y(NV+1:NEQ)
    call force(t,y(1:NV),y(NV+1:NEQ),f(NV+1:NEQ))
  end subroutine sub

end program test_jsunp
```

```
!
! Fortran Interface to the Xbgi-364p/WinBGIm-6.0 Library
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! HOW TO BUILD XBGI (GNU/Linux Mint)
!
!   cd ~/work
!   wget http://libxbgi.sourceforge.net/xbgi-364.tar.gz
!   tar -xf xbgi-364.tar.gz
!   cd xbgi-364/src
!   make
!   make demo
!
!   ./demo
!   make clean
!   cd test
!   make
!   ./mandelbrot
!
!   cd ..
!   mv libXbgi.a ~/programming/lib
!
! HOW TO BUILD WinBGIm-6.0 (MSYS2/MINGW64 shell)
!
!   cd ~/work/WinBGIm-6.0
!   make
!   mv libbgi.a ~/programming/lib/mingw64/libWinBGIm6.0.a
!
!   make clean
!
!   cd all-tests
!   g++ -O3 -Wall -mwindows -I .. test-bgidemo0.cxx \
!     -L ~/programming/lib/mingw64 -lWinBGIm6.0 \
!     -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32 -o test-bgidemo0
!
! HOW TO BUILD THE APP
!
!   cd ~/programming/nbody.apps
!
!   rm -rf {*.mod,~/programming/modules/*} && \
!   gfortran -O3 -Wall $BLD_OPTS -J ~/programming/modules \
!     ../basic-modules/{basic_mods.f90,julian_dates.f90} \
!     ../ode-modules/everhart_integrator.f90 \
!     ../bgi-fortran/{bgi.f90,bgiapp.f90} close_encounters.f90 \
!     -L ~/programming/lib/$PLATFORM $LIBS -o close_encounters$EXE
!
!   ./close_encounters$EXE
!
!   where:
!
!     BLD_OPTS =
!     PLATFORM =
!     LIBS = -lXbgi -lX11 -lm
!     EXE = .out
!
!   for the build on GNU/Linux
!
!     $BLD_OPTS = -static [-mwindows]
!     $PLATFORM = mingw64
!     $LIBS = -lWinBGIm6.0 -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32 \
!             -lstdc++
!     EXE =
!
!   for the build on MSYS2/MINGW64
!
!
! NICE WEB PAGES
!
!   https://phet.colorado.edu/sims/my-solar-system/my-solar-system_en.html
!   http://en.wikipedia.org/wiki/Numerical_model_of_the_Solar_System
!
! APOPHIS TEST
!
```

```
!   With T1 = 2477476.5 D, H = 0, LL = 16, the result is
!
!     CPA at time (D):      2462240.40684  (  2029   4  13.90684)
!     CPA distance |P1-P2| (AU):    2.5991332899197143E-004
!
!   Assuming AU ~ 150E6 km, |P1-P2| ~ 2.6E-4 * 150E6 = 39000 km
!   Wikipedia (http://it.wikipedia.org/wiki/99942_Apophis) says
!   |P1-P2| ~ 36350 km on April 13, 2029...
!

module close_encounters_lib
  use kind_consts, only: DP
  implicit none
  private
  !
  ! GAUSS Units : AU (A) for lengths, Day (D) for times,
  ! Solar Mass (MS) for masses.
  !
  ! With these units,
  !
  !   G*MSUN = MU = KQ = K*K = 2.959122082855910 x 10**(-4)
  !
  ! being K == 0.01720209895 A**(3/2) MS**(-1/2) D**(-1) the
  ! Gauss's Gravitational Constant
  !
  integer, parameter :: NDIM = 3, NDIM1 = NDIM-1, &
       MAX_NBODY = 15, MAX_NV = NDIM*MAX_NBODY, NCLASS = -2
  real(DP), parameter :: Z0 = 0, Z1 = 1
  integer :: body_color(MAX_NBODY) = 0, p1(NDIM) = 0, p2(NDIM) = 0, ll = 10
  ! NV is the number of 2nd order equations.
  integer :: nb, nb1, nv
  real(DP) :: t0 = Z0, t1 = 2454053.0_DP, h = Z0, &
       m(MAX_NBODY) = Z0, mm(MAX_NBODY) = Z0
  real(DP) :: k_view = 1000.0_DP, phi = 270.0_DP, theta = Z0, &
       rot_m(3,3) = Z0
  !
  ! We adopt the variables with this meaning (for example with 4 bodies in 3D)
  !
  !
  !   x(1:3)   = q1(1:3)
  !   x(4:6)   = q2(1:3)
  !   x(7:9)   = q3(1:3)
  !   x(10:12) = q4(1:3)
  !
  !   v(1:3)   = v1(1:3)
  !   v(4:6)   = v2(1:3)
  !   v(7:9)   = v3(1:3)
  !   v(10:12) = v4(1:3)
  !
  ! Notice that the first index sequences,
  !
  !   1   4   7   10
  !
  ! can be produced with
  !
  !   3*i-2,           i = 1,2,3,...,NBODY
  !
  ! i.e.
  !
  !   NDIM*i-(NDIM-1) = 1 + (i-1)*NDIM
  !
  real(DP) :: x(MAX_NV) = Z0, v(MAX_NV) = Z0

  public :: input_data, calc_orbit, run_app

contains

  subroutine read_cards()
    integer :: i, ip1, ip2, cards_unit
    real(DP) :: mu = Z0

    open(newunit = cards_unit, file = 'close_encounters.cards', status = 'old')

    ! Epoch of the data to be read (starting time of integration interval)
    read(cards_unit,*) t0

    ! Number of bodies
    read(cards_unit,*) nb
```

```
   if (nb > MAX_NBODY) then
      write(*,*) 'NB = ', nb, ' .GT. ', MAX_NBODY, '. Exiting...'
      stop
   end if
   nb1 = nb-1
   nv = NDIM*nb

   ! Gravitational parameter (G*M) for Sun (in AU**3/D**2)
   read(cards_unit,*) mu

   ! Planets data: gravitational parameter (in AU**3/D**2),
   ! positions (in AU) and velocities (in AU/D) at time t0
   ! Notice: m(1:nb) is the gravitational parameter (G*mass) NOT the mass..
   do i = 1, nb
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      read(cards_unit,*) m(i)
      read(cards_unit,*) x(ip1:ip2)
      read(cards_unit,*) v(ip1:ip2)
   end do

   ! Computing the constants : -(mu+m(i))
   mm(1:nb) = Z0-(mu+m(1:nb))

   ! The bodies for which we want the Closest Point Approach (CPA) data
   ! (should be in the range 1..nb and ip1 /= ip2)
   read(cards_unit,*) ip1, ip2

   if (ip1 == ip2 .or. (ip1 < 1 .or. nb < ip1) &
        .or. (ip2 < 1 .or. nb < ip2)) then
      write(*,*) 'P1 = ', ip1, ' P2 = ', ip2, &
           '. Wrong request for CPA. Exiting...'
      stop
   end if

   ! Now IP1 and IP2 point to the X coordinate of body P1 and P2,
   ! respectively...
   ip1 = 1+NDIM*(ip1-1)
   p1 = [ ip1, ip1+1, ip1+2 ]

   ip2 = 1+NDIM*(ip2-1)
   p2 = [ ip2, ip2+1, ip2+2 ]

   close(cards_unit)
   write(*,*) 'done!'

   ! Notice that, in the default example, the Earth orbit color (CYAN ?) is
   ! almost all overlapped by that of MOON (LIGHT_BLUE ?).
   ! Obviously, the colors could be defined differently...
   do i = 1, nb
      body_color(i) = i
   end do
end subroutine read_cards

subroutine input_data()
   use math_consts, only: DEG2RAD
   use get_data, only: get
   !
   ! The coordinates system is rectangular, heliocentric and ecliptic,
   ! which means a NOT-inertial reference system, where the SUN is ALWAYS
   ! at rest. See the note
   !
   !   M. Carpino, Introduzione ai metodi di calcolo delle effemeridi e
   !   determinazione orbitale.
   !
   !   (http://www.brera.mi.astro.it/~carpino/didattica/detorb.pdf)
   !
   ! and
   !
   !   G. Matarazzo, Moto perturbato degli N-corpi (Metodo di Cowell) risolto
   !   con l'integratore di Everhart al 15-esimo ordine.
   !
   !   (http://astrodinamica.altervista.org/PDF/MotoPert.pdf)
   !
   ! Just a clarification about the table on page 7 of the last cited work.
   ! The table does not report the close(st) encounters AS computed by
```

```
   ! COW.FOR program. This would mean to compute not only the distance
   ! of the close encounter but also the time at which this occurs.
   ! Instead the table shows only the positions at the times 'tf' of
   ! first column. The times 'tf' are the times of close(st) encounters
   ! AS computed by the astronomer E. Goffin.
   !
   ! This program tries to compute both times and distances of close(st)
   ! encounters. Obviously, we can verify the results of Goffin and
   ! COW.FOR ONLY approximately, in the limit of time step H and
   ! "precision" LL.
   !
   ! Another clarification. Often the data refer to the ecliptic plane
   ! with which most planetary orbits are almost co-planar. So an interesting
   ! point of view is on the equatorial plane. This forms an angle of about
   ! 23 degrees with the ecliptic plane. Put, then, the observer on the
   ! equatorial plane choosing a THETA angle of 90-23 = 67 degrees.
   !
   write(*,'(A)',advance='NO') 'Reading data...'
   call read_cards()
   write(*,*)
   write(*,*) 'Integration starts at time T0 (JD): ',t0
   write(*,*) 'Number of interacting bodies: ', nb
   write(*,*)

   ! The starting integration time, t0, is read from the cards file.
   ! The final time, t1, and the integration step (guess) is read here,
   ! interactively.
   call get('T1 (JD) = ',t1)
   call get('H (D) = ',h)
   write(*,*)

   call get('LL = ',ll)
   if (ll > 20) then
      write(*,*) 'LL TOO HIGH! Exiting... '
      stop
   end if
   write(*,*)

   call get('K_VIEW (AU) = ',k_view)
   call get('PHI (DEG) = ',phi)
   call get('THETA (DEG) = ',theta)
   write(*,*)

   ! Conversion to radians..
   phi = phi*DEG2RAD
   theta = theta*DEG2RAD

   ! With PHI and THETA we can compute ROT_M
   rot_m(1,1) = -sin(phi)
   rot_m(1,2) = cos(phi)
   rot_m(1,3) = Z0

   rot_m(2,3) = sin(theta)
   rot_m(3,3) = cos(theta)

   ! -cos(theta)*cos(phi), -cos(theta)*sin(phi)
   rot_m(2,1) = -rot_m(3,3)*rot_m(1,2)
   rot_m(2,2) = rot_m(3,3)*rot_m(1,1)

   ! sin(theta)*cos(phi), sin(theta)*sin(phi)
   rot_m(3,1) = rot_m(2,3)*rot_m(1,2)
   rot_m(3,2) = -rot_m(2,3)*rot_m(1,1)
 end subroutine input_data

 subroutine force(t,x,v,f)
   real(DP), intent(in) :: t, x(:), v(:)
   real(DP), intent(out) :: f(:)
   integer, save :: i, j, ip1, ip2, jp1, jp2
   real(DP), save :: a(NDIM*MAX_NBODY), d(NDIM)
   !
   ! Initialization of a(:) and field f(:).
   ! In a(:) we store
   !
   !   (r(p)/|r(p)|**3)
   !
   ! where r(p) is the radius vector of planet p from the Sun.
   !
   do i = 1, nb
```

```
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      ! d = qi/|qi|**3
      d = x(ip1:ip2)
      d = d/norm2(d)**3
      a(ip1:ip2) = d
      f(ip1:ip2) = mm(i)*a(ip1:ip2)
   end do

   ! Filling with forces/accelerations the field f(:)
   do i = 1, nb1
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      do j = i+1, nb
         jp1 = 1+NDIM*(j-1)
         jp2 = jp1+NDIM1

         ! d = (qi-qj)/|qi-qj|**3
         d = x(ip1:ip2)-x(jp1:jp2)
         d = d/norm2(d)**3

         f(ip1:ip2) = f(ip1:ip2)-m(j)*(d+a(jp1:jp2))
         f(jp1:jp2) = f(jp1:jp2)+m(i)*(d-a(ip1:ip2))
      end do
   end do
end subroutine force

subroutine calc_orbit()
   use everhart_integrator, only: radau_on, ra15, radau_off
   write(*,*)
   write(*,'(A)',advance='NO') 'Computing the orbits...'
   call radau_on(nv,ll,NCLASS,save_data_flag0=.true.)
   call ra15(t0,t1,x(1:nv),v(1:nv),h,force)
   call radau_off()
   write(*,*) '...done!'

   ! Just to test/debug...
   print *
   print '(a,f15.10,f15.4)', 'H,T =', h, t1
   print '(a,3f15.10)', 'Position (P2) =', x(p2)
   print '(a,3f15.10)', 'Position (P1) =', x(p1)
   print '(a,f15.8)', 'D =', norm2(x(p1)-x(p2))
   print *
end subroutine calc_orbit

subroutine do_projection(p,u,v)
   real(DP), intent(in) :: p(:)
   real(DP), intent(out) :: u, v
   real(DP) :: pv(3)

   pv = matmul(rot_m,p)

   v = (pv(3)/k_view)-Z1

   u = -pv(1)/v
   v = -pv(2)/v
end subroutine do_projection

subroutine run_app()
   use, intrinsic :: iso_fortran_env, only: iostat_end
   use julian_dates, only: jd2cal
   use bgi, only: BLUE, GREEN, RED, setcolor, YELLOW
   use bgiapp, only: bgiapp_dot, bgiapp_line
   real(DP), parameter :: DQ_THRESHOLD = (0.1_DP)**2
   integer  :: nv0, ll0, nclass0, ns, k, kp, data_unit, io_status, &
        year, month
   real(DP) :: h, t, us, vs, &
        dq, dq_min, d(NDIM), t_cpa, p1_cpa(NDIM), p2_cpa(NDIM), &
        dq_ce, t_ce, p1_ce(NDIM), p2_ce(NDIM), day
   real(DP) :: U_SUN, V_SUN
   logical :: find_ce

   ! Opening DATA file
   open(newunit = data_unit, file = 'ra15.data', access = 'stream', &
        form = 'unformatted', status = 'old')
   read(data_unit) nv0, ll0, nclass0
```

```
if (nv0 /= nv ) error stop '*** Mismatch for NV. ***'
if (ll0 /= ll ) error stop '*** Mismatch for LL. ***'
if (nclass0 /= NCLASS ) error stop '*** Mismatch for NCLASS. ***'

! Reading sequence n. 0, i.e. initial conditions. We do not test EOF
! because we assume at least a few sequences (NS > 1)
read(data_unit) ns, h, t, x(1:nv), v(1:nv)

! Just to test/debug...
!print *, 'NS,H,T,X,V =', ns, h, t, x(1:nv), v(1:nv)
!
! Initialization for Close-Encounter (CE) and Closest Point Approach (CPA)
!
! d      is the distance vector between P1 and P2
! t_ce   is the time at CE
! p1_ce  is the position of body P1 at CE
! p2_ce  is the position of body P2 at CE
!
! t_cpa   is the time at CPA
! p1_cpa  is the position of body P1 at CPA
! p2_cpa  is the position of body P2 at CPA
!
! We try to find CEs which are below DQ_THRESHOLD (distance squared
! threshold), i.e. when the flag FIND_CE is set. This occurs the first
! time that DQ < DQ_THRESHOLD, for current search).
!
! We can lose CEs in certain situations. For example, if bodies are at CE,
! i.e. below DQ_THRESHOLD, when we start the integration.
!
! If we start above DQ_THRESHOLD, we should be able to find all the
! CE < DQ_THRESHOLD.
!
find_ce = .false.
d = x(p1)-x(p2)
dq = dot_product(d,d)
dq_ce = dq
t_ce = t
p1_ce = x(p1)
p2_ce = x(p2)

! Being the CPA the minimum of all CE, dq_min is the minimum of all dq_ce.
dq_min = dq_ce
t_cpa = t_ce
p1_cpa = p1_ce
p2_cpa = p2_ce

! Plotting the SUN and the axes
! The Sun position, i.e. the origin of Heliocentric System: notice that
! here we compute ONLY the position. We do not plot the SUN..
call do_projection([ Z0, Z0, Z0 ],U_SUN,V_SUN)

! First, we plot the axes...
! X axis
call setcolor(RED)
call do_projection([ 15*Z1, Z0, Z0 ],us,vs)
call bgiapp_line(U_SUN,V_SUN,us,vs)

! Y axis
call setcolor(GREEN)
call do_projection([ Z0, 15*Z1, Z0 ],us,vs)
call bgiapp_line(U_SUN,V_SUN,us,vs)

! Z axis
call setcolor(BLUE)
call do_projection([ Z0, Z0, 15*Z1 ],us,vs)
call bgiapp_line(U_SUN,V_SUN,us,vs)

! ...then we plot the SUN!!!
call bgiapp_dot(U_SUN,V_SUN,YELLOW)

do
   ! Plotting planets at current position
   do k = 1, nb
      kp = 1+NDIM*(k-1)

      call do_projection(x(kp:kp+2),us,vs)
      call bgiapp_dot(us,vs,body_color(k))
```

```
      end do

      ! We take another step...
      read(data_unit,iostat=io_status) ns, h, t, x(1:nv), v(1:nv)

      if (io_status == iostat_end) exit
      if (io_status > 0) &
           error stop '*** Error occurred while reading file. ***'

      d = x(p1)-x(p2)
      dq = dot_product(d,d)

      ! We are entering the "region" DQ < DQ_THRESHOLD. Hunting can begin...
      if (.not. find_ce .and. dq < DQ_THRESHOLD) find_ce = .true.

      ! We are leaving the "region" DQ < DQ_THRESHOLD. Hunting stops...
      ! ...and we emptied its pouch, i.e. we output the result and
      ! reset essential variables.. DQ_CE is reset to DQ which
      ! is >= DQ_THRESHOLD!
      if (find_ce .and. dq >= DQ_THRESHOLD) then
         call jd2cal(t_ce,1,year,month,day)
         write(*,*)
         write(*,*)
         write(*,'(a,f18.5,a,i6,i4,f10.5,a)') 'CE at time (D): ', t_ce, &
              '  (', year, month, day, ')'
         write(*,*) 'CE P1 position (AU): ', p1_ce
         write(*,*) 'CE P2 position (AU): ', p2_ce
         write(*,*) 'CE distance |P1-P2| (AU): ', sqrt(dq_ce)

         ! We have found a CE.. but is this also the CPA?
         if (dq_ce < dq_min) then
            dq_min = dq_ce
            t_cpa = t_ce
            p1_cpa = p1_ce
            p2_cpa = p2_ce
         end if

         ! Reset of the relevant variables for the next search...
         find_ce = .false.
         dq_ce = dq
      end if

      ! If we are hunting, let's see if we are close the prey..
      if (find_ce .and. (dq < dq_ce)) then
         dq_ce = dq
         t_ce = t
         p1_ce = x(p1)
         p2_ce = x(p2)
      end if
    end do
    call jd2cal(t_cpa,1,year,month,day)
    write(*,*)
    write(*,*)
    write(*,'(a,f18.5,a,i6,i4,f10.5,a)') 'CPA at time (D): ', t_cpa, &
         '  (', year, month, day, ')'
    write(*,*) 'CPA P1 position (AU): ', p1_cpa
    write(*,*) 'CPA P2 position (AU): ', p2_cpa
    write(*,*) 'CPA distance |P1-P2| (AU): ', sqrt(dq_min)

    close(data_unit)
  end subroutine run_app
end module close_encounters_lib

program close_encounters
  use kind_consts, only: DP
  use general_routines, only: system_time
  use bgiapp, only: bgiapp_setup, bgiapp_init, bgiapp_close
  use close_encounters_lib
  implicit none
  real(DP) :: t0, t1

  call input_data()
  call bgiapp_setup(-5.0_DP,5.0_DP,-5.0_DP,5.0_DP,900,900)

  t0 = system_time()
  call calc_orbit()
  t1 = system_time()-t0
```

```
  write(*,*)
  write(*,'(A,F9.3,A)') 'Completed in ',t1,' seconds!'

  call bgiapp_init('Close Encounters in 3D')

  write(*,*)
  write(*,'(A)',advance='NO') 'Please wait, we are working...'

  t0 = system_time()
  call run_app()
  t1 = t1+system_time()-t0

  write(*,*)
  write(*,'(A,F9.3,A)') 'Completed (TOTAL time) in ',t1,' seconds!'

  call bgiapp_close()
end program close_encounters
```

```
2450400.5E0             ! Epoch of the following data: t0 in JD (1996.11.13)
13                      ! Num. of bodies: 9_PLANETS + MOON + 3_ASTEROID
2.9591220828559109E-04   ! SUN gravitational parameter (GM) in AU**3/D**2
4.9125495718310926E-011  ! MERCURY data: mu, P, V
-1.491597147372767E-01 -4.409630314852908E-01 -2.232760294282906E-02
2.099935561673024E-02 -7.614588783381691E-03 -2.549599303902789E-03
7.2434531799395128E-010  ! VENUS data: mu, P, V
-6.119893135637021E-01  3.744913412372688E-01  4.044173605869310E-02
-1.064020054248585E-02 -1.735242543146333E-02  3.772384910458827E-04
8.8876925468881312E-010  ! EARTH data: mu, P, V
6.235834212190300E-01  7.683399260131534E-01  4.751567992370535E-06
-1.364628508960836E-02  1.077936535027717E-02 -1.828322890139825E-07
1.0931889900447183E-011  ! MOON data: mu, P, V
6.229102892685170E-01  7.659462392724490E-01  2.106771983479050E-04
-1.305453769069843E-02  1.063212641103732E-02  1.967671914792972E-05
9.5495319248992543E-011  ! MARS data: mu, P, V
-8.678762425345101E-01  1.387156048737002E+00  5.039199333124769E-02
-1.133319250585803E-02 -6.233312585307682E-03  1.480613776554786E-04
2.8247604533651827E-007  ! JUPITER data: mu, P, V
2.082087540769299E+00 -4.715767658128386E+00 -2.710998509201615E-02
6.809364256702234E-03  3.403616809968890E-03 -1.665752961137376E-04
8.4576151711855840E-008  ! SATURN data: mu, P, V
9.431816319591139E+00  9.254821134396185E-01 -3.911315416992688E-01
-8.404643760719874E-04  5.546167970574444E-03 -6.334347795242694E-05
1.2918949220207392E-008  ! URANUS data: mu, P, V
1.102987764625917E+01 -1.643087121705927E+01 -2.039897402435908E-01
3.243004224175503E-03  2.014199077781569E-03 -3.460988677815098E-05
1.5240407045482162E-008  ! NEPTUNE data: mu, P, V
1.374054785933357E+01 -2.684413121570826E+01  2.360069346593258E-01
2.780267752478550E-03  1.453067809066634E-03 -9.353788642859004E-05
1.9452118462049880E-012  ! PLUTO data: mu, P, V
-1.327085889240211E+01 -2.600534944576927E+01  6.622324629725974E+00
2.903988680023795E-03 -1.866045588646760E-03 -6.311262844723530E-04
2.2297247205467541E-020  ! 1620 Geographos data: mu, P, V
-7.060485772092238E-01  1.252231067126121E+00  2.095930365992838E-01
-8.952871398986725E-03 -9.046046376125343E-03 -2.797321808419076E-03
6.85E-024                ! 99942 Apophis data: mu, P, V
7.107062136151633E-01  3.894508051529238E-01 -3.274807289758988E-03
-7.016197192797620E-03  1.896184795079028E-02 -1.172780694723344E-03
14.03E-014               ! 1 Ceres data: mu, P, V
6.159275815999015E-01 -2.820904493849575E+00 -1.993609022032636E-01
9.599130262761093E-03  1.594095019532529E-03 -1.723388645820340E-03
3 12                     ! The bodies for which we want the CPA data

!
! The above data have been generated with JPL Horizons WEB Interface.
!
! The data refers to:
!
!   Sun body centered
!   Earth (Geocenter)
!   Vector table
!   Reference epoch: J2000.0
!   Coordinate system: Ecliptic and Mean Equinox of Reference Epoch
!
! At JPL WEB site, the gravitational parameters are expressed in km**3/s**2,
! so we have expressed them in AU**3/D**2 with planet_state_vector.f90
! program.
!
! The GM for Apophis has been computed with these data:
!
!   M = 4.6E10 kg         (from Wikipedia, italian version)
!   MSUN = 1.98855E30 kg  (from Wikipedia, english version)
!   G = G*MSUN = 2.9591220828559109E-04 (third line above)
!
! So,
!
!   GM = (4.6E10/1.98855E30)*2.9591220828559109E-04
!      = (4.6/1.98855)*2.9591220828559109 * 1E-24
!      = 6.85E-24
!
! For Ceres, Wikipedia says: M = 9.43E20 kg. So with the same steps:
!
!   GM = (9.43E20/1.98855E30)*2.9591220828559109E-04
!      = (9.43/1.98855)*2.9591220828559109 * 1E-14
!      = 14.03E-14
!
! JPL gives: GM = 63.2 km**3/s**2. Assuming 1 AU ~ 150E6 km, D = 86400 s, a
```

```
! raw extimate gives:
!
!   GM = 63.2 * (86400)**2 / (150E6)**3 = 63.2 * ((8.64)**2 / 1.5**3) * 1E-16
!      = 1397.88 * 1E-16 ~ 13.98 * 1E-14 ~ 14E-14
!
! The "1" in "1 Ceres" means that Ceres was the first asteroid discovered
! (by G. piazzi).
!
```

```
!
! Fortran Interface to the Xbgi-364p Library
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! HOW TO BUILD (GNU/Linux Mint)
!
!   cd ~/work
!   wget http://libxbgi.sourceforge.net/xbgi-364.tar.gz
!   tar -xf xbgi-364.tar.gz
!   cd xbgi-364/src
!   make
!   make demo
!
!   ./demo
!   make clean
!   cd test
!   make
!   ./mandelbrot
!
!   cd ..
!   mv libXbgi.a ~/programming/lib
!   cd ~/programming/nbody.apps
!
!   rm -rf {*.mod,~/programming/modules/*} && \
!   gfortran -O3 -Wall -J ../modules \
!     ../basic-modules/basic_mods.f90 \
!     ../ode-modules/ode_integrators.f90 \
!     ../bgi-fortran/{bgi.f90,bgiapp.f90} planar3body.f90 \
!     -L ../lib -lXbgi -lX11 -lm -o planar3body.out
!
!   ./planar3body.out
!
!
! WEB SITES/DOCS
!
!   http://it.wikipedia.org/wiki/Orbita_osculatrice
!
!   Osculating orbits in the Pythagorean 3-Body problem (video, youtube)
!   https://www.youtube.com/watch?v=rr0JpgKPKgg
!
! EXAMPLES
!
!   t in [0,300], M(:) = [200E4, 10E4, 0.001E4],
!   P1(-6.76266,0), P2(135.237,0), P3(159.237,0)
!   V1(0,-6.71461), V2(0,134.285), V3(0,68.1902)
!
! position and speeds are referred to the CM system. This means, the CM
! is at rest. This is true approximately, in the limit of numeric precision
! and you should consider also that we have computed postions and speeds
! with six significat figures.
!

module planar3body_lib
  use kind_consts, only: DP
  implicit none
  private

  ! Units so that G_NEWTON = 1 (G_NEWTON = 6.67E-11 in SI)
  integer :: id_method = 1
  integer, parameter :: NDIM = 2, NBODY = 3, NEQ = 2*NDIM*NBODY
  real(DP) :: t0 = 0.0_DP, t1 = 10.0_DP, h = 0.00005_DP, eps = 1.0E-12_DP, &
       m(NBODY) = [ 5.0_DP, 3.0_DP, 4.0_DP ]
  !
  ! We adopt the equation found in
  !
  !   D. Gruntz - J. Waldvogel "Orbits in te Planar Three-Body Problem"
  !
  ! i.e.
  !
  !   y(1:2)   = q1(1:2)
  !   y(3:4)   = v1(1:2)
  !
  !   y(5:6)   = q2(1:2)
```

```
  !   y(7:8)   = v2(1:2)
  !
  !   y(9:10)  = q3(1:2)
  !   y(11:12) = v3(1:2)
  !
  ! w(:,:) work space to compute K1, K2, K3, K4. Notice that the method uses
  ! w(:,3) and NOT w(:,4)!
  real(DP) :: y0(NEQ) = [ 1.0_DP, -1.0_DP, 0.0_DP, 0.0_DP, &
       1.0_DP, 3.0_DP, 0.0_DP, 0.0_DP, &
       -2.0_DP, -1.0_DP, 0.0_DP, 0.0_DP ]

  public :: input_data, run_app

contains

  subroutine input_data()
    use get_data, only: get
    real(DP), parameter :: MACHEPS = epsilon(1.0_DP)

    write(*,*) 'Choose the method:'
    write(*,*) '  1 : RK4'
    write(*,*) '  2 : GBS'
    write(*,*) '  3 : RKM'
    call get('ID_METHOD =',id_method)
    ! For GBS or RKM step, the default initial H step can be greather..
    if (id_method == 2 .or. id_method == 3) h = 0.005_DP
    write(*,*)

    call get('T0 = ',t0)
    call get('T1 = ',t1)
    call get('H = ',h)
    if (id_method == 2 .or. id_method == 3) then
       write(*,*)
       call get('EPS = ',eps)
       if (eps < 1000*MACHEPS) then
          write(*,*) 'EPS TOO SMALL! Exiting... '
          stop
       end if
    end if
    write(*,*)

    write(*,*) 'Masses:'
    call get('M1 = ',m(1))
    call get('M2 = ',m(2))
    call get('M3 = ',m(3))
    write(*,*)

    write(*,*) 'Initial position:'
    call get('X1 = ',y0(1))
    call get('Y1 = ',y0(2))
    write(*,*)

    call get('X2 = ',y0(5))
    call get('Y2 = ',y0(6))
    write(*,*)

    call get('X3 = ',y0(9))
    call get('Y3 = ',y0(10))
    write(*,*)

    write(*,*) 'Initial speed:'
    call get('VX1 = ',y0(3))
    call get('VY1 = ',y0(4))
    write(*,*)

    call get('VX2 = ',y0(7))
    call get('VY2 = ',y0(8))
    write(*,*)

    call get('VX3 = ',y0(11))
    call get('VY3 = ',y0(12))
    write(*,*)
  end subroutine input_data

  subroutine sub(x,y,f)
    real(DP), intent(in) :: x, y(:)
    real(DP), intent(out) :: f(:)
    real(DP), save :: d1(NDIM), d2(NDIM),d3(NDIM)
```

```
    !
    ! y(1:2)   = q1(1:2)
    ! y(3:4)   = v1(1:2)
    !
    ! y(5:6)   = q2(1:2)
    ! y(7:8)   = v2(1:2)
    !
    ! y(9:10)  = q3(1:2)
    ! y(11:12) = v3(1:2)
    !

    ! d1 = (q3-q2)/|q3-q2|**3
    d1 = y(9:10)-y(5:6)
    d1 = d1/norm2(d1)**3

    ! d2 = (q1-q3)/|q1-q3|**3
    d2 = y(1:2)-y(9:10)
    d2 = d2/norm2(d2)**3

    ! d3 = (q2-q1)/|q2-q1|**3
    d3 = y(5:6)-y(1:2)
    d3 = d3/norm2(d3)**3

    ! Now computing the field
    f(1:2) = y(3:4)
    f(5:6) = y(7:8)
    f(9:10) = y(11:12)

    f(3:4) = m(2)*d3-m(3)*d2
    f(7:8) = m(3)*d1-m(1)*d3
    f(11:12) = m(1)*d2-m(2)*d1
  end subroutine sub

  subroutine run_app()
    use bgi, only: GREEN, RED, YELLOW
    use bgiapp, only: bgiapp_dot
    use ode_integrators, only: rk4step, deqgbs, deqrkm
    ! For RK4 w(NEQ,3) would be sufficient...
    ! For RKM w(NEQ,6) would be sufficient...
    ! For GBS we need w(NEQ,36)...
    real(DP) :: t, tz, y(NEQ), w(NEQ,36), h0

    h0 = h
    t = t0
    y = y0
    do while (t < t1)

       call bgiapp_dot(y(1),y(2),GREEN)
       call bgiapp_dot(y(5),y(6),RED)
       call bgiapp_dot(y(9),y(10),YELLOW)

       ! We take an ode integrator step
       if (id_method == 1) then
          call rk4step(NEQ,h,t,y,w,sub)
       else
          h = h0
          tz = t+h
          if (id_method == 2) then
             call deqgbs(NEQ,t,tz,y,h,eps,w,sub)
          else
             call deqrkm(NEQ,t,tz,y,h,eps,w,sub)
          end if
          t = tz
       end if
    end do
    !print *
    !print *, t,y
  end subroutine run_app
end module planar3body_lib

program planar3body
  use kind_consts, only: DP
  use general_routines, only: system_time
  use bgiapp, only: bgiapp_setup, bgiapp_init, bgiapp_close
  use planar3body_lib
  implicit none

  real(DP) :: t0, t1
```

```
  call input_data()
  call bgiapp_setup(-5.0_DP,5.0_DP,-5.0_DP,5.0_DP)
  call bgiapp_init('3-Body Planar Orbits')

  write(*,'(A)',advance='NO') 'Please wait, we are working...'

  t0 = system_time()
  call run_app()
  t1 = system_time()

  write(*,*)
  write(*,'(A,F9.3,A)') 'Completed in ',t1-t0,' seconds!'

  call bgiapp_close()
end program planar3body
```

```
!
! Fortran Interface to the Xbgi-364p Library
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! HOW TO BUILD (GNU/Linux Mint)
!
!   cd ~/work
!   wget http://libxbgi.sourceforge.net/xbgi-364.tar.gz
!   tar -xf xbgi-364.tar.gz
!   cd xbgi-364/src
!   make
!   make demo
!
!   ./demo
!   make clean
!   cd test
!   make
!   ./mandelbrot
!
!   cd ..
!   mv libXbgi.a ~/programming/lib
!   cd ~/programming/nbody.apps
!
!   rm -rf {*.mod,~/programming/modules/*} && \
!   gfortran -O3 -Wall -J ../modules \
!     ../basic-modules/basic_mods.f90 \
!     ../ode-modules/{ode_integrators.f90,everhart_integrator.f90} \
!     ../bgi-fortran/{bgi.f90,bgiapp.f90} jsunp.f90 \
!     -L ../lib -lXbgi -lX11 -lm -o jsunp.out
!
!   ./jsunp.out
!
!
! While developping, you should compile with these options:
!
!   gfortran[-mp-4.9] -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1 \
!   -g -fcheck=all -fbacktrace...
!
! REFERENCES
!
!   http://inis.jinr.ru/sl/vol1/CMC/
!   Hairer,_Numerical_Geometric_Integration,1999.pdf
!
! The data refers to Sept. 5, 1994 00:00, i.e. JD 2449600.5
!

module jsunp_lib
  use kind_consts, only: DP
  implicit none
  private
  !
  ! GAUSS Units : AU (A) for lengths, Day (D) for times,
  ! Solar Mass (MSUN) for masses:
  !
  !   MSUN = 1, AU = 149597870 km, G = 2.95912208286 x 10**(-4)
  !
  integer, parameter :: NDIM = 3, NDIM1 = NDIM-1, &
       NB = 5, NB1 = NB-1, NV = NDIM*NB, NEQ = 2*NV, NCLASS = -2
  integer, parameter :: RK4_ID = 1, GBS_ID = 2, RKM_ID = 3, RA15_ID = 4
  real(DP), parameter :: Z0 = 0, Z1 = 1, GN = 2.95912208286E-04_DP
  integer :: body_color(NB) = 0, id_method = RKM_ID, ll = 10
  real(DP) :: t0 = Z0, t1 = 100000.0_DP, h = 0.125_DP, eps = 1.0E-9_DP, &
       m(NB) = [ 0.000954786104043_DP, &
       0.000285583733151_DP, 0.0000437273164546_DP, 0.0000517759138449_DP, &
       Z1/1.3E08_DP ], mm(NB) = Z0
  !
  ! We are looking at the scene from the distance K_VIEW,
  ! in the direction (phi,theta). ROT_M is the matrix to transform
  ! (X,Y,Z) to (XV,YV,ZV). See QFA2, pag. 175+
  !
  real(DP) :: k_view = 1000.0_DP, phi = 270.0_DP, theta = 90.0_DP, &
       rot_m(3,3) = Z0
  !
```

```
  ! We adopt the variables with this meaning (for example with 4 bodies in 3D)
  !
  !
  !   y(1:3)   = q1(1:3)
  !   y(4:6)   = q2(1:3)
  !   y(7:9)   = q3(1:3)
  !   y(10:12) = q4(1:3)
  !
  !   y(13:15) = v1(1:3)
  !   y(16:18) = v2(1:3)
  !   y(19:21) = v3(1:3)
  !   y(22:24) = v4(1:3)
  !
  ! Notice that the first index sequences,
  !
  !   1   4   7   10
  !
  !   13  16  19  22
  !
  ! can be produced with
  !
  !   3*i-2,           i = 1,2,3,...,NBODY
  !
  !   3*i-2 + NEQ/2,   i = 1,2,3,...,NBODY
  !
  ! i.e.
  !
  !   NDIM*i-(NDIM-1) = 1 + (i-1)*NDIM
  !
  !   NDIM*i-(NDIM-1) + NEQ/2 = 1 + (i-1)*NDIM + NEQ/2
  !
  ! These data uses an Heliocentric equatorial rectangular coordinates system
  real(DP) :: y0(NEQ) = [ -3.5023653_DP, -3.8169847_DP, -1.5507963_DP, &
       9.0755314_DP, -3.0458353_DP, -1.6483708_DP, &
       8.3101420_DP, -16.2901086_DP, -7.2521278_DP, &
       11.4707666_DP, -25.7294829_DP, -10.8169456_DP, &
       -15.5387357_DP, -25.2225594_DP, -3.1902382_DP, &
       0.00565429_DP, -0.00412490_DP, -0.00190589_DP, &
       0.00168318_DP, 0.00483525_DP, 0.00192462_DP, &
       0.00354178_DP, 0.00137102_DP, 0.00055029_DP, &
       0.00288930_DP, 0.00114527_DP, 0.00039677_DP, &
       0.0027672_DP, -0.00170702_DP, -0.00136504_DP ]

  public :: input_data, run_app

contains
  subroutine input_data()
    use math_consts, only: DEG2RAD
    use get_data, only: get
    real(DP), parameter :: MACHEPS = epsilon(Z1), &
         MU0 = GN*1.00000597682_DP  ! MU for SUN+inner planets
    integer :: i
    write(*,*) 'Choose the method:'
    write(*,*) '  1 : RK4'
    write(*,*) '  2 : GBS'
    write(*,*) '  3 : RKM'
    write(*,*) '  4 : RA15'
    call get('ID_METHOD =',id_method)
    ! For GBS, RKM or RA15 step, the default initial H step can be greather..
    if (id_method /= RK4_ID) h = 0.25_DP
    write(*,*)
    call get('T0 (D) = ',t0)
    call get('T1 (D) = ',t1)
    call get('H (D) = ',h)
    if (id_method == GBS_ID .or. id_method == RKM_ID) then
       write(*,*)
       call get('EPS = ',eps)
       if (eps < 1000*MACHEPS) then
          write(*,*) 'EPS TOO SMALL! Exiting... '
          stop
       end if
    end if
    if (id_method == RA15_ID) then
       write(*,*)
       call get('LL = ',ll)
       if (ll > 20) then
          write(*,*) 'LL TOO HIGH! Exiting... '
          stop
```

```
      end if
    end if
    write(*,*)

    call get('K_VIEW (AU) = ',k_view)
    call get('PHI (DEG) = ',phi)
    call get('THETA (DEG) = ',theta)
    write(*,*)

    ! Conversion to radians..
    phi = phi*DEG2RAD
    theta = theta*DEG2RAD

    ! With PHI and THETA we can compute ROT_M
    rot_m(1,1) = -sin(phi)
    rot_m(1,2) = cos(phi)
    rot_m(1,3) = Z0

    rot_m(2,3) = sin(theta)
    rot_m(3,3) = cos(theta)

    ! -cos(theta)*cos(phi), -cos(theta)*sin(phi)
    rot_m(2,1) = -rot_m(3,3)*rot_m(1,2)
    rot_m(2,2) = rot_m(3,3)*rot_m(1,1)

    ! sin(theta)*cos(phi), sin(theta)*sin(phi)
    rot_m(3,1) = rot_m(2,3)*rot_m(1,2)
    rot_m(3,2) = -rot_m(2,3)*rot_m(1,1)

    ! Converting masses, m(i), (in MSUN units) to
    ! Gravitational Parameters, mu(i)
    !
    ! HERE we use an Heliocentric Reference System (HRS)
    m(1:NB) = GN*m(1:NB)

    ! Computing the constants : -(MU0+mu(i))
    mm(1:NB) = Z0-(MU0+m(1:NB))

    ! Obviously, the colors could be defined differently...
    do i = 1, NB
       body_color(i) = i+5
    end do
  end subroutine input_data

  subroutine do_projection(p,u,v)
    real(DP), intent(in) :: p(:)
    real(DP), intent(out) :: u, v
    real(DP) :: pv(3)

    pv = matmul(rot_m,p)

    v = (pv(3)/k_view)-Z1

    u = -pv(1)/v
    v = -pv(2)/v
  end subroutine do_projection

  subroutine sub(x,y,f)
    real(DP), intent(in) :: x, y(:)
    real(DP), intent(out) :: f(:)
    ! ip... point to the first half of arrays, iv... to the seconf half.
    ! The same for jp..., jv...
    integer, save :: i, j, ip1, ip2, jp1, jp2, iv1, iv2, jv1, jv2
    real(DP), save :: a(NDIM*NB), d(NDIM)
    !
    ! y(1:2)   = q1(1:2)
    ! y(3:4)   = q2(1:2)
    ! y(5:6)   = q3(1:2)
    !
    ! y(7:8)   = v1(1:2)
    ! y(9:10)  = v2(1:2)
    ! y(11:12) = v3(1:2)
    !

    ! Filling/initializing with speeds the first half of field f(:)
    f(1:NV) = y(NV+1:NEQ)

    !
```

```
   ! Initialization of a(:) and second half of field f(:).
   ! In a(:) we store
   !
   !   (r(p)/|r(p)|**3)
   !
   ! where r(p) is the radius vector of planet p from the Sun.
   !
   do i = 1, NB
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      iv1 = NV+ip1
      iv2 = iv1+NDIM1

      ! d = qi/|qi|**3
      d = y(ip1:ip2)
      d = d/norm2(d)**3
      a(ip1:ip2) = d
      f(iv1:iv2) = mm(i)*a(ip1:ip2)
   end do

   ! Filling with forces/accelerations the second half of field f(:)
   do i = 1, NB1
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      iv1 = NV+ip1
      iv2 = iv1+NDIM1

      do j = i+1, NB
         jp1 = 1+NDIM*(j-1)
         jp2 = jp1+NDIM1

         jv1 = NV+jp1
         jv2 = jv1+NDIM1

         ! d = (qi-qj)/|qi-qj|**3
         d = y(ip1:ip2)-y(jp1:jp2)
         d = d/norm2(d)**3

         f(iv1:iv2) = f(iv1:iv2)-m(j)*(d+a(jp1:jp2))
         f(jv1:jv2) = f(jv1:jv2)+m(i)*(d-a(ip1:ip2))
      end do
   end do
end subroutine sub

subroutine force(t,x,v,f)
   real(DP), intent(in) :: t, x(:), v(:)
   real(DP), intent(out) :: f(:)
   integer, save :: i, j, ip1, ip2, jp1, jp2
   real(DP), save :: a(NDIM*NB), d(NDIM)
   !
   ! Initialization of a(:) and field f(:).
   ! In a(:) we store
   !
   !   (r(p)/|r(p)|**3)
   !
   ! where r(p) is the radius vector of planet p from the Sun.
   !
   do i = 1, NB
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      ! d = qi/|qi|**3
      d = x(ip1:ip2)
      d = d/norm2(d)**3
      a(ip1:ip2) = d
      f(ip1:ip2) = mm(i)*a(ip1:ip2)
   end do

   ! Filling with forces/accelerations the field f(:)
   do i = 1, NB1
      ip1 = 1+NDIM*(i-1)
      ip2 = ip1+NDIM1

      do j = i+1, NB
         jp1 = 1+NDIM*(j-1)
         jp2 = jp1+NDIM1
```

```
        ! d = (qi-qj)/|qi-qj|**3
        d = x(ip1:ip2)-x(jp1:jp2)
        d = d/norm2(d)**3

        f(ip1:ip2) = f(ip1:ip2)-m(j)*(d+a(jp1:jp2))
        f(jp1:jp2) = f(jp1:jp2)+m(i)*(d-a(ip1:ip2))
      end do
    end do
  end subroutine force

  subroutine run_app()
    use bgi, only: BLUE, GREEN, RED, setcolor, YELLOW
    use bgiapp, only: bgiapp_dot, bgiapp_line
    use ode_integrators, only: rk4step, deqgbs, deqrkm
    use everhart_integrator, only: radau_on, ra15, radau_off
    integer  :: k, kp
    ! For RK4 w(NEQ,3) would be sufficient...
    ! For RKM w(NEQ,6) would be sufficient...
    ! For GBS we need w(NEQ,36)...
    real(DP) :: t, tz, y(NEQ), w(NEQ,36), h0, sgh, u, v
    real(DP) :: U_SUN, V_SUN
    logical :: first

    ! Switching on the Everhart integrator if it has been selected...
    ! LL = 12, NCLASS = -2
    if (id_method == RA15_ID) call radau_on(NV,ll,NCLASS)

    ! Initialization for forward/backward integration
    h = sign(abs(h),t1-t0)
    sgh = sign(Z1,h)

    ! General initialization for integration
    h0 = h
    t = t0
    y(1:NEQ) = y0(1:NEQ)

    ! Initialization for plotting SUN
    first = .true.

    do while (sgh*(t+h0-t1) < 0)

       if (first) then
          first = .false.

          ! The Central Body position, i.e. the origin of Heliocentric System
          call do_projection([ Z0, Z0, Z0 ],U_SUN,V_SUN)

          ! X axis
          call setcolor(RED)
          call do_projection([ 15*Z1, Z0, Z0 ],u,v)
          call bgiapp_line(U_SUN,V_SUN,u,v)

          ! Y axis
          call setcolor(GREEN)
          call do_projection([ Z0, 15*Z1, Z0 ],u,v)
          call bgiapp_line(U_SUN,V_SUN,u,v)

          ! Z axis
          call setcolor(BLUE)
          call do_projection([ Z0, Z0, 15*Z1 ],u,v)
          call bgiapp_line(U_SUN,V_SUN,u,v)

          ! The SUN-Central Body!!!
          call bgiapp_dot(U_SUN,V_SUN,YELLOW)
       end if

       do k = 1, NB
          kp = 1+NDIM*(k-1)

          call do_projection(y(kp:kp+2),u,v)
          call bgiapp_dot(u,v,body_color(k))
       end do

       ! We take an ode integrator step
       if (id_method == RK4_ID) then
          call rk4step(NEQ,h,t,y,w,sub)
       else
```

```
          h = h0
          tz = t+h
          select case (id_method)
          case (GBS_ID)
             call deqgbs(NEQ,t,tz,y,h,eps,w,sub)
          case (RKM_ID)
             call deqrkm(NEQ,t,tz,y,h,eps,w,sub)
          case (RA15_ID)
             call ra15(t,tz,y(1:NV),y(NV+1:NEQ),h,force)
          end select
          t = tz
       end if
    end do

    ! Switching OFF the Everhart integrator if it is ON...
    if (id_method == RA15_ID) call radau_off()
  end subroutine run_app
end module jsunp_lib

program jsunp
  use kind_consts, only: DP
  use general_routines, only: system_time
  use bgiapp, only: bgiapp_setup, bgiapp_init, bgiapp_close
  use jsunp_lib
  implicit none
  real(DP) :: t0, t1

  call input_data()
  call bgiapp_setup(-50.0_DP,50.0_DP,-50.0_DP,50.0_DP,900,900)
  call bgiapp_init('Outer Solar System in 3D')

  write(*,'(A)',advance='NO') 'Please wait, we are working...'

  t0 = system_time()
  call run_app()
  t1 = system_time()

  write(*,*)
  write(*,'(A,F9.3,A)') 'Completed in ',t1-t0,' seconds!'

  call bgiapp_close()
end program jsunp
```

```
!
! Fortran Interface to the Xbgi-364p Library
! by Angelo Graziosi  (firstname.lastnameATalice.it)
! Copyright Angelo Graziosi
!
! It is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
!
! HOW TO BUILD (GNU/Linux Mint)
!
!   cd ~/work
!   wget http://libxbgi.sourceforge.net/xbgi-364.tar.gz
!   tar -xf xbgi-364.tar.gz
!   cd xbgi-364/src
!   make
!   make demo
!
!   ./demo
!   make clean
!   cd test
!   make
!   ./mandelbrot
!
!   cd ..
!   mv libXbgi.a ~/programming/lib
!   cd ~/programming/nbody.apps
!
!   rm -rf {*.mod,~/programming/modules/*} && \
!   gfortran -O3 -Wall -J ../modules \
!     ../basic-modules/basic_mods.f90 \
!     ../ode-modules/{ode_integrators.f90,everhart_integrator.f90} \
!     ../bgi-fortran/{bgi.f90,bgiapp.f90} kepler_problem.f90 \
!     -L ../lib -lXbgi -lX11 -lm -o kepler_problem.out
!
!   ./kepler_problem.out
!
!
! While developping, you should compile with these options:
!
!   gfortran[-mp-4.9] -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1 \
!   -g -fcheck=all -fbacktrace...
!
! Kepler's Problem from this note:
!
!   http://inis.jinr.ru/sl/vol1/CMC/
!   Hairer,_Numerical_Geometric_Integration,1999.pdf
!
! and this
!
!   http://wwwusers.ts.infn.it/~gregorio/lessons/cap_iii.pdf
!

module kepler_problem_lib
  use kind_consts, only: DP
  implicit none
  private
  !
  ! You can think that the units are such that m = 1 GM = mu = 1, and
  ! eqs. of motion become
  !
  !   r''(:) = - mu*r(:)/r**3 = -r(:)/r**3
  !
  ! where r(:) - [q1,q2] (being a plane problem, qui stay on that plane..),
  ! and r = |r(:)|.
  !
  ! In these units, the energy is
  !
  !   H_0 = (1/2)*m*v**2 - mu/r = -mu/(2*a) = -1/(2*a)
  !
  ! and tha angular momentum
  !
  !   L_0 = sqrt(mu*p) = sqrt(p)
  !
  ! with p = a*(1-e**2)
  !
  integer, parameter :: NV = 2, NEQ = 2*NV, NCLASS = -2
  integer, parameter :: RK4_ID = 1, GBS_ID = 2, RKM_ID = 3, RA15_ID = 4
```

```
  real(DP), parameter :: Z0 = 0, Z1 = 1
  integer ::  id_method = RKM_ID, ll = 10
  real(DP) :: t0 = Z0, t1 = 6.28_DP, t_step = 0.05_DP, eps = 1.0E-9_DP, &
       e = 0.6_DP, eps_closing = 1.0E-5_DP
  !
  ! We adopt the variables with this meaning
  !
  !
  !   y(1:2)   = q(1:2)
  !   y(3:4)   = v(1:2)
  !
  ! being
  !
  !   q(1) = x,  q(2) = y
  !   v(1) = vx, v(2) = vy
  !
  real(DP) :: y0(NEQ) = Z0

  public :: input_data, run_app

contains

  subroutine input_data()
    use get_data, only: get
    real(DP), parameter :: MACHEPS = epsilon(Z1)

    write(*,*) 'Choose the method:'
    write(*,*) '  1 : RK4'
    write(*,*) '  2 : GBS'
    write(*,*) '  3 : RKM'
    write(*,*) '  4 : RA15'
    call get('ID_METHOD =',id_method)
    ! For GBS, RKM or RA15 step, the default initial T_STEP step can be
    ! greather..
    if (id_method /= RK4_ID) t_step = 0.1_DP
    write(*,*)

    call get('T0 = ',t0)
    call get('T1 = ',t1)
    call get('T_STEP (D) = ',t_step)
    if (id_method == GBS_ID .or. id_method == RKM_ID) then
       write(*,*)
       call get('EPS = ',eps)
       if (eps < 1000*MACHEPS) then
          write(*,*) 'EPS TOO SMALL! Exiting... '
          stop
       end if
    end if
    if (id_method == RA15_ID) then
       write(*,*)
       call get('LL = ',ll)
       if (ll > 20) then
          write(*,*) 'LL TOO HIGH! Exiting... '
          stop
       end if
    end if
    write(*,*)

    call get('E = ',e)
    write(*,*)

    if (e <= Z0 .or. e >= Z1) then
       write(*,*) 'E =',e,' NOT VALID FOR AN ELLIPTIC ORBIT! Exiting... '
       stop
    end if

    ! Initial conditions
    y0(1:NEQ) = [ Z1-e, Z0, Z0, sqrt((Z1+e)/(Z1-e)) ]

    !call get('eps_closing = ',eps_closing)
    !write(*,*)
  end subroutine input_data

  subroutine sub(x,y,f)
    real(DP), intent(in) :: x, y(:)
    real(DP), intent(out) :: f(:)
    real(DP), save :: d(NV)
```

```
  ! Filling/initializing with speeds the first half of field f(:)
  f(1:NV) = y(NV+1:NEQ)

  ! Filling with forces/accelerations the second half of field f(:)
  ! d = r(:)/r**3
  d = y(1:NV)
  f(NV+1:NEQ) = -d/norm2(d)**3
end subroutine sub

subroutine force(t,x,v,f)
  real(DP), intent(in) :: t, x(:), v(:)
  real(DP), intent(out) :: f(:)
  real(DP), save :: d(NV)

  ! Filling with forces/accelerations the field f(:)
  ! d = r(:)/r**3
  d = x(1:NV)
  f(1:NV) = -d/norm2(d)**3
end subroutine force

subroutine run_app()
  use math_consts, only: TWO_PI
  use bgi, only: YELLOW, WHITE
  use bgiapp, only: bgiapp_dot
  use ode_integrators, only: rk4step, deqgbs, deqrkm
  use everhart_integrator, only: radau_on, ra15, radau_off
  ! For RK4 w(NEQ,3) would be sufficient...
  ! For RKM w(NEQ,6) would be sufficient...
  ! For GBS we need w(NEQ,36)...
  real(DP) :: t, tz, y(NEQ), w(NEQ,36), t_step0, sgh, t_closing, &
       d(NV), dq, eps_closing_q
  logical :: first

  ! Switching on the Everhart integrator if it has been selected...
  ! LL = 12, NCLASS = -2
  if (id_method == RA15_ID) call radau_on(NV,ll,NCLASS)

  ! Initialization for forward/backward integration
  t_step = sign(abs(t_step),t1-t0)
  sgh = sign(Z1,t_step)

  ! General initialization for integration
  t_step0 = t_step
  t = t0
  y(1:NEQ) = y0(1:NEQ)

  ! Initialization for orbit closure
  t_closing = Z0
  eps_closing_q = eps_closing*eps_closing

  ! Initialization for plotting the central body
  first = .true.

  do while (sgh*(t+t_step0-t1) < 0)

     if (first) then
        first = .false.

        ! The Central Body!!!
        call bgiapp_dot(Z0,Z0,WHITE)
     end if

     call bgiapp_dot(y(1),y(2),YELLOW)

     ! We take an ode integrator step
     if (id_method == RK4_ID) then
        call rk4step(NEQ,t_step,t,y,w,sub)
     else
        t_step = t_step0
        tz = t+t_step
        select case (id_method)
        case (GBS_ID)
           call deqgbs(NEQ,t,tz,y,t_step,eps,w,sub)
        case (RKM_ID)
           call deqrkm(NEQ,t,tz,y,t_step,eps,w,sub)
        case (RA15_ID)
           call ra15(t,tz,y(1:NV),y(NV+1:NEQ),t_step,force)
        end select
```

```
            t = tz
         end if

         ! This is a raw method to determine the error of clusure...
         if (t > t_closing+Z1) then
            d = y(1:NV)-y0(1:NV)
            dq = dot_product(d,d)
            if (dq < eps_closing_q) then
               t_closing = t
               write(*,*)
               write(*,*) 'T_CLOSING = ',t_closing, 'D_CLOSING = ', sqrt(dq), &
                    'EPS_CLOSING = ', eps_closing
            end if
         end if

      end do

      ! Switching OFF the Everhart integrator if it is ON...
      if (id_method == RA15_ID) call radau_off()

      ll = 14
      call radau_on(NV,ll,NCLASS)

      t_step0 = 0.1_DP
      t = Z0
      tz = TWO_PI*8
      y(1:NEQ) = y0(1:NEQ)
      call ra15(t,tz,y(1:NV),y(NV+1:NEQ),t_step0,force)
      d = y(1:NV)-y0(1:NV)
      write(*,*)
      write(*,*)
      write(*,*) 'Method: EVERHART,  LL = ', ll
      write(*,*) 'Error of closure after 8 revolution: ', norm2(d)
      write(*,*)
      call radau_off()

      t_step0 = 0.1_DP
      t = Z0
      tz = TWO_PI*8
      y(1:NEQ) = y0(1:NEQ)
      call deqgbs(NEQ,t,tz,y,t_step0,eps,w,sub)
      d = y(1:NV)-y0(1:NV)
      write(*,*)
      write(*,*)
      write(*,*) 'Method: GBS,  EPS = ', eps
      write(*,*) 'Error of closure after 8 revolution: ', norm2(d)
      write(*,*)

      t_step0 = 0.1_DP
      t = Z0
      tz = TWO_PI*8
      y(1:NEQ) = y0(1:NEQ)
      call deqrkm(NEQ,t,tz,y,t_step0,eps,w,sub)
      d = y(1:NV)-y0(1:NV)
      write(*,*)
      write(*,*)
      write(*,*) 'Method: RKM,  EPS = ', eps
      write(*,*) 'Error of closure after 8 revolution: ', norm2(d)
      write(*,*)
   end subroutine run_app
end module kepler_problem_lib

program kepler_problem
   use kind_consts, only: DP
   use general_routines, only: system_time
   use bgiapp, only: bgiapp_setup, bgiapp_init, bgiapp_close
   use kepler_problem_lib
   implicit none

   real(DP) :: t0, t1

   call input_data()
   call bgiapp_setup(-2.2_DP,1.2_DP,-1.4_DP,1.4_DP,680,560)
   call bgiapp_init("Keplers's Problem")

   write(*,'(A)',advance='NO') 'Please wait, we are working...'

   t0 = system_time()
```

```
  call run_app()
  t1 = system_time()

  write(*,*)
  write(*,'(A,F9.3,A)') 'Completed in ',t1-t0,' seconds!'

  call bgiapp_close()
end program kepler_problem
```