NSEG-5984 Monte Carlo Methods for Particle Transport

Nagendra Krishnamurthy

December 14, 2011

Source code

```
      module data_all

c mfp - mean free path of a molecule in a DILUTE GAS
c eff_dia - effective diameter of the molecule
c num_den - number density
c nic, njc, nkc - number of cells in each direction
c nif, njf, nkf - number of faces in each direction (one plus the
c     number of cells in that direction)
c t_w - temperature of the wall

      implicit none

c General flow properties
      real :: mfp, num_den, n_eff, eff_dia, temp_init, mol_mass,
     1    gas_den, mft, mol_wt, Kn, mpv

c Simulation parameters
      integer :: n_particles, i_dt, ndt, n_realizations,
     1    i_realization, n_total_coll
      real :: dt, time

c Global variables
      real :: boltzmann_k, avagadro_num
      integer :: n_bins
      integer, allocatable :: vel_bin_test1(:,:,:)

c Particle related datastructure
      type particle
        integer :: ijk(3), dom, glb_num
        real :: coord(3), vel(3), coord_nm1(3)
        type (particle), pointer :: next
      end type particle
      type (particle), pointer :: par_list

      type particle_bin
        type (particle), pointer :: par
        type (particle_bin), pointer :: next
      end type particle_bin

c Domain related datastructure
c boundary_type = 1 - periodic boundary
c               = 2 - thermal diffuse wall
c               = 3 - specular wall
      type domain
```

```
      integer :: boundary_type(3,2), ijk(3,2), n_wall_hits(3,2)
      real :: coord(3,2), t_w(3,2), vel_w(3,2,3), vol,
     1    del_vel_w(3,2,3), area(3,2), force(3,2,3),
     1    force_avg(3,2,3)
    end type domain
    type (domain), allocatable :: dom_array(:)
    integer :: n_domains

    type cell
      integer :: n_par
      real :: vol, mass_den, mom_den(3), vel(3),
     1    ener_den, temp, vel_mag, coord(3)
      type (particle_bin), pointer :: par_bin
    end type cell
    type (cell), allocatable :: node_array(:,:,:),
     1    node_avg(:,:,:,:)
    real, allocatable :: x(:,:,:), y(:,:,:), z(:,:,:)
    integer :: nic, njc, nkc, nif, njf, nkf

    contains

    subroutine insert_par_to_bin(bin_list,par_current)

    implicit none

    type (particle), pointer, intent(in) :: par_current
    type (particle_bin), pointer, intent(inout) :: bin_list
    type (particle_bin), pointer :: par_bin_temp, par_bin_current

    par_bin_temp => bin_list%next
    allocate(par_bin_current)
    nullify(par_bin_current%next)
    bin_list%next => par_bin_current
    if (associated(par_bin_temp)) then
    par_bin_current%next => par_bin_temp
    end if
    par_bin_current%par => par_current

    end subroutine insert_par_to_bin


    subroutine par_initialize(par_current)

    implicit none

    type (particle), intent(inout), pointer :: par_current

    par_current%ijk(:) = 0
    par_current%coord(:) = 0.0
    par_current%vel(:) = 0.0
    nullify(par_current%next)

    end subroutine par_initialize


    subroutine par_insert(par_current)

    implicit none

    type (particle), intent(inout), pointer :: par_current
    type (particle), pointer :: par_temp

    if (associated(par_current%next)) then
    par_temp => par_current%next
    allocate(par_current%next)
    par_current%next%next => par_temp
```

```
      else
      allocate(par_current%next)
      end if

      par_current => par_current%next
      call par_initialize(par_current)

      end subroutine par_insert


      subroutine calculate_magnitude(vec,vec_mag)

      implicit none

      real, intent(in) :: vec(:)
      real, intent(out) :: vec_mag

      vec_mag = dsqrt(vec(1)**2.0 + vec(2)**2.0 + vec(3)**2.0)

      end subroutine calculate_magnitude


      end module data_all
```

01.data.f

```
      module random
c module contains three functions
c ran1 returns a uniform random number between 0-1
c spread returns random number between min - max
c normal returns a normal distribution

      contains

c returns random number between 0 - 1
      function uniform_ran()

      implicit none

      real :: uniform_ran, x

      call random_number(x)
      uniform_ran = x

      end function uniform_ran

c returns random number between min - max
      function uniform_spread(min_val,max_val)

      implicit none

      real :: uniform_spread
      real :: min_val, max_val

      uniform_spread = (max_val - min_val) * uniform_ran() + min_val

      end function uniform_spread

c returns random number (integer) between min - max
      function uniform_int_spread(min_val,max_val)

      implicit none

      integer :: uniform_int_spread
      integer, intent(in) :: min_val, max_val
```

```
      uniform_int_spread = dint((max_val - min_val) * uniform_ran()) +
     1    min_val

      end function uniform_int_spread


c returns a normal distribution
      function normal_ran(mean,sigma)

      implicit none

      real :: normal_ran, tmp, fac, gsave, rsq, r1, r2
      real :: mean, sigma
      integer :: flag

      save flag, gsave

      data flag /0/

      if (flag.eq.0) then
      rsq=2.0
      do while ((rsq .ge. 1.0) .or. (rsq .eq. 0.0))
      r1=2.0*uniform_ran()-1.0
      r2=2.0*uniform_ran()-1.0
      rsq=r1*r1+r2*r2
      enddo
      fac=sqrt(-2.0*log(rsq)/rsq)
      gsave=r1*fac
      tmp=r2*fac
      flag=1
      else
      tmp=gsave
      flag=0
      endif
      normal_ran=tmp*sigma+mean
      end function normal_ran

      end module random
```

02.random.f

```
      program dsmc_solver

      use data_all

      implicit none

      integer :: count_bin, count_par, i, j, k
      type (particle), pointer :: par_current
      type (particle_bin), pointer :: par_bin_current

      call open_inout_files

      call read_input

      call setup_indices

      call allocate_arrays

      call generate_grid

      do i_realization = 1, n_realizations

      write(8,*) 'This is realization number:', i_realization
```

```
      n_total_coll = 0
      dom_array(1)%n_wall_hits(:,:) = 0

      call initialize

      call initial

      do i_dt = 1, ndt

      time = i_dt*dt
c     write(6,*) 'This is time-step', i_dt

c Move the particles to new locations
d     write(8,*) 'Entering free_flight'
      call free_flight

c Compute collisions for the time-step
d     write(8,*) 'Entering compute_collisions'
      call compute_collisions

c Compute average to determine macroscopic properties
d     write(8,*) 'Entering compute_averages'
      call compute_averages

c Store average values for averaging over several realizations
d     write(8,*) 'Entering store_averages'
      call store_averages

      if (mod(i_dt,100) .eq. 0) then
      write(8,*) 'This is time-step', i_dt
      write(8,*) 'Total collisions so far:',
     1    n_total_coll
      write(8,*) 'Forces on walls:', dom_array(1)%force(3,:,2)
      end if

      end do
c end do i_dt = 1, ndt

      write(8,*) 'Total collisions in realization:', i_realization,
     1    n_total_coll
      write(8,*) 'Number of wall hits:', dom_array(1)%n_wall_hits(:,:)
      write(8,*) 'Forces on walls in realization:',
     1    dom_array(1)%force(3,:,2)

      end do
c end do i_realization = 1, n_realizations

      write(8,*) 'Average forces on walls:',
     1    dom_array(1)%force_avg(3,:,2)
      write(8,*) 'Pressure on the walls:',
     1    dom_array(1)%force_avg(3,1,3)/dom_array(1)%area(3,1),
     1    dom_array(1)%force_avg(3,2,3)/dom_array(1)%area(3,2)

c Extrapolate to find fluid velocity at the diffuse wall
      open (unit = 101, file = 'fluid_vel_wall.dat', action = 'write',
     1    status = 'replace', position = 'rewind')

      do i_dt = 1, ndt
      i = 1; j = 1
      write(6,*) node_avg(i,j,1,i_dt)%vel(2),
     1    node_avg(i,j,2,i_dt)%vel(2)
      write(101,*) i_dt*dt/mft, ((node_avg(i,j,1,i_dt)%vel(2)
     1    *(node_array(i,j,2)%coord(3)-z(i,j,1))
     1    - (node_avg(i,j,2,i_dt)%vel(2)*(node_array(i,j,1)%coord(3)
     1    - z(i,j,1))))
     1    /(node_array(i,j,2)%coord(3)-node_array(i,j,1)%coord(3)))
```

```
     1    /dom_array(1)%vel_w(3,1,2)
      end do
c end do i_dt = 1, ndt

      close (101)

      call write_tecplot(1)
      call write_tecplot(2)
      call write_tecplot(3)

 501  format(6(1pe12.5,1x))

      end program dsmc_solver
```

dsmc_solver.f

```
      subroutine open_inout_files

      use data_all

      implicit none

      open(unit=8, file='dsmc.out', status='replace',
     1    form='formatted', access='sequential',
     1    action='write')

      end subroutine open_inout_files
```

open_inout_files.f

```
      subroutine read_input

      use data_all

      implicit none

      real :: pi

d     write(8,*) 'read_input in debug mode'

      nic = 1; njc = 1; nkc = 20
      ndt = 1000
      n_realizations = 1000
      n_particles = 10000
      temp_init = 300.0
      n_bins = 20

      n_domains = 1

      allocate(dom_array(n_domains), node_array(nic,njc,nkc))

      dom_array(1)%coord(1,1) = 0.0; dom_array(1)%coord(1,2) = 1.0e-6
      dom_array(1)%coord(2,1) = 0.0; dom_array(1)%coord(2,2) = 1.0e-6
      dom_array(1)%coord(3,1) = 0.0; dom_array(1)%coord(3,2) = 1.0e-6

      dom_array(1)%vol =
     1    (dom_array(1)%coord(1,2)-dom_array(1)%coord(1,1))*
     1    (dom_array(1)%coord(2,2)-dom_array(1)%coord(2,1))*
     1    (dom_array(1)%coord(3,2)-dom_array(1)%coord(3,1))

      dom_array(1)%area(1,1) = (dom_array(1)%coord(2,2)
     1    - dom_array(1)%coord(2,1))*(dom_array(1)%coord(3,2)
     1    - dom_array(1)%coord(3,1))
      dom_array(1)%area(1,2) = dom_array(1)%area(1,1)
      dom_array(1)%area(2,1) = (dom_array(1)%coord(1,2)
```

```
     1    - dom_array(1)%coord(1,1))*(dom_array(1)%coord(3,2)
     1    - dom_array(1)%coord(3,1))
      dom_array(1)%area(2,2) = dom_array(1)%area(2,1)
      dom_array(1)%area(3,1) = (dom_array(1)%coord(2,2)
     1    - dom_array(1)%coord(2,1))*(dom_array(1)%coord(1,2)
     1    - dom_array(1)%coord(1,1))
      dom_array(1)%area(3,2) = dom_array(1)%area(3,1)

      dom_array(1)%ijk(1,1) = 1; dom_array(1)%ijk(1,2) = nic
      dom_array(1)%ijk(2,1) = 1; dom_array(1)%ijk(2,2) = njc
      dom_array(1)%ijk(3,1) = 1; dom_array(1)%ijk(3,2) = nkc

      dom_array(1)%boundary_type(1,1) = 1
      dom_array(1)%boundary_type(1,2) = 1
      dom_array(1)%boundary_type(2,1) = 1
      dom_array(1)%boundary_type(2,2) = 1
      dom_array(1)%boundary_type(3,1) = 2
      dom_array(1)%boundary_type(3,2) = 3

      dom_array(1)%t_w(:,:) = 0.0
      dom_array(1)%t_w(3,1) = 600.0
      dom_array(1)%t_w(3,2) = 300.0

      dom_array(1)%vel_w(:,:,:) = 0.0
      dom_array(1)%vel_w(3,1,2) = 70.0
      dom_array(1)%vel_w(3,2,2) = 0.0

      pi = dacos(-1.0)

c This mean free path formulation is for an equilibrium gas - refer to
c Bird (1976) pp 17.
      boltzmann_k = 1.3806503e-23

c Number density of any gas under standard conditions (1 atm pressure
c and 0 C).
      gas_den = 1.01
      mol_wt = 29.0e-3
      avagadro_num = 6.0221415e+23
      num_den = 2.68699e+25
c     n_eff = gas_den*avagadro_num*dom_array(1)%vol/mol_wt/n_particles
      n_eff = num_den*dom_array(1)%vol/n_particles
      write(8,*) 'Each simulated particle = ', n_eff, 'molecules'
c     Kn = 0.05

c Effective diameter for air is obtained by substituing the measure
c value of the coefficient of viscosity into the theoretical result
c for hard-sphere molecules. NOTE that the mean molecular spacing
c under standard conditions is obtained to 3.3e-9, which just about
c satisfies the dilute gas condition.
      eff_dia = 3.7e-10

      mfp = 1.0/dsqrt(2.0)/pi/eff_dia**2./num_den
      write(8,*) 'Mean free path:', mfp, 'm'
      Kn = mfp/(dom_array(1)%coord(3,2)-dom_array(1)%coord(3,1))
      write(8,*) 'Kn', Kn
c     mfp = Kn*eff_dia

c     mol_mass = gas_den/num_den
      mol_mass = mol_wt/avagadro_num
      mpv = dsqrt(2.*boltzmann_k*temp_init/mol_mass)
c Setting mass as mass of the congregation of molecules
c     write(8,*) 'Warning: mass is mass of a particle, not molecule'
c     mol_mass = n_eff*mol_mass
      mft = mfp/dsqrt(3.0*boltzmann_k/mol_mass)
      dt = 0.01*mft
c     write(8,*) 'Old time-step:', dt
```

```
c     dt = 0.2*(dom_array(1)%coord(3,2)-dom_array(1)%coord(3,1))
c    1    /nkc/mpv
      write(8,*) 'New time-step:', dt

      end subroutine read_input
```

read_input.f

```
      subroutine setup_indices

      use data_all

      implicit none

      nif = nic+1; njf = njc+1; nkf = nkc+1

      end subroutine setup_indices
```

setup_indices.f

```
      subroutine allocate_arrays

      use data_all

      implicit none

      allocate(x(nif,njf,nkf), y(nif,njf,nkf), z(nif,njf,nkf))
      allocate(node_avg(nic,njc,nkc,ndt))

      allocate(vel_bin_test1(n_bins,2,2))

      end subroutine allocate_arrays
```

allocate_arrays.f

```
      subroutine generate_grid

      use data_all

      implicit none

      integer :: i, j, k
      real :: max_del_z, del_x, del_y, del_z

c Set the cell-size in the z-direction to be 75% of the
c mean-free-path.
      max_del_z = 0.75*mfp

      del_z = (dom_array(1)%coord(3,2)-dom_array(1)%coord(3,1))/nkc

      if (del_z .gt. max_del_z) then
      write(8,*) 'Warning: Grid size in z-direction more than 0.75mfp'
      write(8,*) 'Current, max allowable:', del_z, max_del_z
      end if

c Set cell-size in the other two directions. Since one-dimensional
c flow is being considered their values will not have limit based on
c mfp
      del_x = (dom_array(1)%coord(1,2)-dom_array(1)%coord(1,1))/nic
      del_y = (dom_array(1)%coord(2,2)-dom_array(1)%coord(2,1))/njc

c Setting number of faces
      nif = nic + 1; njf = njc + 1; nkf = nkc + 1
```

```
c Compute the vertex x,y,z's
      do k = 1, nkf
      do j = 1, njf
      do i = 1, nif
      x(i,j,k) = dom_array(1)%coord(1,1) + (i-1)*del_x
      y(i,j,k) = dom_array(1)%coord(2,1) + (j-1)*del_y
      z(i,j,k) = dom_array(1)%coord(3,1) + (k-1)*del_z
      end do
c end do i = 1, nif
      end do
c end do j = 1, njf
      end do
c end do k = 1, nkf

c Compute the nodal x,y,z's
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic

      node_array(i,j,k)%coord(1) = 0.125*
     1    ( x(i,j,k) + x(i+1,j,k) + x(i,j+1,k) + x(i+1,j+1,k)
     1    + x(i,j,k+1) + x(i+1,j,k+1) + x(i,j+1,k+1) + x(i+1,j+1,k+1))
      node_array(i,j,k)%coord(2) = 0.125*
     1    ( y(i,j,k) + y(i+1,j,k) + y(i,j+1,k) + y(i+1,j+1,k)
     1    + y(i,j,k+1) + y(i+1,j,k+1) + y(i,j+1,k+1) + y(i+1,j+1,k+1))
      node_array(i,j,k)%coord(3) = 0.125*
     1    ( z(i,j,k) + z(i+1,j,k) + z(i,j+1,k) + z(i+1,j+1,k)
     1    + z(i,j,k+1) + z(i+1,j,k+1) + z(i,j+1,k+1) + z(i+1,j+1,k+1))

      allocate(node_array(i,j,k)%par_bin)
      nullify(node_array(i,j,k)%par_bin%next)

c Assume cartesian grid
      node_array(i,j,k)%vol = (x(i+1,j,k) - x(i,j,k))*
     1    (y(i,j+1,k) - y(i,j,k)) * (z(i,j,k+1) - z(i,j,k))

      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

c Compute the nodal volumes
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic

      node_array(i,j,k)%coord(1) = 0.125*
     1    ( x(i,j,k) + x(i+1,j,k) + x(i,j+1,k) + x(i+1,j+1,k)
     1    + x(i,j,k+1) + x(i+1,j,k+1) + x(i,j+1,k+1) + x(i+1,j+1,k+1))
      node_array(i,j,k)%coord(2) = 0.125*
     1    ( y(i,j,k) + y(i+1,j,k) + y(i,j+1,k) + y(i+1,j+1,k)
     1    + y(i,j,k+1) + y(i+1,j,k+1) + y(i,j+1,k+1) + y(i+1,j+1,k+1))
      node_array(i,j,k)%coord(3) = 0.125*
     1    ( z(i,j,k) + z(i+1,j,k) + z(i,j+1,k) + z(i+1,j+1,k)
     1    + z(i,j,k+1) + z(i+1,j,k+1) + z(i,j+1,k+1) + z(i+1,j+1,k+1))

      allocate(node_array(i,j,k)%par_bin)
      nullify(node_array(i,j,k)%par_bin%next)

      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
```

```
c end do k = 1, nkc

c Traverse through each cell to compute averages
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic
      node_avg(i,j,k,:)%mass_den = 0.0
      node_avg(i,j,k,:)%vel(1) = 0.0
      node_avg(i,j,k,:)%vel(2) = 0.0
      node_avg(i,j,k,:)%vel(3) = 0.0
      node_avg(i,j,k,:)%temp = 0.0
      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc


      end subroutine generate_grid
```

generate_grid.f

```
      subroutine initial

      use data_all

      implicit none

      call par_initial

      end subroutine initial
```

initial.f

```
      subroutine par_initial

      use data_all
      use random

      implicit none

c-----------------------------------------------------------------------
      interface

      subroutine locate_ijk(xyz,ijk)
      integer, intent(inout) :: ijk(:)
      real, intent(in) :: xyz(:)
      end subroutine locate_ijk

      subroutine maxwellian_dist(temp,my_dir,par_current)
      use data_all
      integer, intent(in) :: my_dir
      real, intent(in) :: temp
      type (particle), intent(inout), pointer :: par_current
      end subroutine maxwellian_dist

      end interface
c-----------------------------------------------------------------------

      integer :: i_par, i_dir
      real :: rand_num
      type (particle), pointer :: par_current

d     write(8,*) 'par_initial in debug mode'
```

```
      par_current => par_list

      do i_par = 1, n_particles

      call par_insert(par_current)
c     par_current => par_current%next

      par_current%glb_num = i_par

c Determine particle locations
      do i_dir = 1, 3
      par_current%coord(i_dir) = dom_array(1)%coord(i_dir,1)
     1    + uniform_ran()*(dom_array(1)%coord(i_dir,2)
     1    - dom_array(1)%coord(i_dir,1))
      end do
      par_current%dom = 1

c Locate particles' i,j,k's
      call locate_ijk(par_current%coord,par_current%ijk)

c Determine initial velocities for particles
      do i_dir = 1, 3
      call maxwellian_dist(temp_init,i_dir,par_current)
      end do
c end do i_dir = 1, 3

d     write(8,*) 'Particle:', i_par
d     write(8,*) 'location', par_current%coord(:)
d     write(8,*) 'ijk', par_current%ijk(:)
d     write(8,*) 'velocities', par_current%vel(:)

c end do i_particle = 1, n_particles
      end do

      end subroutine par_initial
```

par_initial.f

```
      subroutine initialize

      use data_all

      implicit none

      integer :: i, j, k
      type (particle), pointer :: par_current

      call random_seed()

      if (.not. associated(par_list)) then
      allocate(par_list)
      else
      do while (associated(par_list%next))
      par_current => par_list%next
      par_list%next => par_list%next%next
      deallocate(par_current)
      end do
      nullify(par_list%next)
      end if
      call par_initialize(par_list)

      end subroutine initialize
```

initialize.f

```
      subroutine free_flight

      use data_all

      implicit none

      integer :: i_par
      type (particle), pointer :: par_current

c-----------------------------------------------------------------------
      interface

      subroutine boundary_interaction(par_current)
      use data_all
      type (particle), intent(inout), pointer :: par_current
      end subroutine boundary_interaction

      subroutine advection(time_step,par_current)
      use data_all
      real, intent(in) :: time_step
      type (particle), pointer, intent(inout) :: par_current
      end subroutine advection

      subroutine locate_ijk(xyz,ijk)
      integer, intent(inout) :: ijk(:)
      real, intent(in) :: xyz(:)
      end subroutine locate_ijk

      end interface
c-----------------------------------------------------------------------

d     write(8,*) 'free_flight in debug mode'

      dom_array(1)%del_vel_w(:,:,:) = 0.0

      par_current => par_list%next

      do while (associated(par_current))

c     write(6,*) 'free flight particle', par_current%glb_num

      par_current%coord_nm1(:) = par_current%coord(:)

      call advection(dt,par_current)

c Compute wall interactions and new particle positions for those which
c collided with a wall
      call boundary_interaction(par_current)

      call locate_ijk(par_current%coord(:),par_current%ijk(:))

      par_current => par_current%next

      end do
c end do while (associated(par_current))

      end subroutine free_flight
```

free_flight.f

```
      subroutine advection(time_step,par_current)

      use data_all

      implicit none
```

```
      real, intent(in) :: time_step
      integer :: i_par, i_dir
      type (particle), pointer, intent(inout) :: par_current

d     write(8,*) 'advection in debug mode'

      do i_dir = 1, 3
      par_current%coord(i_dir) = par_current%coord(i_dir)
     1    + par_current%vel(i_dir)*time_step
      end do
c end do i_dir = 1, 3

      end subroutine advection
```

advection.f

```
      subroutine check_par_out_of_bounds(par_current,par_bounds_flag)

      use data_all

      implicit none

      type (particle), intent(in) :: par_current
      logical, intent(out) :: par_bounds_flag
      integer :: my_dom

d     write(8,*) 'check_par_out_of_bounds in debug mode'

      my_dom = par_current%dom

      if ((par_current%coord(1) .ge. dom_array(my_dom)%coord(1,1))
     1    .and.
     1    (par_current%coord(1) .le. dom_array(my_dom)%coord(1,2))
     1    .and.
     1    (par_current%coord(2) .ge. dom_array(my_dom)%coord(2,1))
     1    .and.
     1    (par_current%coord(2) .le. dom_array(my_dom)%coord(2,2))
     1    .and.
     1    (par_current%coord(3) .ge. dom_array(my_dom)%coord(3,1))
     1    .and.
     1    (par_current%coord(3) .le. dom_array(my_dom)%coord(3,2))
     1    ) then
      par_bounds_flag = .true.
      else
      par_bounds_flag = .false.
      end if

      end subroutine check_par_out_of_bounds
```

check_par_out_of_bounds.f

```
      subroutine boundary_interaction(par_current)

      use data_all
      use random

      implicit none

      type (particle), intent(inout), pointer :: par_current
      integer :: apply_bc_flag, i_dir, i_dir_flag, my_dom, my_dir,
     1    my_dir_flag, min_time_loc(2), alt_dir_flag, vel_bin_idx
      real :: time_to_bound(3,2), coord_prev(3), dt_rem, distance,
     1    distance_prev, vel_max, vel_mag, vel_old(3)
      logical :: par_bounds_flag
```

```
c-----------------------------------------------------------------------
      interface

      subroutine advection(time_step,par_current)
      use data_all
      real, intent(in) :: time_step
      type (particle), pointer, intent(inout) :: par_current
      end subroutine advection

      end interface
c-----------------------------------------------------------------------

d     write(8,*) 'boundary_interaction in debug mode'

      vel_max = 3.0*dsqrt(2.0*boltzmann_k*temp_init/mol_mass)
      my_dom = par_current%dom
      coord_prev(:) = par_current%coord_nm1(:)
      dt_rem = dt

      call check_par_out_of_bounds(par_current,par_bounds_flag)

c Apply boundary conditions till all the boundaries have been that
c have been passed are processed
      do while (.not. par_bounds_flag)

      apply_bc_flag = 0
      time_to_bound = 1.0E+10

      do i_dir = 1, 3

      do i_dir_flag = 1, 2

      distance = par_current%coord(i_dir)
     1    - dom_array(my_dom)%coord(i_dir,i_dir_flag)
      distance_prev = coord_prev(i_dir)
     1    - dom_array(my_dom)%coord(i_dir,i_dir_flag)
c If the distance to the boundary before and after the advection
c changed signs, that means the particle crossed that boundary.
      if (distance*distance_prev .lt. 0.0) then
      time_to_bound(i_dir,i_dir_flag) = abs(distance_prev
     1    /par_current%vel(i_dir))
      apply_bc_flag = 1
      end if
c end if (distance*distance_nm1 .lt. 0.0) then

      end do
c end do i_dir_flag = 1, 2

      end do
c end do i_dir = 1, 3

      if (apply_bc_flag .eq. 1) then

c Locate the boundary which will be approached first
      min_time_loc = minloc(time_to_bound)
      my_dir = min_time_loc(1)
      my_dir_flag = min_time_loc(2)

c     print*, 'boundary', my_dir, my_dir_flag

      par_current%coord(:) = coord_prev(:)

      call advection(time_to_bound(my_dir,my_dir_flag),par_current)
      par_current%coord(my_dir) =
     1    dom_array(my_dom)%coord(my_dir,my_dir_flag)
```

```
      dt_rem = dt_rem - time_to_bound(my_dir,my_dir_flag)

      if (dom_array(my_dom)%boundary_type(my_dir,my_dir_flag)
     1    .eq. 1) then
c Setting up periodic boundary condition

      if (my_dir_flag .eq. 1) then
      alt_dir_flag = 2
      else
      alt_dir_flag = 1
      end if

      par_current%coord(my_dir) =
     1    dom_array(my_dom)%coord(my_dir,alt_dir_flag)

      else if (dom_array(my_dom)%boundary_type(my_dir,my_dir_flag)
     1    .eq. 2) then
c Setting up diffuse thermal wall conditions

      dom_array(1)%n_wall_hits(my_dir,my_dir_flag) =
     1    dom_array(1)%n_wall_hits(my_dir,my_dir_flag) + 1
      vel_old(:) = par_current%vel(:)

      do i_dir = 1, 3

      if (i_dir .eq. my_dir) then
      par_current%vel(i_dir) = dsqrt(
     1    -2.0*boltzmann_k
     1    *dom_array(my_dom)%t_w(my_dir,my_dir_flag)
     1    *dlog(1.0-uniform_ran())/mol_mass)
      else
      par_current%vel(i_dir) = dsqrt(
     1    boltzmann_k*dom_array(my_dom)%t_w(my_dir,my_dir_flag)
     1    /mol_mass)*normal_ran(0.0,1.0)
     1    + dom_array(my_dom)%vel_w(my_dir,my_dir_flag,i_dir)
      end if

      end do
c end do i_dir = 1, 3

c Flip direction of the velocity component in the wall normal
c direction if the wall is the higher index (imax, jmax or kmax)
      if (my_dir_flag .eq. 2) par_current%vel(my_dir) =
     1    -par_current%vel(my_dir)

      do i_dir = 1, 3
      dom_array(1)%del_vel_w(my_dir,my_dir_flag,i_dir) =
     1    dom_array(1)%del_vel_w(my_dir,my_dir_flag,i_dir) +
     1    (par_current%vel(i_dir) - vel_old(i_dir))
      end do
c end do i_dir = 1, 3

      else if (dom_array(my_dom)%boundary_type(my_dir,my_dir_flag)
     1    .eq. 3) then
c Setting up specular wall conditions

      par_current%vel(my_dir) = -par_current%vel(my_dir)

      end if
c end if (dom_array(my_dom)%boundary_type(my_dir) .eq. 1) then

c Mark the boundary as processed
      time_to_bound(:,:) = 1.0E+10

      else
```

```
      end if
c end if (apply_bc_flag .eq. 0) then

      coord_prev(:) = par_current%coord(:)
      call advection(dt_rem,par_current)
      call check_par_out_of_bounds(par_current,par_bounds_flag)

      end do
c end do while (.not. par_bounds_flag)

      end subroutine boundary_interaction
```

boundary_interaction.f

```
      subroutine collide_particles(par_1,par_2)

      use data_all
      use random

      implicit none

      real :: vel_cm(3), phi, sin_phi, cos_phi, cos_theta, sin_theta,
     1    pi, rel_vel_mag, rel_vel(3)
      type (particle), pointer, intent(inout) :: par_1, par_2

      pi = dacos(-1.0)

c Hard-sphere model

c Center of mass velocity
      vel_cm(:) = 0.5*(par_1%vel(:) + par_2%vel(:))

c Estimation of azimuthal angle (phi)
      phi = 2.0*pi*uniform_ran()
      sin_phi = dsin(phi)
      cos_phi = dcos(phi)
      cos_theta = 2.0*uniform_ran()-1.0
      sin_theta = dsqrt(1.0 - cos_theta**2.0)

c Relative velocity after collision
      call calculate_magnitude((par_1%vel(:)-par_2%vel(:)),
     1    rel_vel_mag)
      rel_vel(1) = rel_vel_mag*(sin_theta*cos_phi)
      rel_vel(2) = rel_vel_mag*(sin_theta*sin_phi)
      rel_vel(3) = rel_vel_mag*(cos_theta)

c Post-collision velocities
      par_1%vel(:) = vel_cm(:) + 0.5*rel_vel(:)
      par_2%vel(:) = vel_cm(:) - 0.5*rel_vel(:)

      end subroutine collide_particles
```

collide_particles.f

```
      subroutine bin_particles

      use data_all

      implicit none

      integer :: i, j, k, min_particles_bin
      type (particle), pointer :: par_current
      type (particle_bin), pointer :: par_bin_current
```

```
      par_current => par_list%next

      do while (associated(par_current))

      i = par_current%ijk(1)
      j = par_current%ijk(2)
      k = par_current%ijk(3)

      par_bin_current => node_array(i,j,k)%par_bin

      call insert_par_to_bin(par_bin_current,par_current)
      node_array(i,j,k)%n_par = node_array(i,j,k)%n_par + 1

      par_current => par_current%next

      end do
c end do while (associated(par_current))

c Check if each cell has at least 30 cells, if not issue a warning.
      min_particles_bin = minval(node_array(:,:,:)%n_par)
      if (min_particles_bin .lt. 30) write(8,*) 'Warning:', i, j, k,
     1    'cell has less than 30 particles'

      end subroutine bin_particles
```

bin_particles.f

```
      subroutine compute_collisions

      use data_all
      use random

      implicit none

      integer :: i, j, k, max_num_coll_success, num_coll_success,
     1    pair_num, par_num_1, par_num_2, par_num_temp, i_par
      real :: max_rel_vel, sum_rel_vel_mag, rel_vel_mag,
     1    avg_rel_vel_mag, pi
      type (particle), pointer :: par_current, par_1, par_2
      type (particle_bin), pointer :: par_bin_current

c-----------------------------------------------------------------------
      interface
      subroutine collide_particles(par_1,par_2)
      use data_all
      type (particle), pointer, intent(inout) :: par_1, par_2
      end subroutine collide_particles
      end interface
c-----------------------------------------------------------------------

d     write(8,*) 'compute_collisions in debug mode'

      pi = dacos(-1.0)

      call bin_particles

c Go through each cell and perform collisions
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic

c Compute an estimate for maximum relative speed. This is a
c trade-off between efficiency of the collision pair selection scheme
c and computing the actual maximum relative speed for all potential
c collision partners in each cell.
```

```
      call compute_max_rel_vel(i,j,k,max_rel_vel)

c Initial guess for max_num_coll_success
      max_num_coll_success = 100
      num_coll_success = 0
      pair_num = 0
      sum_rel_vel_mag = 0.0

      do while (num_coll_success .le. max_num_coll_success)

      pair_num = pair_num + 1

c Select one out of the total number of particles present in the cell
      par_num_1 = uniform_int_spread(1,node_array(i,j,k)%n_par)

c Select the second particle - from the total number of particles
c remaining (n_particles_bin - 1). Later adjust the number to reflect
c the actual number on the list.
      par_num_2 = uniform_int_spread(1,node_array(i,j,k)%n_par-1)
      if (par_num_2 .ge. par_num_1) then
      par_num_2 = par_num_2 + 1
      else
      par_num_temp = par_num_1
      par_num_1 = par_num_2
      par_num_2 = par_num_temp
      end if

      par_bin_current => node_array(i,j,k)%par_bin
      do i_par = 1, par_num_1
      par_bin_current => par_bin_current%next
      end do
c end do i_par = 1, par_num_1
      par_1 => par_bin_current%par
      do i_par = par_num_1+1, par_num_2
      par_bin_current => par_bin_current%next
      end do
c end do i_par = par_num_1+1, par_num_2
      par_2 => par_bin_current%par

      call calculate_magnitude((par_1%vel(:)-par_2%vel(:)),
     1    rel_vel_mag)

c Use rejection technique to determine if the collision will be
c successful
      if ((rel_vel_mag/max_rel_vel) .gt. uniform_ran()) then
      num_coll_success = num_coll_success + 1
      call collide_particles(par_1,par_2)
      end if

c Updating of average relative speed for the cell (is this correct?)
      sum_rel_vel_mag = sum_rel_vel_mag + rel_vel_mag
      avg_rel_vel_mag = sum_rel_vel_mag/pair_num

c Compute maximum number of collisions allowed in the cell
      max_num_coll_success = (node_array(i,j,k)%n_par**2.0
     1    *pi*eff_dia**2.0*max_rel_vel*n_eff*dt
     1    /2.0/node_array(i,j,k)%vol)
c    1    *(rel_vel_mag/max_rel_vel)
     1    *(avg_rel_vel_mag/max_rel_vel)

c     write(8,*) 'max coll value', max_num_coll_success,
c    1    avg_rel_vel_mag

      end do
c end do while (num_coll_success .le. max_num_coll_success)
```

```
c     write(8,*) 'max, allowed', max_num_coll_success,
c    1    num_coll_success
      n_total_coll = n_total_coll + num_coll_success

      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

      end subroutine compute_collisions
```

compute_collisions.f

```
      subroutine compute_max_rel_vel(my_i,my_j,my_k,max_rel_vel)

c Compute the maximum of each velocity components within each cell.
c This is done by traversing through each particle present in the cell
c and finding the maximum absolute value of each of the three velocity
c components

      use data_all

      implicit none

      integer, intent(in) :: my_i, my_j, my_k
      real, intent (out) :: max_rel_vel

      integer :: i_dir, count_bin
      real :: max_vel(3)
      type (particle), pointer :: par_current
      type (particle_bin), pointer :: par_bin_current

d     write(8,*) 'compute_max_rel_vel in debug mode'

      par_bin_current => node_array(my_i,my_j,my_k)%par_bin%next
      max_vel(:) = 0.0

      do while (associated(par_bin_current))

      par_current => par_bin_current%par

      do i_dir = 1, 3
      max_vel(i_dir) = dmax1(max_vel(i_dir),
     1    dabs(par_current%vel(i_dir)))
      end do
c end do i_dir = 1, 3

      par_bin_current => par_bin_current%next

      end do
c end do while (associated(par_current))

c Maximum possible relative velocity is obtained when two particles
c approach each other, with velocity components that are maximum for
c the cell and are opposite of each other. Hence, each of the
c component of the max relative velocity vector will be twice the max
c velocity vector.
      call calculate_magnitude(2.0*max_vel,max_rel_vel)

      end subroutine compute_max_rel_vel
```

compute_max_rel_vel.f

```
      subroutine delete_bin_particles

      use data_all

      implicit none

      integer :: i, j, k, count_bin
      type (particle_bin), pointer :: par_bin_current, par_bin_temp

c Go through each cell and delete binned particles
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic
      par_bin_current => node_array(i,j,k)%par_bin%next
      nullify(node_array(i,j,k)%par_bin%next)
      node_array(i,j,k)%n_par = 0
      do while (associated(par_bin_current))
      par_bin_temp => par_bin_current%next
      deallocate(par_bin_current)
      par_bin_current => par_bin_temp
      end do
c end do while (associated(par_bin_current))
      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

      end subroutine delete_bin_particles
```

delete_bin_particles.f

```
      subroutine compute_averages

      use data_all

      implicit none

      integer :: i, j, k, i_dir, i_dir_flag, my_dir
      real :: sum_mass_den, sum_mom_den(3), sum_ener_den, vel_mag
      type (particle), pointer :: par_current
      type (particle_bin), pointer :: par_bin_current

c Traverse through each cell to compute averages
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic

      sum_mass_den = 0.0
      sum_mom_den(:) = 0.0
      sum_ener_den = 0.0

      par_bin_current => node_array(i,j,k)%par_bin%next
      do while (associated(par_bin_current))

      par_current => par_bin_current%par

      sum_mass_den = sum_mass_den + mol_mass
c     sum_mass_den = sum_mass_den + mol_mass*n_eff
      sum_mom_den(:) = sum_mom_den(:)
     1    + mol_mass*par_current%vel(:)
c    1    + mol_mass*n_eff*par_current%vel(:)
      call calculate_magnitude(par_current%vel,vel_mag)
      sum_ener_den = sum_ener_den
```

```
     1    + 0.5*mol_mass*vel_mag**2.0
c    1    + 0.5*mol_mass*n_eff*vel_mag**2.0

      par_bin_current => par_bin_current%next

      end do
c end do while (associated(par_bin_current))

      node_array(i,j,k)%mass_den = sum_mass_den/node_array(i,j,k)%vol

      node_array(i,j,k)%mom_den(:) =
     1    sum_mom_den(:)/node_array(i,j,k)%vol
      node_array(i,j,k)%vel(:) =
     1    node_array(i,j,k)%mom_den(:)/node_array(i,j,k)%mass_den
      call calculate_magnitude(node_array(i,j,k)%vel,
     1    node_array(i,j,k)%vel_mag)

      node_array(i,j,k)%ener_den = sum_ener_den/node_array(i,j,k)%vol
      node_array(i,j,k)%temp = 2.0*mol_mass/3.0/boltzmann_k
     1    *((node_array(i,j,k)%ener_den/node_array(i,j,k)%mass_den)
     1    - (0.5*(node_array(i,j,k)%vel_mag**2.0)))

      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

c Compute the force on every wall
      do i_dir = 1, 3
      do i_dir_flag = 1, 2
      dom_array(1)%force(i_dir,i_dir_flag,:) =
     1    (dom_array(1)%del_vel_w(i_dir,i_dir_flag,:)*
     1    n_eff*mol_mass)
     1    /((dt*i_dt)*dom_array(1)%area(i_dir,i_dir_flag))
      end do
c end do i_dir_flag = 1, 2
      end do
c end do i_dir = 1, 3
      dom_array(1)%force_avg(:,:,:) = (dom_array(1)%force_avg(:,:,:)
     1    *(i_realization-1) + dom_array(1)%force(:,:,:))
     1    /i_realization

      call delete_bin_particles

      end subroutine compute_averages
```

compute_averages.f

```
      subroutine store_averages

      use data_all

      implicit none

      integer :: i, j, k, i_dir

c Traverse through each cell to compute averages
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic
      node_avg(i,j,k,i_dt)%mass_den =
     1    (node_avg(i,j,k,i_dt)%mass_den*(i_realization-1)
     1    + node_array(i,j,k)%mass_den)/i_realization
```

```
      do i_dir = 1, 3
      node_avg(i,j,k,i_dt)%vel(i_dir) =
     1    (node_avg(i,j,k,i_dt)%vel(i_dir)*(i_realization-1)
     1    + node_array(i,j,k)%vel(i_dir))/i_realization
      end do
c end do i_dir = 1, 3
      node_avg(i,j,k,i_dt)%temp =
     1    (node_avg(i,j,k,i_dt)%temp*(i_realization-1)
     1    + node_array(i,j,k)%temp)/i_realization
      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

      end subroutine store_averages
```

store_averages.f

```
      subroutine write_tecplot(i_plot)

c i_plot = 1 - velocity distribution
c        = 2 - instantaneous solution
c        = 3 - averaged solution

      use data_all

      implicit none

      integer, intent(in) :: i_plot
      integer :: i, j, k, idt, vel_bin_idx, i_bin, vel_bin(n_bins)
      real :: vel_mag, vel_max
      type (particle), pointer :: par_current

      select case (i_plot)

      case (1)

      vel_bin(:) = 0
      vel_max = 3.0*dsqrt(2.0*boltzmann_k*temp_init/mol_mass)

      par_current => par_list%next

      do while (associated(par_current))

      call calculate_magnitude(par_current%vel,vel_mag)
      vel_bin_idx = dint(vel_mag/vel_max*n_bins) + 1
      if (vel_bin_idx .gt. n_bins) vel_bin_idx = n_bins
      vel_bin(vel_bin_idx) = vel_bin(vel_bin_idx) + 1
      par_current => par_current%next
      end do
c end do while (associated(par_current))

      open (unit = 101, file = 'speed_dist.dat', action = 'write',
     1    status = 'replace', position = 'rewind')

      do i_bin = 1, n_bins
      write(101,504) (i_bin-0.5)/n_bins*vel_max, vel_bin(i_bin)
      end do
c end do i_bin = 1, n_bins

      close (101)

      open (unit = 102, file = 'speed_dist_specular_wall.dat',
```

```
     1    action = 'write', status = 'replace', position = 'rewind')

      do i_bin = 1, n_bins
      write(102,505) (i_bin-0.5)/n_bins*vel_max,
     1    vel_bin_test1(i_bin,1,2), vel_bin_test1(i_bin,2,2)
      end do
c end do i_bin = 1, n_bins

      close (102)

      open (unit = 103, file = 'speed_dist_diffuse_wall.dat',
     1    action = 'write', status = 'replace', position = 'rewind')

      do i_bin = 1, n_bins
      write(103,505) (i_bin-0.5)/n_bins*vel_max,
     1    vel_bin_test1(i_bin,1,1), vel_bin_test1(i_bin,2,1)
      end do
c end do i_bin = 1, n_bins

      close (103)


      case (2)

      open (unit = 102, file = 'solution.dat', action = 'write',
     1    status = 'replace', position = 'rewind')

      write(102,501) 'Instataneous solution', time
      write(102,502) 1, nic, njc, nkc

      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic
      write(102,503) node_array(i,j,k)%coord(:),
     1    node_array(i,j,k)%mass_den,
     1    node_array(i,j,k)%vel(:), node_array(i,j,k)%temp
      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc

      close (102)

      case (3)

      open (unit = 103, file = 'solution_avg.dat', action = 'write',
     1    status = 'replace', position = 'rewind')

      write(103,501) 'Average solution', time
      write(103,502) 1, nic, njc, nkc

      idt = ndt
      do k = 1, nkc
      do j = 1, njc
      do i = 1, nic
      write(103,503) node_array(i,j,k)%coord(:),
     1    node_avg(i,j,k,idt)%mass_den,
     1    node_avg(i,j,k,idt)%vel(:), node_avg(i,j,k,idt)%temp
      end do
c end do i = 1, nic
      end do
c end do j = 1, njc
      end do
c end do k = 1, nkc
```

```
      close (103)

      end select

 501  format ('TITLE = "',A70, 1pe12.5, '"',/, 'VARIABLES = "x"',/,
     1    '"y"',/, '"z"',/,'"rho"',/,'"u"',/,'"v"',/,'"w"',/,'"t"')
 502  format (1x, 'ZONE T = "zone', I3, '",', 'I = ', I3, ',',
     1    'J = ', I3, ',', 'K = ', I3, ',', 'ZONETYPE = Ordered, ',/,
     1    'DATAPACKING = POINT')
 503  format (1x, 8(1pe12.5,2x))
 504  format (1x, 1(1pe12.5,2x), I9)
 505  format (1x, 1(1pe12.5,2x), 2(I9))

      end subroutine write_tecplot
```

write_tecplot.f

```
      subroutine maxwellian_dist(temp,my_dir,par_current)

      use data_all
      use random

      implicit none

      integer, intent(in) :: my_dir
      real, intent(in) :: temp
      type (particle), intent(inout), pointer :: par_current

      par_current%vel(my_dir) = dsqrt(boltzmann_k*temp
     1    /mol_mass)*normal_ran(0.0,1.0)

      end subroutine maxwellian_dist
```

maxwellian_dist.f

```
      subroutine locate_ijk(xyz,ijk)

      use data_all

      implicit none

      integer :: i_dir, max_index
      integer, intent(inout) :: ijk(:)
      real, intent(in) :: xyz(:)

      do i_dir = 1, 3
      max_index = dom_array(1)%ijk(i_dir,2)
     1    - dom_array(1)%ijk(i_dir,1) + 1
      ijk(i_dir) = dint(
     1    ((xyz(i_dir) - dom_array(1)%coord(i_dir,1))
     1    /(dom_array(1)%coord(i_dir,2) - dom_array(1)%coord(i_dir,1))
     1    *max_index)) + 1
c     if (ijk(i_dir) .gt. dom_array(1)%ijk(i_dir,2))
c    1    ijk(i_dir) = dom_array(1)%ijk(i_dir,2)
      end do

      end subroutine locate_ijk
```

locate_ijk.f