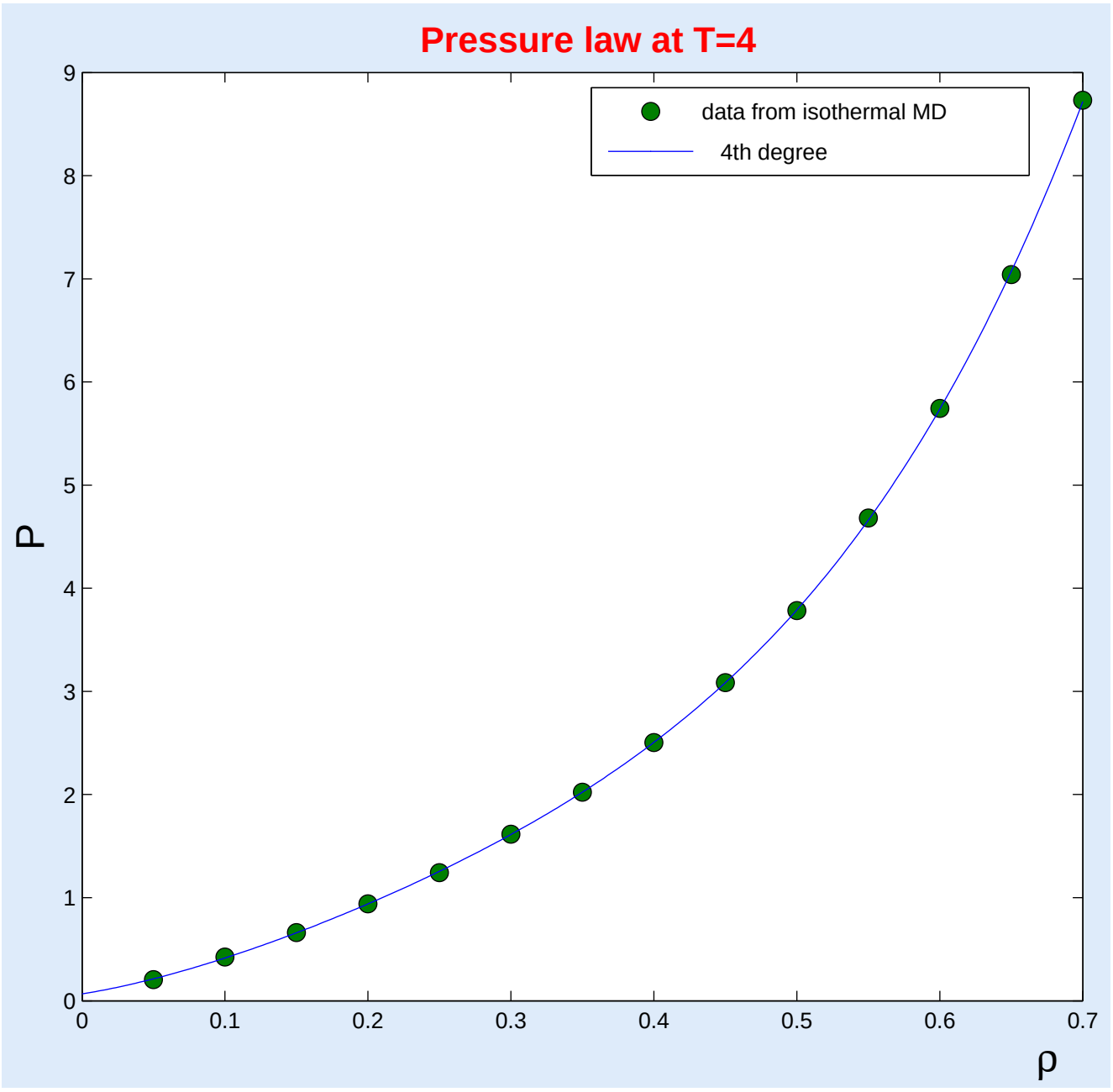
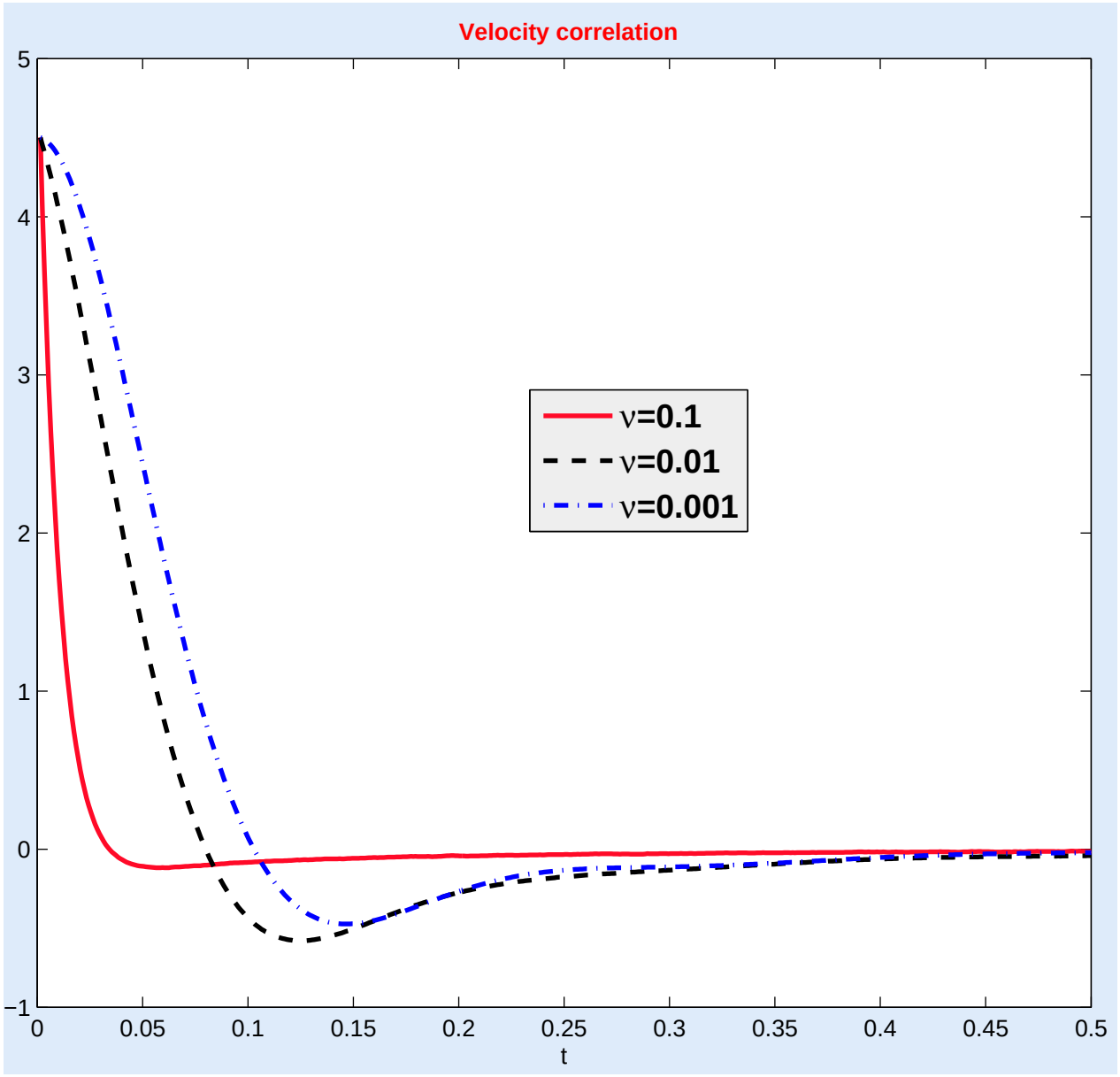
```
module cell

  public

  integer, parameter :: mx, my, mz
  integer, parameter :: ncell= mx*my*mz, mapsize= 13*ncell
  integer :: list(nmax), head(ncell), map(mapsize)

contains

  function icell(i, j, k) result(m)

    !:: number the cells

    implicit none

    integer :: i, j, k, m

    m= 1 + mod(i - 1 + mx, mx) &
         + mod(j - 1 + my, my) * mx &
         + mod(k - 1 + mz, mz) * my * mx

    return

  end function icell


  subroutine maps

    ! set up a list of neighboring cells

    implicit none

    integer :: i, j, k, imap

    do k=1, mz
       do j=1, my
          do i=1, mx

             imap= ( icell( i, j, k) -1 )*13

             !:: link to the following cells
             map( imap + 1  ) = icell( i + 1, j    , k     )
             map( imap + 2  ) = icell( i + 1, j + 1, k     )
             map( imap + 3  ) = icell( i    , j + 1, k     )
             map( imap + 4  ) = icell( i - 1, j + 1, k     )
             map( imap + 5  ) = icell( i + 1, j    , k - 1 )
             map( imap + 6  ) = icell( i + 1, j + 1, k - 1 )
             map( imap + 7  ) = icell( i    , j + 1, k - 1 )
             map( imap + 8  ) = icell( i - 1, j + 1, k - 1 )
             map( imap + 9  ) = icell( i + 1, j    , k + 1 )
             map( imap + 10 ) = icell( i + 1, j + 1, k + 1 )
             map( imap + 11 ) = icell( i    , j + 1, k + 1 )
             map( imap + 12 ) = icell( i - 1, j + 1, k + 1 )
             map( imap + 13 ) = icell( i    , j    , k + 1 )

          end do
       end do
    end do
```

```
  return

end subroutine maps


subroutine links

  implicit none

  real(8) :: csize
  integer :: m, i

  csize= bsize(1)/dble(mx)

  head = 0

  do i=1, nmax
     m= 1 + int( (r(i,1) + 0.5) * csize ) &
          + int( (r(i,2) + 0.5) * csize ) * mx &
          + int( (r(i,3) + 0.5) * csize ) * mx * my

     list(i)= head(m)
     head(m)= i
  end do

  return

end subroutine links

subroutine comp_force

  implicit none

  real(8) :: rij(3), fij(3), pij

  integer :: i, j, k, m, n, j0


  do m=1, mcell

     i = head(m)

     do while (i ≠ 0)

        !--- Part I: all the particles in the current cell

        j = list(i)

        do while (j ≠ 0)

           rij = x(i,:)-x(j,:)

           call LennJo(rij, pij, fij)

           f(i,:)= f(i,:) + fij
           f(j,:)= f(j,:) - fij

           p(i)= p(i) + pij
```

```
            p(j)= p(j) + pij

            j=list(j)
         end do

         !--- Part II: particles in neighboring cells
         j0= 13 * (m - 1)

         do n=1, 13

            k= map(j0 + n)

            j= head(k)

            do while (j ≠ 0)

               rij=x(i,:)-x(j,:)

               call LennJo(rij, pij, fij)

               f(i,:)= f(i,:) + fij
               f(j,:)= f(j,:) - fij

               p(i)= p(i) + pij
               p(j)= p(j) + pij

               j=list(j)
            end do

         end do
         i= list(i)

      end do

  end subroutine comp_force

  subroutine LennJo(rij, pij, fij)

    implicit none

    real(8), intent(in) :: rij(3)
    real(8), intent(out):: fij(3), pij

    real(8) :: r2i, r6i, r2

    !:: periodic boundary condition
    r2= 0d0
    do k=1, nd
       dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
       r2   = r2 + dr(k)*dr(k)
    end do

    if(r2 > rcut2) then

       fij = 0d0
       pij = 0d0
       return

    else
```

```
        r2i= 1D0/r2
        r6i= r2i*r2i*r2i

        fij = r2i*r6i*(r6i-.5D0)*dr
        pij = r6i*(r6i - 1d0) - pcut

        return

     end if

  end subroutine LennJo

end module cell
```


Pressure law at T=4
data from isothermal MD
4th degree
P
ρ
0
1
2
3
4
5
6
7
8
9
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7


Velocity correlation
5
4
3
2
1
0
−1
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
t
ν=0.1
ν=0.01
ν=0.001

```
program main

  !::  MOLECULAR DYNAMICS SIMULATION IN A TWO DIMENSIONAL LENNARD-JONES SYSTEM
::

  implicit none

  !:: size of the system
  integer, parameter :: nx=32, ny=32, na=2, nmax= nx*ny*na, nd=2

  !:: time period
  integer, parameter :: nstep=60000

  !:: sample frequency and number of samples
  integer, parameter :: nsf= 20, nsamp= nstep/nsf

  !:: step size for the integrator
  real(8), parameter :: delt= 3D-4

  !:: particle position, velocity, force, potential, etc.
  real(8) :: x(nmax,nd), v(nmax,nd), f(nmax,nd), p(nmax), xs(nmax, nd)
  real(8) :: bsize(nd)

  !:: density
  real(8), parameter :: rho= 0.8d0, vol= dble(nmax)/rho

  !:: Verlet list
  integer, parameter :: ncf=10 ! check the list every ncf steps
  real(8), parameter :: rcut = 2.5d0, rlist=3.2d0
  real(8), parameter :: rcut2= rcut*rcut
  real(8), parameter :: rlst2= rlist*rlist
  integer, parameter :: maxnb= 40
  integer :: point(nmax), list(nmax*maxnb)

  !:: adjust the potential energy
  real(8), parameter :: pcut= (rcut**(-12.0) - rcut**(-6.0))

  !:: Quantities being sampled
  real(8), dimension(nsamp) :: tmpr, pr, ekin, epot

  integer :: i, n

  !:: initialize the system ...
  call INIT_POS

  call INIT_VEL

  !:: time integration
  do n=1, nstep

     call INTEGRATE

     call SAMP_DATA

  end do

  call OUTPUT

contains
```

```
subroutine init_pos

  !:: initial the postion the the particle on a triangular lattice

  implicit none

  real(8) :: a0, dx, dy, dr(2), r2
  integer :: i, j, m, n

  a0= dsqrt( 2d0/dsqrt(3d0)/rho )

  dx= a0; dy= a0*dsqrt(3d0)

  bsize= (/ dble(nx)*dx, dble(ny)*dy /)

  x(1, :)= (/0d0, 0d0/)
  x(2, :)= (/dx/2d0, dy/2d0/)

  n=0
  do i=1, nx
     do j=1, ny
        do m=1, 2
           x(n+m, :)= x(m, :) + (/dble(i-1)*dx, dble(j-1)*dy/)
        end do
        n= n + 2
     end do
  end do

  call comp_force

  return

end subroutine init_pos

subroutine init_vel

  !:: initialize the velocity

  use ran_mod

  integer :: isd(4), i, k

  isd= (/3, 5, 7 ,9/)

  do i=1, nmax
     do k=1, nd
        v(i, k)= dlarnd(3, isd)*.4d0
     end do
  end do

  !:: shift the velocity so that the mean is zero
  do k=1, nd
     v(:,k)=v(:,k) - sum(v(:,k))/dble(nmax)
  end do

  return

end subroutine init_vel
```

```
subroutine comp_force

  implicit none

  integer, save :: nf= 0
  logical :: update

  real(8) :: dr(2), r2, r2i, r6i, ff(2), pp
  integer :: i, j, k, jbeg, jend, jnb, nlist

  nf= nf + 1
  if(mod(nf, ncf) ≡ 0) call check(update)

  !:: update the Verlet's list
  if (update) then

     call savex

     update= .false.

     !:: setting up the Verlet's list
     nlist=0
     do i = 1, nmax-1
        point(i)=nlist+1

        do j = i+1, nmax
           dr=x(i,:)-x(j,:)

           !:: periodic boundary condition
           r2= 0d0
           do k=1, nd
              dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
              r2   = r2 + dr(k)*dr(k)
           end do

           if (r2 < rlst2) then
              !:: add the particle to the list
              nlist=nlist+1
              list(nlist)=j
           end if
        end do
     end do
     point(nmax)=nlist+1
  end if

  !:: calculate the force and energy
  f= 0d0; p= 0d0
  do i=1, nmax-1
     jbeg = point(i)
     jend = point(i+1)-1
     do jnb= jbeg, jend
        j  = list(jnb)
        dr=x(i,:)-x(j,:)

        !:: periodic boundary condition
        r2= 0d0
        do k=1, nd
```

```
            dr(k)= dr(k) − dnint(dr(k)/bsize(k))*bsize(k)
            r2   = r2 + dr(k)*dr(k)
         end do

         if(r2 ≤ rcut2) then
            r2i= 1D0/r2
            r6i= r2i**3
            ff = r2i*r6i*(r6i−.5D0)*dr

            f(i,:)= f(i,:) + ff
            f(j,:)= f(j,:) − ff

            pp  = r6i*(r6i − 1d0) − pcut
            p(i)= p(i) + pp
            p(j)= p(j) + pp

         end if
      end do
   end do

   f= f*48d0
   p= p*4D0

   return

end subroutine comp_force

subroutine savex

   !:: save the array x to xs

   xs= x

end subroutine savex

subroutine check(update)

   !:: check the Verlet's list

   implicit none

   logical, intent(out) :: update

   real(8) :: dispmx, dr(2), r2
   integer :: i, k

   dispmx = 0D0
   do i = 1, nmax
      dr= x(i, :) − xs(i, :)

      r2= 0d0
      do k=1, nd
         dr(k)= dr(k) − dnint(dr(k)/bsize(k))*bsize(k)
         r2 = r2 + dr(k)*dr(k)
      end do

      dispmx = Dmax1(r2, dispmx)
   end do
```

```
    dispmx = 2D0 * Dsqrt (dispmx)
    update = (dispmx > (rlist -rcut))

    return

  end subroutine check

  subroutine integrate

    v= v + f*delt/2d0
    x= x + v*delt

    call adj_pos

    call comp_force

    v= v + f*delt/2d0

    return

  end subroutine integrate

  subroutine samp_data

    implicit none

    integer, save :: ntime = 0
    integer :: nc

!    call samp_vacf
    call samp_rdf

    ntime= ntime + 1
    if(mod(ntime, nsf) ≠ 0) then
       return
    else
       nc= ntime/nsf
       call samp_engr (nc)
       call samp_tmpr (nc)
       call samp_prssr(nc)
    end if

    return

  end subroutine samp_data

  subroutine samp_engr(nc)

    !:: sample the energy

    implicit none

    integer, intent(in) :: nc

    !:: potential energy
    epot(nc)= sum(p)/dble(nmax*2)

    !:: kinetic energy
    ekin(nc)= sum(v*v)/dble(nmax*2)
```

```
  return

end subroutine samp_engr

subroutine samp_tmpr(nc)

  !:: sample the temperature

  implicit none

  integer, intent(in) :: nc
  integer :: i, k

  tmpr(nc)= sum(v*v)/dble(nmax*nd)

  return

end subroutine samp_tmpr

subroutine samp_prssr(nc)

  !:: sample the velocity

  implicit none

  integer, intent(in) :: nc

  real(8) :: dr(2), r2, r2i, r6i, ff(2)
  integer :: i, j, k, jbeg, jend, jnb

  !:: calculate the force and energy
  pr(nc)=0d0
  do i=1, nmax-1
     jbeg = point(i)
     jend = point(i+1)-1
     do jnb= jbeg, jend

        j  = list(jnb)

        dr=x(i,:)-x(j,:)

        !:: periodic boundary condition
        r2= 0d0
        do k=1, nd
           dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
           r2   = r2 + dr(k)*dr(k)
        end do

        if(r2 ≤ rcut2) then
           r2i=1D0/r2
           r6i=r2i**3
           ff =r2i*r6i*(r6i-.5D0)*dr

           do k=1, nd
              pr(nc)= pr(nc) + ff(k)*dr(k)
           end do

        end if
```

```
      end do
   end do

   pr(nc)= pr(nc)*48d0

   do i=1, nmax
      do k=1, nd
         pr(nc) = pr(nc) + v(i,k)*v(i,k)
      end do
   end do

   pr(nc)= pr(nc)/(dble(nd)*vol)

   return

end subroutine samp_prssr

subroutine output

   implicit none

   integer :: i, n

   !:: save the position ...
   open(12, file= 'pos.dat')
   do i=1, nmax
      write(12, '(2F16.7)') x(i, :)
   end do
   close(12)

   !:: save the velocity ...
   open(12, file= 'vel.dat')
   do i=1, nmax
      write(12, '(2F16.7)') v(i, :)
   end do
   close(12)

   !:: save the temperature
   open(13, file= 'tmpr.dat')
   do n=1, nsamp
      write(13, '(F16.7)') tmpr(n)
   end do
   close(13)

   !:: save the pressure
   open(14, file= 'pressure.dat')
   do n=1, nsamp
      write(14, '(F16.7)') pr(n)
   end do
   close(14)

   !:: save the pressure
   open(15, file= 'engr.dat')
   do n=1, nsamp
      write(15, '(2F16.7)') ekin(n), epot(n)
   end do
   close(15)

   return
```

```
end subroutine output

subroutine adj_pos

  implicit none

  integer :: i, k

  do i=1, nmax
     do k=1, nd
        if(x(i,k) > bsize(k)) x(i,k)= x(i,k) - bsize(k)
        if(x(i,k) < 0d0) x(i,k)= x(i,k) + bsize(k)
     end do
  end do

  return

end subroutine adj_pos

subroutine samp_vacf

  !:: sample the velocity auto-correlation ---


  !  0         it0       2*it0      3*it0      4*it0       5*it0     t0max*it0

  !  |----------|----------|----------|----------|-----------|----------|
  !

  implicit none

  !:: sample the velocity autocorrelation
  integer, parameter :: it0=200, nsamp=4, neq= 4000
  integer, parameter :: tmax = 2000
  integer, parameter :: t0max= tmax/it0+1
  real(8), save :: vacf(tmax), r2t(tmax), dtime, time_vr(tmax), vrdr(nd)
  real(8), save :: xx0(nmax, nd, t0max), vv0(nmax, nd, t0max)
  integer, save :: ntel, tt0, t1, idelt, time0(t0max), ntime(tmax)

  integer :: i, j, k

  integer, save :: nc= 0
  logical, save :: init_vacf=.true.

  nc= nc + 1
  if(mod(nc, nsamp)≠0 ∨ nc<neq) return

  if(init_vacf) then
     dtime = delt * dble(nsamp)
     vacf = 0D0
     r2t = 0D0
     ntime = 0
     ntel =0
     tt0=0

     init_vacf=.false.
  end if
```

```
   ntel=ntel+1

   !:: Define a new {t=0} and save the position and velocity
   if ( mod(ntel−1, it0) ≡ 0) then
      tt0= tt0+1
      t1= mod(tt0−1, t0max) + 1

      time0(t1)=ntel
      do i=1, nmax
         xx0(i, :, t1) =x(i,:)
         vv0(i, :, t1) =v(i,:)
      end do

   end if

   !:: compute the correlation between the current time
   !:: and all the previous points where (t=0).
   do j=1, min(tt0, t0max)
      idelt= ntel− time0(j) +1
      if(idelt ≤ tmax) then
         ntime(idelt) = ntime(idelt) +1
         do i=1, nmax
            vacf(idelt)= vacf(idelt) + sum(v(i,:) * vv0(i,:,j))

            do k=1, nd
               vrdr(k)= x(i,k)− xx0(i,k,j)
               vrdr(k)= vrdr(k)  − Dnint(vrdr(k)/bsize(k))*bsize(k)
               r2t (idelt)= r2t (idelt) + vrdr(k)**2
            end do

         end do
      end if
   end do

   if (nc ≡ nstep) then
      do i=1, tmax
         time_vr(i) = dtime* (i +.5D0)
         vacf(i) = vacf(i) /dble(nmax * ntime(i))
         r2t (i) = r2t (i) /dble(nmax * ntime(i))
      end do

      open (12, File='vacf0.dat')
      do j=1, tmax
         write(12, '(3E20.10)') time_vr(j), vacf(j), r2t(j)
      end do
      close(12)

   end if

   return

end subroutine samp_vacf
```

```
  subroutine samp_rdf

    !:: radial distribution function

    implicit none

    integer, parameter :: nbin= 100, nsamp=50, neq= 4000
    real(8), parameter :: rmax= 10d0, dg= rmax/dble(nbin)

    real(8), save :: g(nbin)=0, rg(nbin)
    integer, save :: ngt=0, nc=0

    real(8) :: dr(nd), r, r2, vb

    integer :: i, j, k, nh

    nc= nc + 1
    if(mod(nc,nsamp) ≠ 0 ∨ nc<neq) return

    ngt= ngt + 1
    do i=1, nmax−1
       do j=i+1, nmax
          dr= x(i,:)−x(j,:)

          r2= 0d0
          do k=1, nd
             dr(k)= dr(k)  − Dnint(dr(k)/bsize(k))*bsize(k)
             r2= r2 + dr(k)*dr(k)
          end do

          r = dsqrt(r2)
          nh= r/dg + 1

          if(nh≤nbin) g(nh)= g(nh) + 2d0

       end do
    end do

    if(nc ≡ nstep) then
       do i=1, nbin
          rg(i) = dble(i−1)*dg

          vb = Dacos(−1d0)*dble(i**2 − (i−1)**2)*dg**2 !volume of the bin
          g(i)= g(i)/(nmax*rho*vb*ngt)
       end do

       open (12, File='rdf0.dat')
       do i=1, nbin
          write(12, '(2E20.10)') rg(i), g(i)
       end do
       close(12)

    end if

  end subroutine samp_rdf

end program main
```

```
program main

  !:: MOLECULAR DYNAMICS SIMULATION IN A THREE DIMENSIONAL LENNARD-JONES SYSTEM
  !::
  !::        ISOTHERMAL SIMULATION WITH ANDERSEN'S THERMOSTATS
  !.....................................................................

  use ran_mod

  implicit none

  !:: size of the system
  integer, parameter :: nx=9, ny=9, nz=12, na=4, nmax= nx*ny*nz*na, nd=3

  !:: time period
  integer, parameter :: nstep= 40000

  !:: sample frequency and number of samples
  integer, parameter :: nsf= 20, nsamp= nstep/nsf

  !:: step size for the integrator
  real(8), parameter :: delt= 1D-3

  !:: particle position, velocity, force, potential, etc.
  real(8) :: x(nmax,nd), v(nmax,nd), f(nmax,nd), p(nmax), xs(nmax, nd)
  real(8) :: bsize(nd)

  !:: density
  real(8), parameter :: rho= 0.1d0, vol= dble(nmax)/rho

  !:: Verlet list
  integer, parameter :: ncf=20 ! check the list every ncf steps
  real(8), parameter :: rcut = 2.5d0, rlist=3.2d0
  real(8), parameter :: rcut2= rcut*rcut
  real(8), parameter :: rlst2= rlist*rlist
  integer, parameter :: maxnb= 140
  integer :: point(nmax), list(nmax*maxnb)

  !:: target temperature and collosion frequency
  real(8), parameter :: tp= 3d0
  real(8), parameter :: nu= 0.001d0

  !:: adjust the potential energy
  real(8), parameter :: pcut= (rcut**(-12.0) - rcut**(-6.0))

  !:: Quantities being sampled
  real(8), dimension(nsamp) :: tmpr, pr, ekin, epot

  integer :: i, ntime

  !:: initialize the system ...
  call INIT_POS

  call INIT_VEL

  !:: time integration
  do ntime=1, nstep

     call INTEGRATE
```

```
     call SAMP_DATA

  end do

  call OUTPUT

  print*, sum(pr)/dble(nsamp)

contains

  subroutine init_pos

    !:: initial the postion the the particle on a f.c.c. lattice

    implicit none

    real(8) :: a0, dx, dy, dz
    integer :: i, j, k, m, n

    a0= dexp( dlog(4d0/rho)/3d0 )

    dx= a0; dy= a0; dz=a0

    bsize= (/ dble(nx)*dx, dble(ny)*dy, dble(nz)*dz /)


    x(1, :)= (/0D0,   0D0,   0D0/)
    x(2, :)= (/0D0,   a0/2D0, a0/2D0/)
    x(3, :)= (/a0/2D0, 0D0,   a0/2D0/)
    x(4, :)= (/a0/2D0, a0/2D0,  0D0/)

    n=0
    do i=1, nx
       do j=1, ny
          do k=1, nz
             do m=1, na
                x(n+m, :)= x(m, :) + (/dble(i-1)*a0, dble(j-1)*a0, dble(k-1)*a0/
)
             end do
             n=n+ na
          end do
       end do
    end do

    call comp_force

    return

  end subroutine init_pos

  subroutine init_vel

    !:: initialize the velocity

    integer :: isd(4), i, k

    isd= (/3, 5, 7 ,9/)
```

```
   !:: sample the velocity from (0,1) Gaussian
   do i=1, nmax
      do k=1, nd
         v(i, k)= dlarnd(3, isd)
      end do
   end do

   !:: shift the velocity so that the mean is zero
   do k=1, nd
      v(:,k)=v(:,k) - sum(v(:,k))/dble(nmax)
   end do

   v= dsqrt(tp)*v

   return

end subroutine init_vel


subroutine comp_force

   implicit none

   integer, save :: nf= 0
   logical, save :: update = .true.

   real(8) :: dr(nd), r2, r2i, r6i, ff(nd), pp
   integer :: i, j, k, jbeg, jend, jnb, nlist

   if(mod(nf, ncf) ≡ 0) call check(update)
   nf= nf + 1

   !:: update the Verlet's list
   if (update) then

      call savex

      update= .false.

      !:: setting up the Verlet's list
      nlist=0
      do i = 1, nmax-1
         point(i)=nlist+1

         do j = i+1, nmax
            dr=x(i,:)-x(j,:)

            !:: periodic boundary condition
            r2= 0d0
            do k=1, nd
               dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
               r2   = r2 + dr(k)*dr(k)
            end do

            if (r2 < rlst2) then
               !:: add the particle to the list
               nlist=nlist+1
               list(nlist)=j
            end if
```

```
         end do
      end do
      point(nmax)=nlist+1
   end if

   !:: calculate the force and energy
   f= 0d0; p= 0d0
   do i=1, nmax-1
      jbeg = point(i)
      jend = point(i+1)-1
      do jnb= jbeg, jend
         j  = list(jnb)
         dr=x(i,:)-x(j,:)

         !:: periodic boundary condition
         r2= 0d0
         do k=1, nd
            dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
            r2   = r2 + dr(k)*dr(k)
         end do

         if(r2 ≤ rcut2) then
            r2i= 1D0/r2
            r6i= r2i**3
            ff = r2i*r6i*(r6i-.5D0)*dr

            f(i,:)= f(i,:) + ff
            f(j,:)= f(j,:) - ff

            pp  = r6i*(r6i - 1d0) - pcut
            p(i)= p(i) + pp
            p(j)= p(j) + pp

         end if
      end do
   end do
   f= f*48d0
   p= p*4D0

   return

end subroutine comp_force

subroutine savex

   !:: save the array x to xs

   xs= x

end subroutine savex

subroutine check(update)

   !:: check the Verlet's list

   implicit none

   logical, intent(out) :: update
```

```
   real(8) :: dispmx, dr(nd), r2
   integer :: i, k

   dispmx = 0D0
   do i = 1, nmax
      dr= x(i, :) − xs(i, :)

      r2= 0d0
      do k=1, nd
         dr(k)= dr(k) − dnint(dr(k)/bsize(k))*bsize(k)
         r2 = r2 + dr(k)*dr(k)
      end do

      dispmx = Dmax1(r2, dispmx)
   end do

   dispmx = 2D0 * Dsqrt (dispmx)
   update = (dispmx > (rlist −rcut))

   return

 end subroutine check

 subroutine integrate

   implicit none

   real(8) :: tmp
   integer, save :: isd(4)= (/3,5,6,7/)
   integer :: i

   v= v + f*delt/2d0
   x= x + v*delt

   call adj_pos

   call comp_force

   v= v + f*delt/2d0

   print*, ntime, sum(v*v)+sum(p)

   !:: Andersen's thermostats
   do i=1, nmax
      if(dlarnd(1, isd) < nu) then
         v(i,1)= dlarnd(3, isd)
         v(i,2)= dlarnd(3, isd)
         v(i,3)= dlarnd(3, isd)
         v(i,:)= v(i,:)*dsqrt(tp)
      end if
   end do

   return

 end subroutine integrate

 subroutine samp_data

   implicit none
```

```
    integer, save :: ntime = 0
    integer :: nc

!    call samp_vacf
!    call samp_rdf

    ntime= ntime + 1
    if(mod(ntime, nsf) ≠ 0) then
       return
    else
       nc= ntime/nsf
       call samp_engr (nc)
       call samp_tmpr (nc)
       call samp_prssr(nc)
    end if

    return

  end subroutine samp_data

  subroutine samp_engr(nc)

    !:: sample the energy

    implicit none

    integer, intent(in) :: nc

    !:: potential energy
    epot(nc)= sum(p)/dble(nmax*2)

    !:: kinetic energy
    ekin(nc)= sum(v*v)/dble(nmax*2)

    return

  end subroutine samp_engr

  subroutine samp_tmpr(nc)

    !:: sample the temperature

    implicit none

    integer, intent(in) :: nc
    integer :: i, k

    tmpr(nc)= sum(v*v)/dble(nmax*nd)

    return

  end subroutine samp_tmpr

  subroutine samp_prssr(nc)

    !:: sample the velocity

    implicit none
```

```
   integer, intent(in) :: nc

   real(8) :: dr(nd), r2, r2i, r6i, ff(nd)
   integer :: i, j, k, jbeg, jend, jnb

   !:: calculate the force and energy
   pr(nc)=0d0
   do i=1, nmax-1
      jbeg = point(i)
      jend = point(i+1)-1
      do jnb= jbeg, jend

         j  = list(jnb)

         dr=x(i,:)-x(j,:)

         !:: periodic boundary condition
         r2= 0d0
         do k=1, nd
            dr(k)= dr(k) - dnint(dr(k)/bsize(k))*bsize(k)
            r2   = r2 + dr(k)*dr(k)
         end do

         if(r2 ≤ rcut2) then
            r2i=1D0/r2
            r6i=r2i**3
            ff =r2i*r6i*(r6i-.5D0)*dr

            do k=1, nd
               pr(nc)= pr(nc) + ff(k)*dr(k)
            end do

         end if
      end do
   end do

   pr(nc)= pr(nc)*48d0

   do i=1, nmax
      do k=1, nd
         pr(nc) = pr(nc) + v(i,k)*v(i,k)
      end do
   end do

   pr(nc)= pr(nc)/(dble(nd)*vol)

   return

 end subroutine samp_prssr

 subroutine output

   implicit none

   integer :: i, n

   !:: save the position ...
   open(12, file= 'pos1.dat')
```

```
    do i=1, nmax
       write(12, '(2F16.7)') x(i, :)
    end do
    close(12)

    !:: save the velocity ...
    open(12, file= 'vel1.dat')
    do i=1, nmax
       write(12, '(2F16.7)') v(i, :)
    end do
    close(12)

    !:: save the temperature
    open(13, file= 'tmpr1.dat')
    do n=1, nsamp
       write(13, '(F16.7)') tmpr(n)
    end do
    close(13)

    !:: save the pressure
    open(14, file= 'pressure1.dat')
    do n=1, nsamp
       write(14, '(F16.7)') pr(n)
    end do
    close(14)

    !:: save the pressure
    open(15, file= 'engr1.dat')
    do n=1, nsamp
       write(15, '(2F16.7)') ekin(n), epot(n)
    end do
    close(15)

    return

  end subroutine output

  subroutine adj_pos

    implicit none

    integer :: i, k

    do i=1, nmax
       do k=1, nd
          if(x(i,k) > bsize(k)) x(i,k)= x(i,k) - bsize(k)
          if(x(i,k) < 0d0) x(i,k)= x(i,k) + bsize(k)
       end do
    end do

    return

  end subroutine adj_pos

  subroutine samp_vacf

    !:: sample the velocity auto-correlation ---
```

```
!  0         it0       2*it0      3*it0      4*it0       5*it0      t0max*it0

!  |----------|----------|----------|----------|-----------|----------|
!


implicit none

!:: sample the velocity autocorrelation
integer, parameter :: it0=20, nsamp=1, neq= 40000
integer, parameter :: tmax = 500
integer, parameter :: t0max= tmax/it0+1
real(8), save :: vacf(tmax), r2t(tmax), dtime, time_vr(tmax), vrdr(nd)
real(8), save :: xx0(nmax, nd, t0max), vv0(nmax, nd, t0max)
integer, save :: ntel, tt0, t1, idelt, time0(t0max), ntime(tmax)

integer :: i, j, k

integer, save :: nc= 0
logical, save :: init_vacf=.true.

nc= nc + 1
if(mod(nc, nsamp)≠0 ∨ nc<neq) return

if(init_vacf) then
   dtime = delt * dble(nsamp)
   vacf = 0D0
   r2t = 0D0
   ntime = 0
   ntel =0
   tt0=0

   init_vacf=.false.
end if

ntel=ntel+1

!:: Define a new {t=0} and save the position and velocity
if ( mod(ntel−1, it0) ≡ 0) then
   tt0= tt0+1
   t1= mod(tt0−1, t0max) + 1

   time0(t1)=ntel
   do i=1, nmax
      xx0(i, :, t1) =x(i,:)
      vv0(i, :, t1) =v(i,:)
   end do

end if

!:: compute the correlation between the current time
!:: and all the previous points where (t=0).
do j=1, min(tt0, t0max)
   idelt= ntel− time0(j) +1
   if(idelt ≤ tmax) then
      ntime(idelt) = ntime(idelt) +1
      do i=1, nmax
         vacf(idelt)= vacf(idelt) + sum(v(i,:) * vv0(i,:,j))
```

```
            do k=1, nd
               vrdr(k)= x(i,k)- xx0(i,k,j)
               vrdr(k)= vrdr(k)  - Dnint(vrdr(k)/bsize(k))*bsize(k)
               r2t (idelt)= r2t (idelt) + vrdr(k)**2
            end do

         end do
      end if
   end do

   if (nc ≡ nstep) then
      do i=1, tmax
         time_vr(i) = dtime* (i +.5D0)
         vacf(i) = vacf(i) /dble(nmax * ntime(i))
         r2t (i) = r2t (i) /dble(nmax * ntime(i))
      end do

      open (12, File='vacf1.dat')
      do j=1, tmax
         write(12, '(3E20.10)') time_vr(j), vacf(j), r2t(j)
      end do
      close(12)

   end if

   return

end subroutine samp_vacf

subroutine samp_rdf

   !:: radial distribution function

   implicit none

   integer, parameter :: nbin= 100, nsamp=50, neq= 40000
   real(8), parameter :: rmax= 6d0, dg= rmax/dble(nbin)

   real(8), save :: g(nbin)=0, rg(nbin)
   integer, save :: ngt=0, nc=0

   real(8) :: dr(nd), r, r2, vb

   integer :: i, j, k, nh

   nc= nc + 1
   if(mod(nc,nsamp) ≠ 0 ∨ nc<neq) return

   ngt= ngt + 1
   do i=1, nmax-1
      do j=i+1, nmax
         dr= x(i,:)-x(j,:)

         r2= 0d0
         do k=1, nd
            dr(k)= dr(k)  - Dnint(dr(k)/bsize(k))*bsize(k)
            r2= r2 + dr(k)*dr(k)
         end do
```

```
           r = dsqrt(r2)
           nh= r/dg + 1

           if(nh≤nbin) g(nh)= g(nh) + 2d0

        end do
     end do

     if(nc ≡ nstep) then
        do i=1, nbin
           rg(i) = dble(i−1)*dg

           vb = dble(i**nd − (i−1)**nd)*dg**nd !volume of the bin
           if(nd≡2) then
              vb= vb * dacos(−1d0)
           elseif(nd≡3) then
              vb= vb * dacos(−1d0)*4d0/3d0
           end if
           g(i)= g(i)/(nmax*rho*vb*ngt)
        end do

        open (12, File='rdf1.dat')
        do i=1, nbin
           write(12, '(2E20.10)') rg(i), g(i)
        end do
        close(12)

     end if

   end subroutine samp_rdf

end program main
```