

Exercises Lecture XIV

Classical fluids:

simulation with Metropolis Monte Carlo (and with Molecular Dynamics)

1. Monte Carlo Simulation of hard disks

Write a code for a Monte Carlo simulation of hard disks in 2D. One example is `hd-MC.f90`, which makes use of the *periodic boundary conditions* and the *minimum image convention* to calculate the minimum distance (function `separation`).

If σ is the *diameter of the disks*, the highest possible density is $\rho_{max}=2/(\sqrt{3}\sigma^2)$. It is convenient to use σ as unit length and measure all lengths in terms of σ and use the *reduced density*, defined in general as $\rho^* = \rho\sigma^d$, where d is the dimensionality of the system. The highest possible *reduced density* is $\rho_{max}^* = \rho_{max}\sigma^2 = 1.1547$, corresponding to the maximum *packing fraction* $f = area_{occupied}/area_{available} = \pi/(2\sqrt{3}) = 0.9069$.

- (a) Start simulating the fluid with a density close to the maximum one. To this purpose, it is convenient to set the initial positions of the particles on a hexagonal (or triangular) lattice that ensures the maximum *packing fraction*. Choose for instance $N = 16$ and a rectangular box with dimension $L_x = 4.41\sigma$ and $L_y = 0.5\sqrt{3}L_x$. Calculate ρ^* and compare it with ρ_{max}^* . A reasonable first choice for the maximum random displacement in a Monte Carlo simulation is $dx_{max} = dy_{max} = 0.1\sigma$. Calculate the corresponding *acceptance ratio*. Allow at least 500 MC steps for equilibration and average over $nmcs \geq 500$. Calculate $g(r)$. A reasonable choice for the bin width dr for the calculation of $g(r)$ is $dr=0.1$.
- (b) Reduce progressively ρ^* , saving the configuration of a run and using it as the input for the new run at lower ρ^* . Keeping the ratio L_x/L_y fixed, it is sufficient to rescale homogeneously all the positions. It may be convenient to vary progressively also dx_{max} and dy_{max} in order to keep an *acceptance ratio* of the order of 50%. Calculate $g(r)$ for $\rho^* = 0.95, 0.92, 0.88, 0.85, 0.80, 0.70, 0.60$, and 0.30 ; plot and compare the profiles (how many peaks? where? ...)
- (c) Take “snapshots” of the disks at intervals of about 10 to 20 MC steps per particle. Do you see any evidence of the solid becoming a fluid at lower densities?

2. Monte Carlo simulation of a Lennard-Jones system

Consider particles interacting with the Lennard-Jones potential:

$$v(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right]$$

It is convenient to use the adimensional quantities $E^* = E/\epsilon$, $T^* = k_B T/\epsilon$, $\rho^* = \rho\sigma^2$ for energy, temperature and density respectively. For numerical simulations use Periodic Boundary Conditions and the *minimum image convention*, with a *cutoff radius* for the potential consistent with the size of the simulation box. (*No draft code is given*)

- (a) Start with $T^* = 0$ and calculate the energy E_0^* of the ground state of the system. Choose $N = 16$, $L_x = 4.5\sigma$, $L_y = (6\sqrt{3}/2)\sigma$, and the particles on a triangular lattice: the system is therefore close to the equilibrium, and a few MC steps are already enough to have a good estimate of E_0^* . Does the energy per particle change if you consider bigger systems at the same density?
- (b) Increase the temperature $T^* = 3.5$ and calculate E^* and $g(r)$.
- (c) Describe qualitatively $g(r)$ and compare it with the hard disks case.
- (d) Repeat the calculations for a smaller density, expanding by a factor of 1.5 the dimensions of L_x and L_y . Compare with the previous results and with the hard disks case with the same density.

3. Molecular dynamics of a Lennard-Jones system (Optional)

The program LJ-MD.f90, from Gould-Tobochnik, considers a bidimensional Lennard-Jones system, and makes use of the *velocity-Verlet* algorithm for the numerical integration of the Newton equations of motion to perform a *molecular dynamics* simulation. In 1D (with obvious extension in higher dimensions) the algorithm is:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2$$

$$v_{n+1} = v_n + \frac{1}{2} (a_{n+1} + a_n) \Delta t$$

The new position x_{n+1} is used to find the new acceleration a_{n+1} , which is used together with a_n to obtain the new velocity v_{n+1} .

- (a) Consider a system with $N = 16$ particles in a square box with $L = 6$. Choose $\Delta t = 0.01$ and test the program: follow the trajectories of the 16 particles making a plot, and check that the total energy is approximately conserved.
- (b) Calculate the pair correlation function $g(r)$ for some of the cases with density and temperature proposed in the exercise with Monte Carlo simulation. Compare the results obtained with the two methods.


```

        read(*,'(a)') ans
        if (ans=='y') then
            call read_config()
        else
            ! CREATES a new triangular lattice
            dx = Lx / float(Nx)          ;          dy = Ly / float(Ny)
! max. packing
! dx and dy must be separately .ge. sqrt(3)/2 (in units of sigma)
! and the product dx*dy .ge. sqrt(3)/2
            expr = ((dx<(sqrt(3.)/2)) .or. (dy<(sqrt(3.)/2) ) .or. & ((dx*dy)<(sqrt(3.)/2)))
            if (expr) call error('error in initial',' dx*dy = ',dx*dy)
            m = 0
            do j = 1,Ny,2
                do i = 1,Nx
                    m = m + 2
                    if (m>N) call error('error in initial',' m>N! m = ',float(m))
                    ! this should never occur! check!!!
                    x(m-1) = (i-0.75)*dx
                    y(m-1) = (j-0.5) *dy
                    x(m)   = (i-0.25)*dx
                    y(m)   = (j+0.5) *dy
                end do
            end do
        end if
        return
    end subroutine

subroutine move()
! pick up a particle randomly, generates trial coordinates,
! apply PBC, check overlap
    integer :: itrial,i
    real    :: xtrial,ytrial,r,ran1,xold,yold
    real , dimension(2) :: ran2
    do i=1,N
        call random_number(ran1)
        itrial = int(N*ran1 + 1)    ! choose a particle
        if(itrial>N) then
            print*,'error generating ran1'
            stop
        end if
        xold=x(itrial)
        yold=y(itrial)
        call random_number(ran2)
        xtrial = x(itrial) + (2.*ran2(1)-1.)*dxmax
        ytrial = y(itrial) + (2.*ran2(2)-1.)*dymax
        call cell(xtrial,ytrial)
        call overlap(itrial,xtrial,ytrial)
    end do

```

```

        end do
        return
end subroutine

subroutine cell(xtrial,ytrial)
! input  xtrial e ytrial,
! output : the same but rescaled using PB (x in [0,Lx[ and y in [0,Ly[)
        real , intent(inout):: xtrial,ytrial
! floor(A) return the largest integer < or = A
! What follows is OK whatever xtrial and ytrial are
        xtrial = xtrial - Lx*floor(xtrial/Lx)
        ytrial = ytrial - Ly*floor(ytrial/Ly)
        if(xtrial<0 .or. xtrial>=Lx .or. ytrial<0 .or. ytrial>=Ly) &
call error('error in cell',' xtrial =',xtrial)
        return
end subroutine

subroutine separation(dx,dy)
! in input  distances dx and dy , which (being already rescaled)
!           are between 0 and L
! in output the same, but rescaled using minimum image convention
!           and therefore = or < L/2)
        real , intent(inout) :: dx,dy
        dx = dx - Lx*int(kx*dx)
        dy = dy - Ly*int(ky*dy)
        return
end subroutine

subroutine overlap(itrial,xtrial,ytrial)
! decide to accept or not after cheching overlap;
! in case, update arrays x,y
        integer, intent(in) :: itrial
        integer :: j
        real ,intent(in) :: xtrial,ytrial
        real :: r2,dx,dy
! disks overlap if distance < 1 (diameter)
do j = 1,N
        if (itrial /= j) then
                dx = x(j) - xtrial
                dy = y(j) - ytrial
                call separation(dx,dy)      ! calculate true distances
                r2 = dx*dx + dy*dy
                if (r2<1.0) return          ! overlap!
        end if
end do
accept = accept + 1

```

```

        x(itrial) = xtrial
        y(itrial) = ytrial
    return
end subroutine

```

```

subroutine correl()
! array gcum contains the number of particles at distance
! between r and r+dr from a given particle.
! gcum is calculated for N/2 particles (to count once each pair)
    integer :: ibin,i,j
    real :: r2,dx,dy
    do i=1,N-1
        do j=i+1,N
            dx = x(i) - x(j)
            dy = y(i) - y(j)
            call separation(dx,dy)
            r2 = dx*dx + dy*dy
            ibin = int(sqrt(r2)/dr)+1
            if (ibin<=nbin) then
gcum(ibin) = gcum(ibin) + 1
!ngcum=ngcum +1
            end if
        end do
    end do
    return
end subroutine

```

```

subroutine output()
! calculate the normalized pair correlation function g(r) ; this is done after each MC step
! dividing gcum for the number of samples, density, area 2\pi r dr
! of the circular strip from the referring particle. Divide by N/2
! because gcum has been calculated by summing over N/2 particles.
! Note: max. separation is max(Lx/2,Ly/2) (in PBC)
    integer :: ir
    real :: rmax,xnorm,r,g,area,pi,rho_max
    pi = 2*asin(1.)
    rho_max=2/sqrt(3.)
    rmax = nbin * dr
    write(*,fmt='(a23,1x,f6.2,a1)') ' rho/rho_max=',rho/rho_max
    write(*,*) ' acceptance ratio:',real(accept)/(N*nmcs)
    open(unit=3,file='g_of_r.dat',status='unknown')
    write(3,*) '# number of particles (even) =',N
    write(3,*) '# dimensions Lx, Ly of the box =',Lx,Ly
    write(3,*) '# reduced density (rho)=',rho
    write(3,*) '# data production MC steps per particle (nmcs)=',nmcs
    write(3,*) '# bin dr for g(r) =',dr

```

```

write(3,*)'# acceptance ratio =',float(accept)/(N*nmcs)
write(3,*)'# rmax =',rmax
xnorm = 2./(rho*nmcs*N)
do ir = 1,nbin
    r      = ir*dr + 0.5*dr  ! r in the middle of the circular shell
    area   = 2.0*pi*r*dr    ! area of the shell
    g      = gcum(ir)*xnorm/area
    write(3,*)r,g
end do
close(3)
return
endsubroutine

```

```

subroutine save_config()
!save config in a file
integer :: i
character(len=20) :: fileout
print*, ' output filename to save config in a file >'
read(*,'(a)')fileout
open(unit=3,file=fileout,status='replace',action='write')
write(3,*)Lx,Ly
do i=1,N
write(3,*)x(i),y(i)
enddo
return
endsubroutine

```

```

subroutine read_config()
real :: Lxold,Lyold,xscale,yscale
character(len=20) :: filein
integer :: i
print*, ' filename old config. >'
read(*,'(a)') filein
open(unit=2,file=filein,status='unknown')
read(2,*)Lxold,Lyold
xscale = Lx / Lxold
yscale = Ly / Lyold
do i = 1,N
    read(2,*) x(i),y(i)
    x(i) = x(i) * xscale  ! uniformly rescale all x positions
    y(i) = y(i) * yscale  ! uniformly rescale all y positions
enddo
close(2)
endsubroutine

```

```

subroutine error(comment1,comment2,dummy)
character(len =16) :: comment1
character(len =10) :: comment2
real :: dummy
write(*,'(1x,a16,a10,1x,f10.4)')comment1,comment2,dummy
stop
return
endsubroutine

endmodule common

!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

program hdMC
use common
implicit none
integer :: imcs
character(1) :: ans

Lx=4.41*sqrt(0.9503/0.3)
Ly=0.5*sqrt(3.)*Lx
print*, 'default values for Lx,Ly=',Lx,Ly
print*, ' change? (y/n)'
read(*,'(a)')ans
if(ans=='y')then
    print*, ' insert Lx,Ly >'
    read(*,*)Lx,Ly
end if

call initial()

do imcs = 1, nequil    !equilibration
    call move()
enddo

gcum = 0. ;    accept = 0
do imcs = 1, nmcs
    call move()
    call correl()
enddo

call output()
call save_config()
deallocate(x,y,gcum)
stop
endprogram hdMC

```



```

integer, public :: N
real (kind = double), public :: Lx,Ly,t,timebig
real(kind = double),public,dimension (:),allocatable :: x,y,vx,vy
real(kind = double),public,dimension (:),allocatable :: collision_time, partner

```

contains

```

subroutine initial(vsum,rho,area)
  real (kind = double), intent (out) :: vsum,rho,area
  real :: a_x,a_y,vmax,rnd
  integer :: i,nx,ny,row,col
  character(len = 20) :: start,file_name
  character(len = 100) :: dum
  t = 0.0
  print *, "read file (f) or lattice start (l) = "
  read *, start
  if (start == "f" .or. start == "f") then
    print *, "file name = "
    read *, file_name
    open (unit=7,file=file_name,status="old",action="read")
    read (unit=7,fmt = *) N
    read (unit=7,fmt = *) Lx,Ly
    allocate(x(N))
    allocate(y(N))
    allocate(vx(N))
    allocate(vy(N))
    read (unit=7,fmt = *) dum
    do i = 1,N
      read (unit=7,fmt = *) x(i),y(i)
      print *, x(i),y(i)
    end do
    read (unit=7,fmt = *) dum
    do i = 1, N
      read (unit=7,fmt = *) vx(i),vy(i)
    end do
    close(unit=7)
  else if (start == "l" .or. start == "l") then
    print *, "N (a square...)= "
    read *, N      ! assume that sqrt(N) is an integer
    allocate(x(N))
    allocate(y(N))
    allocate(vx(N))
    allocate(vy(N))
    print *, "Lx = "
    read *, Lx
    print *, "Ly = "
  end if
end subroutine initial

```

```

read *, Ly
print *, "vmax = "
read *, vmax
nx = sqrt(real(N))
ny = nx
if (nx >= Lx .or. nx >= Ly) then
    print *, "box too small"
    stop
end if
a_x = Lx/nx          ! "lattice" spacing
a_y = Ly/ny
i = 0
do col = 1,nx
    do row = 1,ny
        i = i + 1
        x(i) = (col - 0.5)*a_x
        y(i) = (row - 0.5)*a_y
        ! choose random positions and velocities
        call random_number(rnd)
        vx(i) = (2*rnd - 1)*vmax
        call random_number(rnd)
        vy(i) = (2*rnd - 1)*vmax
    end do
end do
end if
call check_overlap()      ! check if two disks overlap
call check_momentum()
allocate(partner(N))
allocate(collision_time(N))
area = Lx*Ly
rho = N/area
timebig = 1.0e10
vsum = 0                  ! virial sum
do i = 1,N
    partner(i) = N
end do
collision_time(N) = timebig
! set up initial collision lists
do i = 1,N
    call uplist(i)
end do
end subroutine initial

subroutine check_momentum()
real (kind = double) :: vxsum, vysum, vxcm, vyvm
!vxsum = 0.0

```

```

!vysum = 0.0
! compute total center of mass velocity (momentum)
vxsum = sum(vx)
vysum = sum(vy)
vxcm = vxsum/N
vycom = vysum/N
vx = vx - vxcm
vy = vy - vycom
end subroutine check_momentum

subroutine minimum_collision_time(i,j,tij)

  real (kind = double), intent (out) :: tij
  integer, intent (out):: i,j

  integer :: k
  ! locate minimum collision time
  tij = timebig
  do k = 1, N
    if (collision_time(k) < tij) then
      tij = collision_time(k)
      i = k
    end if
  end do
  j = partner(i)
end subroutine minimum_collision_time

subroutine move(tij)
  real (kind = double), intent (in) :: tij
  integer :: k
  do k = 1,N
    collision_time(k) = collision_time(k) - tij
    x(k) = x(k) + vx(k)*tij
    y(k) = y(k) + vy(k)*tij
    x(k) = pbc(x(k),Lx)
    y(k) = pbc(y(k),Ly)
  end do
end subroutine move

subroutine reset_list(i,j)
  integer, intent (in) :: i,j
  integer :: k,test
  ! reset collision list for relevant particles
  do k = 1,N
    test = partner(k)
    if (k == i .or. test == i .or. k == j .or. test == j) then

```

```

        call uplist(k)
    end if
end do
call downlist(i)
call downlist(j)
end subroutine reset_list

subroutine check_overlap()
    real (kind = double) :: tol,dx,dy,r2,r
    integer :: i,j
    tol = 1.0e-4
    do i = 1,N - 1
        do j = i + 1,N
            dx = separation(x(i) - x(j),Lx)
            dy = separation(y(i) - y(j),Ly)
            r2 = dx*dx + dy*dy
            if (r2 < 1.0) then
                r = sqrt(r2)
                if (1.0 - r > tol) then
                    print *, "particles ",i," and ",j,"overlap"
                    stop
                end if
            end if
        end do
    end do
end subroutine check_overlap

subroutine kinetic_energy(ke)
    real (kind = double), intent (out) :: ke
    integer :: i
    ke = 0.0
    do i = 1,N
        ke = ke + vx(i)*vx(i) + vy(i)*vy(i)
    end do
    ke = 0.5*ke
    print "(a,f13.6)", "kinetic energy =", ke/N
end subroutine kinetic_energy

subroutine uplist(i)
    integer, intent (in) :: i
    integer :: j
    ! look for collisions with particles j > i
    if (i == N) then
        return
    end if
    collision_time(i) = timebig

```

```

do j = i + 1,N
    call check_collision(i,j)
end do
end subroutine uplist

subroutine downlist(j)
    integer, intent (in) :: j
    integer :: i
    ! look for collisions with particles i < j
    if (j == 1) then
        return
    end if
    do i = 1,j - 1
        call check_collision(i,j)
    end do
end subroutine downlist

subroutine check_collision(i,j)
    integer, intent (in) :: i,j
    real (kind = double) :: dx,dy,dvx,dvy,bij,tij,r2,v2,discr
    integer :: xcell,ycell
    ! consider collisions between i and periodic images of j
    do xcell = -1,1
        do ycell = -1,1
            dx = x(i) - x(j) + xcell*Lx
            dy = y(i) - y(j) + ycell*Ly
            dvx = vx(i) - vx(j)
            dvy = vy(i) - vy(j)
            bij = dx*dvx + dy*dvy
            if (bij < 0) then
                r2 = dx*dx + dy*dy
                v2 = dvx*dvx + dvy*dvy
                discr = bij*bij - v2*(r2 - 1.0)
                if (discr > 0.0) then
                    tij = (-bij - sqrt(discr))/v2
                    if (tij < collision_time(i)) then
                        collision_time(i) = tij
                        partner(i) = j
                    end if
                end if
            end if
        end do
    end do
end do
end subroutine check_collision

subroutine contact(i,j,virial)

```

```

real (kind = double), intent (out) :: virial
integer, intent (in) :: i,j
real (kind = double) :: dx,dy,dvx,dvy,factor,delvx,delvy
! compute collision dynamics for particles i and j at contact
dx = separation(x(i) - x(j),Lx)
dy = separation(y(i) - y(j),Ly)
dvx = vx(i) - vx(j)
dvy = vy(i) - vy(j)
factor = dx*dvx + dy*dvy
delvx = - factor*dx
delvy = - factor*dy
vx(i) = vx(i) + delvx
vx(j) = vx(j) - delvx
vy(i) = vy(i) + delvy
vy(j) = vy(j) - delvy
virial = delvx*dx + delvy*dy
end subroutine contact

subroutine output(collisions,temperature,vsum,rho,area)
real (kind = double), intent (in) :: temperature,vsum,rho,area
integer, intent (in) :: collisions
real :: mean_virial,mean_pressure
print "(a,i6)", "collisions =",collisions
print "(a,f10.4)", "t =",t
print *, ""
mean_virial = vsum/(2.0*t)
mean_pressure = rho*temperature + mean_virial/area
print "(a,f10.4)", "P =", mean_pressure
end subroutine output

subroutine save_config(i,j,tij)
real (kind = double), intent (inout) :: tij
integer, intent (inout) :: i,j
character(len = 32) :: config
integer :: k
! move particles away from collision for final configuration
call minimum_collision_time(i,j,tij)
call move(0.5*tij)
print *, "file name of configuration?"
read *, config
print *, config
open (unit=1,file=config,status="replace",action="write")
write (unit=1, fmt="(i4)") N
write (unit=1, fmt="(2f13.6)") Lx,Ly
write (unit=1,fmt="(t3,a,t20,a)") "x","y"
do k = 1,N

```

```

        write (unit=1, fmt="(2f13.6)") x(k),y(k)
    end do
    write(unit=1, fmt="(t3,a,t20,a)") "vx","vy"
    do k = 1,N
        write(unit=1,fmt="(2f13.6)") vx(k),vy(k)
    end do
    close(unit=1)
end subroutine save_config

end module common

program hd
    ! dynamics of system of hard disks
    ! program based in part on fortran program of Allen and Tildesley
    use periodic
    use common

    real (kind = double) :: virial,temperature,vsum,rho,area,ke,tij
    integer :: collisions, i,j,nshow
    call initial(vsum,rho,area)
    call kinetic_energy(ke)
    temperature = ke/N
    collisions = 0                ! number of collisions
    nshow = 10                   ! show output every nshow collisions
    do
        if (collisions > 100) then
            exit
        end if
        call minimum_collision_time(i,j,tij)
        ! move particles forward and reduce collision times by tij
        call move(tij)
        t = t + tij
        collisions = collisions + 1
        call contact(i,j,virial)    ! compute collision dynamics
        vsum = vsum + virial
        if (modulo(collisions,nshow) == 0) then
            call output(collisions,temperature,vsum,rho,area)
        end if
        ! reset collision list for relevant particles
        call reset_list(i,j)
        ! check for overlaps to debug program
        ! call check_overlap()
    end do
    call save_config(i,j,tij)
end program hd

```

```

!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
!  LJ-MD.f90 (from Gould-Tobochnik)
!
!  simulate a 2D system of particles interacting via the
!  Lennard-Jones potential;
!  Use periodic boundary conditions and minimum image convention;
!  Use velocity-Verlet algorithm to integrate the equation of motion
!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
module periodic
public :: separation, pbc
integer, public, parameter :: double = 8
contains

function separation(ds,L) result (separation_result)
! minimum image convention.
! NOTE that this implementation assumes that the max dist between
! particles is L in each direction, i.e. it is OK if called after
! the function pbd
  real (kind = double), intent (in) :: ds,L
  real (kind = double) :: separation_result
  if (ds > 0.5*L) then
    separation_result = ds - L
  else if (ds < -0.5*L) then
    separation_result = ds + L
  else
    separation_result = ds
  end if
end function separation

function pbc(pos,L) result (f_pbc)
! use PBC, fold a coordinate from [-L,2L] in [0,L]
  real (kind = double), intent (in) :: pos,L
  real (kind = double) :: f_pbc
  if (pos < 0.0) then
    f_pbc = pos + L
  else if (pos > L) then
    f_pbc = pos - L
  else
    f_pbc = pos
  end if
end function pbc

end module periodic

module common

```

```

use periodic
private

public :: initial,allocate_arrays,Verlet,accel,force
public :: check_momentum,save_config,output

integer, public :: N
real (kind = double), public :: Lx,Ly,t,dt,dt2
real (kind = double), public, dimension (:), allocatable :: x,y,vx,vy,ax,ay
integer, dimension(2), public :: seed

contains

subroutine initial(ke,kecum,pecum,vcum,area)
  real (kind = double), intent (out) :: ke,kecum,pecum,vcum,area
  character(len = 20) :: start,file_name
  character(len = 100) :: dum
  integer :: n1,i,row,col
  real :: a_x,a_y,vmax,rnd
  dt = 0.005
  dt2 = dt*dt
  ! seed(1) = 1239
  ! seed(2) = 1111
  ! call random_seed(put=seed)
  print *, "read data (d), read file (f), or lattice start (l) ="
  read *, start
  if (start == "D" .or. start == "d") then
    N = 16 ! for the sake of simplicity, a perfect square
    call allocate_arrays()
    Lx = 6.0
    Ly = 6.0
    x(1:8) = (/ 1.09,3.12,0.08,0.54,2.52,3.03,4.25,0.89 /)
    x(9:16) = (/ 2.76,3.14,0.23,1.91,4.77,5.10,4.97,3.90 /)
    y(1:8) = (/ 0.98,5.25,2.38,4.08,4.39,2.94,3.01,3.11 /)
    y(9:16) = (/ 0.31,1.91,5.71,2.46,0.96,4.63,5.88,0.20 /)
    vx(1:8) = (/ -0.33,0.12,-0.08,-1.94,0.75,1.70,0.84,-1.04 /)
    vx(9:16) = (/ 1.64,0.38,-1.58,-1.55,-0.23,-0.31,1.18,0.46 /)
    vy(1:8) = (/ -0.78,-1.19,-0.10,-0.56,0.34,-1.08,0.47, 0.06 /)
    vy(9:16) = (/ 1.36,-1.24,0.55,-0.16,-0.83,0.65,1.48,-0.51 /)
  else if (start == "l" .or. start == "L") then
    print *, "N (a square...)= "
    read *, N ! assume that sqr(N) is an integer
    call allocate_arrays()
    print *, "Lx = "
    read *, Lx
    Ly = 0.5*sqrt(3.0)*Lx
  end if
end subroutine initial

```

```

n1 = sqrt(real(N))
a_x = Lx/n1          ! lattice spacing
a_y = 0.5*sqrt(3.0)*a_x
vmax = 1.0
i = 0
! triangular lattice
do row = 1,n1
  do col = 1,n1
    i = i + 1
    x(i) = (col + 0.5*modulo(row,2) - 1)*a_x
    y(i) = (row - 0.5)*a_y
    ! choose random velocities
    call random_number(rnd)
    vx(i) = (2*rnd - 1)*vmax
    call random_number(rnd)
    vy(i) = (2*rnd - 1)*vmax
  end do
end do
do i = 1,N
  x(i) = pbc(x(i),Lx)
  y(i) = pbc(y(i),Ly)
end do
else if (start == "f" .or. start == "f") then
  print *, "file name = "
  read *, file_name
  open (unit=7,file=file_name,status="old",action="read")
  read (unit=7,fmt = *) N
  read (unit=7,fmt = *) Lx,Ly
  call allocate_arrays()
  read (unit=7,fmt = *) dum
  do i = 1,N
    read (unit=7,fmt = *) x(i),y(i)
  end do
  read (unit=7,fmt = *) dum
  do i = 1, N
    read (unit=7,fmt = *) vx(i),vy(i)
  end do
  close(unit=7)
end if
call check_momentum()
ke = 0.0                ! kinetic energy
do i = 1,N
  ke = ke + vx(i)*vx(i) + vy(i)*vy(i)
end do
ke = 0.5*ke
! initialize sums

```

```

    kecum = 0.0
    pecum = 0.0
    vcum = 0.0
    area = Lx*Ly
    ! print heading for data
    write(unit=8,fmt="(t6,a,t17,a,t27,a,t37,a)")"time","E","T","P"
end subroutine initial

subroutine allocate_arrays()
    allocate(x(N))
    allocate(y(N))
    allocate(vx(N))
    allocate(vy(N))
    allocate(ax(N))
    allocate(ay(N))
end subroutine allocate_arrays

subroutine Verlet(ke,pe,virial)
    real (kind = double), intent (out) :: ke
    real (kind = double), intent (inout) :: pe,virial
    integer :: i
    real (kind = double) :: xnew,ynew

    do i = 1, N
        xnew = x(i) + vx(i)*dt + 0.5*ax(i)*dt2
        ynew = y(i) + vy(i)*dt + 0.5*ay(i)*dt2
        x(i) = pbc(xnew,Lx)
        y(i) = pbc(ynew,Ly)
        ! partially update velocity using old acceleration
        vx(i) = vx(i) + 0.5*ax(i)*dt
        vy(i) = vy(i) + 0.5*ay(i)*dt
    end do
    call accel(pe,virial)          ! new acceleration
    ke = 0.0
    do i = 1, N
        ! complete the update of the velocity using new acceleration
        vx(i) = vx(i) + 0.5*ax(i)*dt
        vy(i) = vy(i) + 0.5*ay(i)*dt
        ke = ke + vx(i)*vx(i) + vy(i)*vy(i)
    end do
    ke = 0.5*ke
    t = t + dt
end subroutine Verlet

subroutine accel(pe,virial)
    real (kind = double), intent (inout) :: pe,virial

```

```

real (kind = double) :: dx,dy,fxij,fyij,pot
integer :: i,j
do i = 1, N
    ax(i) = 0.0
    ay(i) = 0.0
end do
pe = 0.0
virial = 0.0
do i = 1, N - 1
    ! compute total force on particle i
    do j = i + 1, N
        ! due to particles j > i
        dx = separation(x(i) - x(j),Lx)
        dy = separation(y(i) - y(j),Ly)
        ! acceleration = force because mass = 1 in reduced units
        call force(dx,dy,fxij,fyij,pot)
        ax(i) = ax(i) + fxij
        ay(i) = ay(i) + fyij
        ax(j) = ax(j) - fxij    ! Newton's third law
        ay(j) = ay(j) - fyij
        pe = pe + pot
        virial = virial + dx*ax(i) + dy*ay(i)
    end do
end do
end subroutine accel

subroutine force(dx,dy,fx,fy,pot)
! Lennard-Jones potential
    real (kind = double), intent (in) :: dx,dy
    real (kind = double), intent (out) :: fx,fy,pot

    real (kind = double) :: r2,rm2,rm6,f_over_r

    r2 = dx*dx + dy*dy
    rm2 = 1.0/r2
    rm6 = rm2*rm2*rm2
    f_over_r = 24*rm6*(2*rm6 - 1)*rm2
    fx = f_over_r*dx
    fy = f_over_r*dy
    pot = 4.0*(rm6*rm6 - rm6)
end subroutine force

subroutine check_momentum()
    real (kind = double) :: vxsum, vysum,vxcm,vycm
    ! compute total center of mass velocity (momentum)
    vxsum = sum(vx)
    vysum = sum(vy)
    vxcm = vxsum/N

```

```

    vycm = vysum/N
    vx = vx - vxcm
    vy = vy - vycm
end subroutine check_momentum

subroutine save_config()
    character(len = 32) :: config
    integer :: i
    print *, "file name of configuration?"
    read *, config
    print *, config
    open (unit=1,file=config,status="replace",action="write")
    write (unit=1, fmt="(i4)")N
    write (unit=1, fmt="(2f13.6)")Lx,Ly
    write (unit=1,fmt="(t3,a,t20,a)")"x","y"
    do i=1,N
        write (unit=1, fmt="(2f13.6)") x(i),y(i)
    end do
    write(unit=1, fmt="(t3,a,t20,a)") "vx","vy"
    do i = 1,N
        write(unit=1,fmt="(2f13.6)")vx(i),vy(i)
    end do
    close(unit=1)
end subroutine save_config

subroutine output(ke,pe,virial,kecum,vcum,ncum,area)
    integer, intent (inout) :: ncum
    real (kind = double), intent(in) :: ke,pe,virial,area
    real (kind = double), intent(inout) :: kecum,vcum
    real (kind = double) :: E,mean_ke,P

    ncum = ncum + 1
    E = ke + pe                ! total energy
    kecum = kecum + ke
    vcum = vcum + virial
    mean_ke = kecum/ncum      ! still need to divide by N
    P = mean_ke + (0.5*vcum)/ncum ! mean pressure * area
    P = P/area
    ! mean_ke/N      ! mean kinetic temperature
    write(unit=8,fmt="(4f10.4)") t,E,mean_ke/N,P
end subroutine output

end module common

program md
! program adapted from Gould & Tobochnik, Chapter 8

```

```

! simple N(N-1)/2 calculation of forces
use periodic
use common
real (kind = double) :: E
real (kind = double) :: ke,kecum,pecum,vcum,area,pe,virial
integer :: ncum
open(unit=8,file='energy',status='replace')
call initial(ke,kecum,pecum,vcum,area)
call accel(pe,virial)
E = ke + pe                ! total energy
ncum = 0                   ! number of times data accumulated
do
  if (t > 2.0) exit
  call Verlet(ke,pe,virial)
  call output(ke,pe,virial,kecum,vcum,ncum,area)
end do
close(8)
call save_config()
end program md

```