A simple Molecular Dynamics Program

Algorithm 1

```
program MD                                  schematic MD program
        call init                           initialization
        call neighbors                      Verlet, cell lists
        t = 0
do while (t.lt. tmax)                       MD loop
        call force (f,en)                   determine forces
        call integrate (f,en)               integrate EOM
        call sample                         calculate averages
        call update                         store r(t), v(t), a(t)
        call neighbors                      check/update lists
        t = t + h                           h = MD time step
enddo
stop
end
```

Algorithm 2: Initialization of MD program

```
subroutine init                          MD initialization
        sumv = 0                         Zero all velocities
        sumv2 = 0                        Zero kinetic energy
do i = 1, npart
        x(i) = lattice_pos(i)            place particles in lattice positions
        v(i) = (ranf() – 0.5)            give random velocities
        sumv = sumv + v(i)               center of mass velocity
        sumv2 = sumv2 + v(i)**2          kinetic energy
enddo
        sumv = sumv/npart                center of mass velocity
        sumv2 = sumv2/npart              mean squared velocity
        scalef = sqrt(3*temp/sumv2)      scale factor for velocities
do i = 1, npart                          set desired kinetic energy and
        v(i) = (v(i) – sumv)*scalef      set velocity center of mass to zero
        xm(i) = x(i) – v(i)*h            position previous time step
enddo
return
end
```

Comments to Algorithm 2.

x(i) = lattice_pos(i)

- obviously, this is a generic lattice.
- one can code for generating fcc, hcp, bcc, sc, etc lattices.

v(i) = (ranf() - 0.5)

- this initial velocity distribution is not Maxwellian.
- ranf() just uniform distribution in (-0.5, 0.5) for each v_α.
- velocities needs rescaling to adjust to desired T.

In thermal equilibrium: $\left\langle v_\alpha^2 \right\rangle = k_B T / m; \quad \alpha = x, y, z$

Instantaneous T at time t becomes: $$k_B T(t) = \sum_i^N \frac{m v_{\alpha,i}^2(t)}{3N}$$

To adjust T(t) to T (desired), simply scale all velocities with $[T/T(t)]^{1/2}$. Initial setting of T is not critical, T changes anyway during equilibration.

xm(i) = x(i) – v(i)*h

- simple approximation for $x(t - \Delta t)$.
- only needed if Verlet integrator is used, 1st step only.
- not needed for Velocity Verlet, which is self-starting.

Algorithm 3: Calculation of Forces

```
subroutine force(f,en)                    determine forces and energy
        en = 0                            set energy to zero
    do i = 1, npart
        f(i) = 0                          set forces to zero
    enddo
do i = 1, npart – 1                       loop over the i of pairs (list)
  do j = i+1, npart                       inner loop over the j of pairs (list)
        xr = x(i)–x(j)                    calculate Rij for each pair
        xr = xr–box*nint(xr/box)          PBC
        r2 = xr**2                        store Rij squared for each pair
        if (r2.lt.rc2) then               test cut-off
            r2i = 1/r2
            r6i = r2i**3
            ff = 48*r2i*r6i*(r6i–0.5)     LJ forces for each pair
            f(i) = f(i) + ff*xr
            f(j) = f(j) – ff*xr
            en=en+4*r6I*(r6i–1)-ecut      accumulate LJ energy for all pairs
        endif
  enddo
enddo
return
end
```

Comments to Algorithm 3.

Always good practice to ZERO important quantities such as E and F.

For efficiency reasons, factors 4 and 48 are taken out of the force loop.

Term ecut is the value of the potential at $r = r_c$. This is the implementation of potential truncation and shift in MD.

For LJ, this becomes: $\text{ecut} = 4\left(\frac{1}{r_c^{12}} - \frac{1}{r_c^{6}}\right)$

For any valid pair, one must compute the force, and contribution to potential energy, for each cartesian component. For the x-component of the force:

$$f_x(r) = -\frac{\partial u(r)}{\partial x} = -\left(\frac{x}{r}\right)\left(\frac{\partial u(r)}{\partial r}\right)$$

For a Lennard-Jones system (in reduced units): $f_x(r) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} - 0.5\frac{1}{r^{6}}\right)$

Algorithm 4: Integration of Equations of Motion

```
subroutine integrate (f,en)                integrate EOM
        sumv = 0
        sumv2 = 0
do i = 1, npart                            loop over all atoms
        xx = 2*x(i)-xm(i)*h*f(i)           Verlet algorithm – new positions
        vi = (xx-xm(i))/ (2*h)             new velocities
        sumv=sumv+vi                       center of mass velocity
        sumv2=sumv2+vi**2                  accumulate total kinetic energy
        xm(i) = x(i)                       update positions previous time step
        x(i) = xx                          update positions current time step
enddo
        temp = sumv2 / (3*npart)           instantaneous temperature
        etot = en+sum2 / (2*npart)         total energy: E = V + K
return
end
```

Obs: Loop over atoms contains inner loops over x, y, z for positions and velocities. This obviously applies to all previous algorithms.

Algorithm 5: Making a Verlet List

```
subroutine new_verlet_list
            do i = 1, npart                          initialize list
                nlist (i) = 0
                xv (i) = x (i)                       store positions of particle
            enddo
do i = 1, npart –1                                   loop over pairs
      do j = i + 1, npart
            xr = x (i) – x (j)                       calculate Δr for x, y, z
            if (xr .gt. hbox) then                   nearest image
                  xr = xr – box
            else if (xr .lt. -hbox) then
                  xr = xr + box
            endif
            if (abs (xr) .lt. rv) then               add to lists if Δr < skin (rv)
                  nlist (i) = nlist (i) + 1          total nr. of part. in list for particle i
                  nlist (j) = nlist (j) + 1          total nr. of part. in list for particle j
                  list (i, nlist (i)) = j            Verlet list of particle i
                  list (j, nlist (j)) = i            Verlet list of particle j
            endif
      enddo
enddo
return
end
```

Obs: Array xv(i) contains position of particles at the moment the list is made.

Comments to Algorithm 5.

In MD, forces are calculated at the same time, $f_{ij} = -f_{ji}$, so it is sufficient to have a Verlet list with half the number of particles, as long as the i-j interaction is accounted for in either the list of particle I or j.

In MC, each particle is considered separately, therefore it is convenient to have a complete Verlet list for each particle.

Remember, the list must be updated periodically. For well behaved systems, such as in 3D simulations of solids far away from their melting point, frequent updates are not required once equilibrium has been reached. Updates at 10, 50 even 100 steps are sufficient.

In many studies, however, for example irradiation of surfaces, one needs to update the list essentially every time step, which makes the use of Verlet lists inefficient. Linked cells lists are the solution in this situations.

Algorithm 6: Making a Cell List

```
subroutine new_cell_list (rc)                    makes cell list with cell size r_c using
                                                 a linked-list algorithm
rn = box / int (box/rc)                          determine size of cells r_n ≥ r_c
      do icel = 0, ncel – 1
            hoc (icel) = 0                       set head of chain to 0 for each cell.
      enddo
do i = 1, npart                                  loop over the particles
            icel = int (x (i) / rn)              determine cell number
            link_l (i) = hoc (icel)              link list the head of chain of cell icel
            hoc (icel) = i                       make particle i the head of chain
enddo
return
end
```

Obs:

In each cell, a particle i is named head-of-chain and stored in *hoc(icel)*.

To this particle the next particle in the cell (chain) is linked via the array linked-list array *link_l(i)*. If link_l(i) = 0, no more particles in cell (chain).

Algorithm 7: Making a Verlet List Using a Cell List.

```
subroutine make_verlet_using_cell_list
            call new_cell_list (rv)                          make the cell lists (Blue font)
                  do i = 1, npart                            initialize list
                  nlist (i) = 0
                  xv (i) = x (i)                             store positions of particles
            enddo
do i = 1, npart
   icel = int (x(i) / rn)                                    determine cell number
   do ncel = 1, neigh                                        loop over neighbor cells
        jcel = neigh (icel, ncel)                            number of the neighbor
        j = hoc (jcel)                                       head of chain of the cell
            do while (j .ne. 0)                              make Verlet (Green font)
                  if (i .ne. j) then
                        xr = x(i) – x(j)                     calculate Δr
                        if (xr .gt. hbox) then               nearest image, PBC
                              xr = xr – box
                        else if (xr .lt. –hbox) then
                              xr = xr + box
                        endif
                        if (abs (xr) .lt. rv) then           add to Verlet list if Δr < skin
                              nlist (i) = nlist (i) + 1
                              nlist (j) = nlist (j) + 1
                              list (i, nlist (i)) = j
                              list (j, nlist (j)) = i
                        endif
                  endif
            j = link_l (j)                                   next particle in cell list
            enddo
   enddo
enddo
return
end
```

Comments to Algorithms 5, 6 and 7.

Verlet lists scale scale as N^2, cell lists are order N methods.

Use Verlet only if $N_{atoms\_in\_list} <<< N_{total}$ in system.

Cell lists are not advantageous for small systems.

In MC, when using Verlet and/or Verlet_Cells combinations, it is important to limit maximum random particle displacements $\Delta r_{max} < 2\,[r_v - r_c]$, i.e. twice the size of the skin. In MD, the only limitation comes from the integration scheme used.

If coded properly, one can play several tricks with the linked-cell-list method. Most often, cell size can be reduced to allow only 1 atom/cell. Scheme is still very effective since cells don't change neighbors, one only needs to identify in which cell a particle is.

Calculating Properties in MD Simulations

Energy calculations

Potential energy per atom: $$U(t) = \frac{1}{N}\sum_{i=1}^{N} U_i(t) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j>i}^{N} U(|r_i(t) - r_j(t)|$$

Most efficiently computed together with forces during force calculations.

Kinetic energy per atom: $$K(t) = \frac{1}{2N}\sum_{i=1}^{N} m_i v_i^2(t)$$

Almost always computed when integrating the EOM, with the integrator.

Total energy per atom: $$E_{tot}(t) = U(t) + K(t)$$

Usually done after obtaining K(t), unless long range, other corrections added.

Equi-partition of energy, temperature: the average energy of every quadratic term in energy for classical systems has the same value, 1/2 $k_B T$.

For a 3D one-component system:

$$\langle U(t) \rangle \approx \langle K(t) \rangle \approx \frac{3}{2} k_B T; \quad T = \frac{2}{3k_B} \langle K(t) \rangle; \quad T = \frac{1}{3k_B N} \sum_{i=1}^{N} m_i v_i^2(t)$$

Why perform energy calculations?

* Check for total energy conservation.

* Energy flow from kinetic to potential energy can indicate the occurrence of a phase transition in the system.

* A jump in the caloric curve E(T) points to a first order phase transition (melting).

* In systems far from equilibrium analysis of the energy flow redistribution gives useful information.

* A detailed atomic-level analysis of energy-per-atom in a static/quenched configuration can help to identify defects and analyze the structure.

Checking that the MD simulations runs properly

Time step selection

At first, one needs to choose a proper time step, Δt. Its size depends generally on the typical atomic motion.

Low temperature systems consisting of heavy atoms can tolerate larger Δt. High temperature systems with light atoms moving faster need smaller Δt.

The rule of thumb is that you want enough points, 20 – 100, to correctly describe the real trajectory of an atom moving around (i.e. its period).

On average, the safest bet is to start with a time step of the order of femptoseconds, or 10^{-15} seconds.

Energy Conservation

In microcanonical, (NVE), simulations, the total energy, i.e. the sum of potential and kinetic energy has to be conserved in the limit of the infinitesimal time step.

As you increase Δt, the total energy will start to fluctuate and depending on your algorithm could also have a systematic drift.

A systematic drift is very bad because it means that your system is cooling down, that is, the total energy is going to get closer and closer to the potential energy (the KE is decreasing until your system slows down to a hold).

A simple algorithm like the Verlet does not show any systematic drift.

You want the fluctuation in your total energy to be much smaller than the fluctuation in your potential energy say, 2 or 3 orders of magnitude smaller so that locally your system is varying its potential energy but its total energy is, actually, unchanged.

You know that Verlet-like algorithms scale with Δt^4. It is then possible to easily find the optimal time step for which the total energy curve becomes flat.

Thermalization

You need to make sure that you reached thermodynamic equilibrium, so that you lose memory of your initial conditions.

If we let the system evolve long enough, our trajectory will diverge from our initial exact trajectory that was poorly chosen, and it will become more and more representative of the thermodynamical ensemble that we are simulating. It's still very tricky to understand if we are reaching equilibrium.

The thermodynamical observable (average) to monitor is the mean square displacement (how much atoms fluctuate around their initial position).

At the beginning atoms will keep moving. The first peak at ~ 1/3 of a psec is a measure of the period of oscillation of an atom. At the very beginning of the simulation, atoms move and then they start feeling the potential from the other atoms and they settle in an oscillatory movement. And then you are starting to average over more and more oscillations.

In a 3D (bulk) simulation, you'll converge fairly quickly to the final exact results. In 2D simulations, surface atoms will take more time before finally converging to the average.

Mean Square Displacement (MSD)

$$\left\langle \Delta r^2 \right\rangle = \frac{1}{N}\sum_{i=1}^{N}\left[r_i(t_n) - r_i(t_0)\right]^2;$$ where n = number of time origins/samples

To monitor/compute MSD, equilibrium must first be reached. Only after that one should start computing properties/time averages. This holds true for all other properties.

Very useful quantity to monitor as it is connected with a number of important, measurable macroscopic properties:

Lindemann Criterion: $C_L = \sqrt{\left\langle \Delta r^2 \right\rangle}\Big/ d;$ where d = nearest neighbor distance

The criterion states that melting might be expected when $\langle \Delta r^2 \rangle$ reaches at least 10% of d.

Debye Temperature: $\Theta_D^2 = 3\hbar^2 T \Big/ m k_B \left\langle \Delta r^2 \right\rangle$

Can be calculated for bulk and surface layers, compared with theory and experimental values (surfaces).

Self-Diffusion Coefficient:

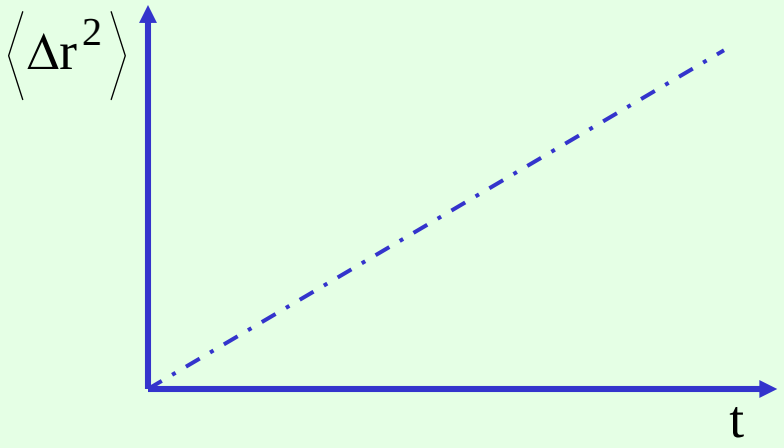
The Einstein expression for an isotropic fluid at equilibrium connects MSD and the self-diffusion coefficient:

$$2tD = \frac{1}{3}\left\langle \left| r_i(t) - r_i(t=0) \right|^2 \right\rangle$$

The time average above is only strictly valid in the limit of long times:

$$D = \lim_{t\to\infty} \frac{1}{6t}\left\langle \left| r_i(t) - r_i(t=0) \right|^2 \right\rangle$$

In practice, the MSD is accumulated over long enough times, and D is calculated from the slope of MSD vs. time.



Pressure:

To calculate the pressure, one can uses the virial expression for a homogenous system of particles:

$$P = \frac{Nk_BT}{V} + \frac{1}{3V}\left\langle \sum_{i=1}^{N} r_i \cdot f_{ij} \right\rangle$$

For pair-additive potentials, this becomes:

$$P = \frac{Nk_BT}{V} + \frac{1}{6V}\left\langle \sum_{i=1}^{N}\sum_{j\neq i}^{N} r_{ij} \cdot f_{ij} \right\rangle$$

where P is the pressure, V is the volume of primary simulation cell. The 1/6 factor is due to the fact that $f_{ij} = -f_{ji}$, to avoid double counting.

Best place to calculate the pressure is in the force subroutine, where all cartesian components of Δrij and fij are available.

Obs: This represents the “internal” pressure of the MD “ensemble” of particles.

Specific Heat:

$$\frac{\left\langle T^2 \right\rangle - \left\langle T \right\rangle^2}{\left\langle T \right\rangle^2} = \frac{3}{2N}\left(1 - \frac{3Nk_B}{2C_V}\right)$$

Best time to accumulate these time averages in the integrator subroutine.

Elastic moduli and constants:

E_{tot} is minimized for different a and bulk modulus B evaluated from the E vs. V curve.
Small tetragonal distortions can be applied to primary simulation cell:

$$\varepsilon = \begin{pmatrix} +\delta/2 & 0 & 0 \\ 0 & +\delta/2 & 0 \\ 0 & 0 & -\delta \end{pmatrix}$$

to calculate strain energy: $U_{tetr} = \frac{3}{4}\delta^2(C_{11} - C_{12})$

One can then use $B = \frac{C_{11} + 2C_{12}}{3}$ to calculate:

$$C_{11} = B + \frac{2}{3}(C_{11} - C_{12}) \qquad C_{12} = B - \frac{1}{3}(C_{11} - C_{12})$$

From trigonal distortions: $\varepsilon = \begin{pmatrix} 0 & \delta & 0 \\ \delta & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ one obtains: $U_{tri} = 2\delta^2 C_{44}$

to calculate:

The shear modulus: $$G = \frac{3C_{44} + C_{11} - C_{12}}{5}$$

Young's modulus: $$E = \frac{9BG}{3B + G}$$

Poisson ratio: $$\nu = \frac{1}{2}\left(1 - \frac{E}{3B}\right)$$

Autocorrelation Functions, Transport Coefficients:

One can use the so-called Green-Kubo relations to make the connection between a macroscopic property, more precisely a response property of the system, to equilibrium fluctuations of microscopic properties, called autocorrelation functions.

Diffusion Coefficient

The diffusion coefficient is a response property of the system to a concentration inhomogeneity, and is connected to the velocity-velocity autocorrelation function:

$$D = \lim_{t' \to \infty} \int_0^{t'} dt'' \left\langle v_x(t' - t'') v_x(0) \right\rangle = \int_0^{\infty} dt \left\langle v_x(t) v_x(0) \right\rangle$$

The velocity-velocity autocorrelation function represents the microscopic fluctuations at equilibrium. In MD, the Einstein relation and Green-Kubo relation for D are strictly equivalent.

Time autocorrelation functions can easily be calculated in MD:

$$\Psi(t) = \left\langle v_i(t_0) \cdot v_i(t_0 + t) \right\rangle = \frac{1}{N} \left\langle \sum_{i=1}^{N} v_i(t_0) \cdot v_i(t_0 + t) \right\rangle$$

Algorithm 8: Diffusion Coefficient using Velocity Autocorrelation Function.

```
Subroutine diffusion coefficient (switch, nsamp)
                                                          switch = 0 – initialization
                                                          switch = 1 – sample
                                                          switch = 2 – results

if  (switch.eq.0)  then                                   initialization
      ntel = 0                                            time counter
      dtime = dt * nsamp                                  time between two samples
      do i = 1, tmax                                      tmax total number of time steps
            ntime (i) = 0                                 number of samples for time i
              vacf (i) = 0
              r2t (i) = 0
      enddo

else if (switch.eq.1) then                                sample
      ntel = ntel +1
      if (mod (ntel, it0). eq. 0) then                    decide to take a new t = 0
            t0 = t0 + 1                                   update number of t = 0
            tt0 = mod (t0 – 1, t0max) + 1                 define new t = 0 each it0 time this sub is called
            time0 (tt0) = ntel                            store the time of t = 0
            do i = 1, npart
                  x0 (i, tt0) = x (i)                     store position for given t = 0
                  vx0 (i, tt0) = vx (i)                   store velocity for given t = 0
            enddo
      endif
      do t = 1, min (t0, t0max)                           update vacf and r2, for t = 0
            delt = ntel – time0 (t) + 1                   actual time minus t = 0
            if (delt . lt . tmax) then
                  ntime (delt) = ntime (delt) + 1
                  do i = 1, npart
                        vacf (delt) = vacf (delt) +vx (i) * vx0 (i, t)      update velocity autocorr.
                        r2t (delt) = r2t (delt) +      (x (i) – x0 (i, t)) ** 2      update mean-square displ.
                  enddo
            endif
      enddo
```

…….CONTINUE……

Algorithm 8: Diffusion Coefficient using Velocity Autocorrelation Function.

…….CONTINUE……

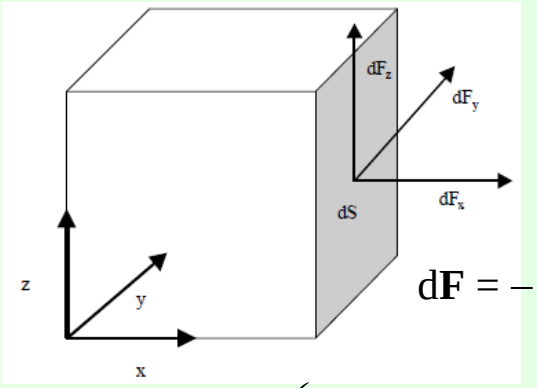
```
else if  (switch.eq.2)  then                          determine results
      do i = 1, tmax
               time = dtime * (i + 0.5)                time
               vacf (i) = vacf (i) / (npart * ntime (i))   volume velocity autocorr
               r2t (i) = r2t (i) / (npart * ntime (i))     mean square displacement
      enddo
endif
return

end
```

Shear Viscosity:

$$\eta = \frac{V}{k_B T}\int_0^\infty dt \left\langle P_{\alpha\beta}(0)\cdot P_{\alpha\beta}(t)\right\rangle;$$

where $\alpha\beta = xy, xz, yx, yz, zx, zy$ are

$$P_{\alpha\beta}(t) = \frac{1}{V}\left[\sum_i m_i v_{i\alpha}(t) v_{i\beta}(t) + \frac{1}{2}\sum_i \sum_{j\neq i} r_i v_{i\alpha}(t) f_{ij\beta}(t)\right]$$



$\mathbf{dF} = -\,\mathbf{dS}\cdot\mathbf{P}$

$$\mathbf{P} = \begin{pmatrix} P_{xx} & P_{xy} & P_{xz} \\ P_{yx} & P_{yy} & P_{yz} \\ P_{zx} & P_{zy} & P_{zz} \end{pmatrix}$$

Bulk Viscosity:

$$\eta_V = \frac{V}{9k_BT}\int_0^\infty dt\left\langle \delta P_{\alpha\alpha}(0)\cdot\delta P_{\beta\beta}(t)\right\rangle = \frac{V}{k_BT}\int_0^\infty dt\left\langle \delta P(0)\cdot\delta P(t)\right\rangle$$

where sum is over $\alpha,\beta = x, y, z$ and $P = 1/3\sum_\alpha P_{\alpha\alpha}$

Finally, rotational invariance gives: $\eta_V + \frac{4}{3}\eta = \frac{V}{k_BT}\int_0^\infty dt\left\langle \delta P_{\alpha\alpha}(0)\cdot\delta P_{\alpha\alpha}(t)\right\rangle$

Thermal Conductivity:

$$\lambda = \frac{V}{k_BT^2}\int_0^\infty dt\left\langle \dot{q}_\alpha(0)\cdot\dot{q}_\alpha(t)\right\rangle;$$ where $\alpha = x, y, z$ and the heat flux is:

$$\dot{q}_{i\alpha}(t) = \frac{1}{V}\left[E_i(t)v_{i\alpha}(t) + \frac{1}{2}\sum_i\sum_{j\neq i} r_{ij\alpha(t)}\left[v_i(t)\cdot f_{ij}(t)\right]\right]$$ with energy given by:

$$E_i(t) = \frac{1}{2}m_i v_i^2(t) + \frac{1}{2}\sum_{j\neq i}\Phi\left[r_{ij}(t)\right]$$

Pair distribution Function:

Also known as the radial distribution function, g(r), gives the probability of locating pairs of atoms separated by a distance r, relative to that for a completely random distribution at the same density. It is defined as:

$$g(r) = \rho^{-2} \left\langle \sum_{i=1}^{N} \sum_{j \neq i}^{N} \delta(r_i) \delta(r_j - r) \right\rangle = \frac{V}{N^2} \left\langle \sum_{i=1}^{N} \sum_{j \neq i}^{N} \delta(r - r_{ij}) \right\rangle$$

In the MD simulation, the delta function is replaced by a finite function (eg. given the value 1) over a small range of separations. A histogram of all pair separations that fall within this range is then accumulated over time during the MD run.

Very useful in MD study of fluids/liquids, as theory of liquids is based on pair correlations. g(r) is effectively a measure of the fluid structure. It's also needed to calculate "tail" corrections to potential energy and pressure induced by the truncation at the edge of simulation box (see next two slides).

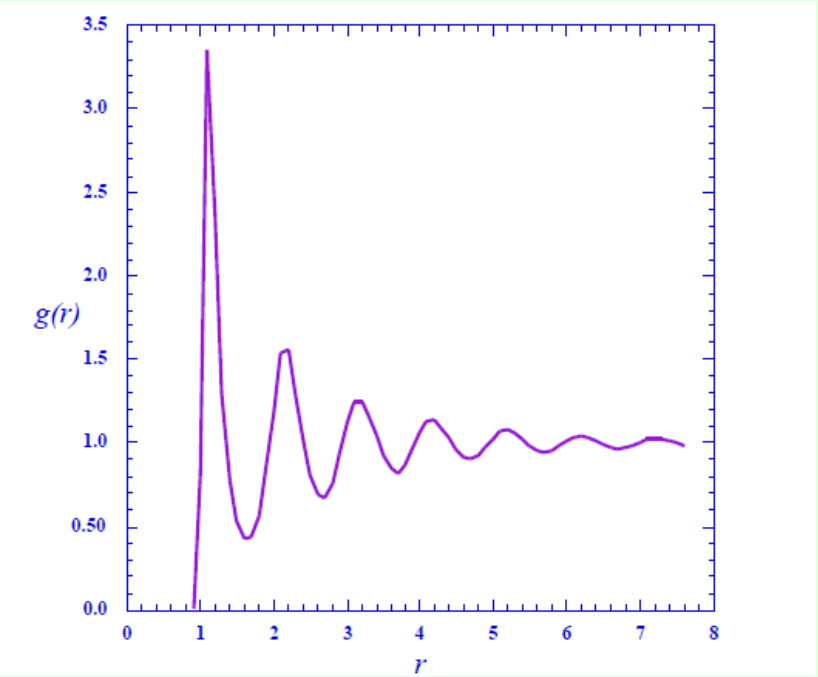
Various peaks in g(r) plots denote nearest neighbor shells.

Note that $r \to \infty$, $g(r) \to 1$.

Solid structures show up in g(r) in a more subtle manner, with bumps at nearest, next-nearest neighbors etc. height and location of peaks depends on type of lattice.

The Fourier transform of g(r) is called the static structure function, S_k. Its calculation is important as it can be compared with experimental data from neutron or X-ray scattering.

Typical g(r) plot.



The pair distribution function is also very important given that the ensemble average of any pair function ca be expressed as:

$$\left\langle a(r_i, r_j) \right\rangle = \frac{1}{V^2} \int dr_i dr_j g(r_i, r_j) a(r_i, r_j)$$

which is equivalent to: $$\langle A \rangle = \left\langle \sum_{i}^{N} \sum_{j>i}^{N} a(r_{ij}) \right\rangle = \frac{1}{2} N\rho \int_{0}^{\infty} a(r) g(r) 4\pi r^2 dr$$

If pair additivity holds, one can then write energy as:

$$E = \frac{3}{2} N k_B T + 2\pi N \rho \int_{0}^{\infty} r^2 v(r) g(r) dr$$

Similarly, the pressure can be expressed as:

$$PV = N k_B T - \frac{2}{3} \pi N \rho \int_{0}^{\infty} r^2 w(r) g(r) dr, \text{ with the pair virial: } w(r) = r \frac{d\Phi_{pot}(r)}{dr}$$

and the chemical potential as:

$$\mu = k_B T \ln(\rho \Lambda^3) + 4\pi\rho \int_{0}^{1} d\xi \int_{0}^{\infty} r^2 v(r) g(r, \xi) dr, \quad \text{where} \quad \Lambda = (h^2 / 2\pi m k_B T)^{1/2}$$

is the thermal de Broglie wavelength, and g(r,ξ) depends upon ξ, a param. coupling the two atoms which require separate integration.

Algorithm 9: Pair (Radial) Distribution Function.

```
Subroutine gr (switch)
                                                        switch = 0 – initialization
                                                        switch = 1 – sample
                                                        switch = 2 – results

if  (switch.eq.0)  then                                 initialization
      ngr = 0
      delr = box / (2*nhis)                             bin size
      do i = 1, nhis                                    nhis total number of bins
            g (i) = 0
      enddo

else if (switch.eq.1) then                              sample
      ngr = ngr +1
      do i = 1, npart-1
            do j = i+1, npart                           loop over all pairs
                  r = x (i) –x (j)
                  xr = xr – box*nint (xr/box)           PBC
                  r = sqrt (xr**2)
                  if (r.lt.box/2) then                  only within half box length
                        ig = int (r/delr)
                        g(ig) = g(ig) + 2               contribution for particle i & j
                  endif
            enddo
      enddo

else if (switch.eq.2) then                              determine g(r)
      do i = 1, nhis
            r = delg * (i+0.5)                          distance r
            vb = ((i+1)**3 - i**3)*delg**3              volume between bin i+1 & i
            nid = (4/3)*pi*vb*rho                       no. of ideal gas part. in vb
            g (i) = g (i) / (ngr*npart*nid)             normalize g(r)
      enddo
ednif

return
end
```

Time averaging

So far, most of the procedures discussed showed how to calculate properties at time t. However, in MD simulations meaningful results are only obtained if proper time averages are calculated over sufficiently long periods of time.

For any phase variable A(t) defined as $A(t) = \sum_{i=1}^{N} A_i(t)\delta(t - t_i)$

the time average < A(t) > over an arbitrary sampling interval τ is computed as:

$$\langle A(t) \rangle = \frac{1}{\tau}\int_0^\tau dt A(t) = \frac{1}{\tau}\int_0^\tau dt \sum_{i=1}^{N} A_i(t)\delta(t - t_i) = \frac{1}{\tau}\sum_{i=1}^{N}\int_0^\tau dt A_i(t)\delta(t - t_i) = \frac{1}{\tau}\sum_{i=1}^{N} A_i(t_i)$$

It is always a good practice to accumulate the totals of all quantities as they are computed, i.e the sum of all $A_i(t_i)$. Time averaging can easily, and should, be done in a separate subroutine.

What and how to do (FAQ)

Always check total energy and momentum conservation with all external and constraint forces switched off! Check EOM are correctly integrated, make sure forces are correctly calculated and that you don't miss interactions in your neighbor lists.

Program crashes at start up – check magnitude of forces, if too large lattice structure may be incorrect (atoms too close or too far). If lattice is OK, check forces and neighbor lists.

Temperature is not constant when simulation thermostat is switched on – check EOM and that thermostat constrain implementation is correct.

Program runs smoothly, energy and momentum are conserved, but average quantities keep growing – check the accumulation of particularly property, check your averaging routine, that you take the average over correct number of samples and totals are indeed accumulated over that number of samples.

Everything appears in good order, program, time averages all behave normally, so can you be sure results are correct? Compare with known, correct results from literature. For basic properties data is available in text books (cohesive energy, bulk/shear/Young moduli, etc. If successful, the code is probably OK, ready to do exiting science!