Molecular Dynamics for Multicomponent Systems

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville

Begun: January, 2002
Last Updated: March 3, 2002

Table of Contents

This handout assumes that you already have read the first two hand-outs in the series:

- The Working Man's Guide to Molecular Dynamics Simulations
- The Working Man's Guide to Obtaining Self Diffusion Coefficients from Molecular Dynamics Simulations

I. Introduction

Since many engineering problems deal with multicomponent mixtures, it is essential that we be able to simulate them. In this section of notes, we explain how we converted the single component molecular dynamics code to a multicomponent code for an arbitrary number of components.

The code is attached in the appendix. You can see for yourself that the basic structure of the code is unchanged. We merely need to change numerous details here and there. The numerical algorithm sections, the predictor and the corrector, do not need to be altered at all.

II. Intermolecular Potentials: Mixing Rules

In dealing with multicomponent mixtures, we need one additional piece of information. Presumably, we have a pairwise intermolecular potential, perhaps the Lennard-Jones 12-6 potential:

$$U_{LJ}(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{1}$$

The collision diameter, σ, and the well-depth, ε, are functions of the chemical identities of the two molecules that form the pair of interest. Generally, we know the values of the parameters only for pure components, meaning that both molecules in the pair are the same species. Therefore, we must have some method to obtain the “mixed parameters” from the pure parameters. One such method is called the Lorenz-Barthelot Mixing Rules.[fn] The collision diameter for a molecule of type A with a molecule of type B is the arithmetic average of the two pure component collision diameters.

$$\sigma_{AB} = \frac{(\sigma_A + \sigma_B)}{2} \tag{2}$$

The energetic parameter for the interaction of a molecule of type A with a molecule of type B is the geometric average of the two pure component parameters:

$$\varepsilon_{AB} = \sqrt{\varepsilon_A \varepsilon_B} \tag{3}$$

In this way, we can obtain parameters for AA, BB, and AB pairs in a binary mixture. Multicomponent mixture parameters are generated the same way, since the method assumes pairwise interactions.

If you look at the subroutine that performs the force evaluation, you will see that we have a vector labeled ntype which designates the chemical species of each molecule. This vector is used as the index to obtain the correct value of LJ parameters for any pair of atoms included in the neighbor list.

In the single component case, the FORTRAN code in the force evaluation loop looked like:

```
do m = 1, Nnbr, 1
    i = Nnbrlist(m,1)
    j = Nnbrlist(m,2)
    dis(1:3) = r(i,1:3) - r(j,1:3)
    do k = 1, 3, 1
        if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
        if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
    enddo
    dis2 = sum(dis*dis)
    if (dis2 .le. rcut2) then
        dis2i = 1.d0/dis2
        dis6i = dis2i*dis2i*dis2i
        dis12i = dis6i*dis6i
        U = U + ( sig12*dis12i - sig6*dis6i )
        fterm = (2.d0*sig12*dis12i - sig6*dis6i )*dis2i
        f(i,1:3) = f(i,1:3) + fterm*dis(1:3)
        f(j,1:3) = f(j,1:3) - fterm*dis(1:3)
        virial = virial - fterm*dis2
    endif
enddo
```

Now in the multicomponent case, we replace the two lines of code in red with the following five lines

```
        sig12t = sig12(ntype(i),ntype(j))
        sig6t = sig6(ntype(i),ntype(j))
        epst = eps(ntype(i),ntype(j))
        U = U + epst*( sig12t*dis12i - sig6t*dis6i )
        fterm = epst*(2.d0*sig12t*dis12i - sig6t*dis6i )*dis2i
```

The first three lines find the appropriate Lennard-Jones parameters for the atom pair. The next two lines are analogous to the the two lines replaced in the single component code.

Another notable change in moving from single component to multi-component codes involve changes in long range corrections to the potential energy and pressure.

III. Checking the Code

The easiest way to check a multicomponent code is to run a binary simulation where all of the components have the same σ, ε, and molecular weight. This simulation should yield the same results as the pure component simulations. For the code attached at the end of these notes, we verified that we could obtain the same liquid and vapor phase results using this procedure.

Warning: This code assumes that you have at least two molecules of each component. If you have 0 or 1 molecules of any given component, the code will crash gracelessly.

A note on the code provided: The multicomponent molecular dynamics code provided has a subroutine to compute the self-diffusion coefficients tacked onto the end of the simulation.

References

1. Shukla, K.P., Haile, J.M., *Mol.Phys.* **62** 1987 p. 617.

Appendix A. Multicomponent Molecular Dynamics Code in Fortran

```
      program mddriver
c
c  This code performs molecular dynamics simulations
c  in the canonical ensemble (specify T, V, and N)
c  for multicomponent mixtures
c
c  Author:  David Keffer
c  Department of Chemical Engineering, University of TN
c  Last Updated:  January 9, 2002
c
cx global maxstp kmsd N dt
c***********************************************************
c  VARIABLE DEFINITIONS AND DIMENSIONS
c***********************************************************
      implicit double precision (a-h, o-z)
      logical :: lmsd, lscale, lselfd                    ! logical variables
      character*12 :: cmsd, cout                 ! character variables
      double precision, allocatable :: props(:,:,:), r(:,:), v(:,:),
     & a(:,:), d3(:,:), d4(:,:), d5(:,:), f(:,:), rwopbc(:,:),
     & sigvec(:), epsvec(:), sig(:,:), eps(:,:), MW(:), xmass(:),
     &  xmassi(:), sig6(:,:), sig12(:,:), ulong(:), vlong(:), xmf(:),
     & tfac(:)
      integer, allocatable :: Nnbrlist(:,:), ntype(:), Nvec(:)
      double precision, dimension(1:5) :: dtv
      double precision, dimension(0:5) :: alpha
      double precision :: kb
      REAL(4), dimension(1:2) :: TA
c***********************************************************
c  PROGRAM INITIALIZATION
c***********************************************************
c
c     This code uses length units of Angstroms (1.0e-10 s)
c     time = fs (1.0e-15 s)
c     xmass = (1.0e-28 kg)
c     energy = aJ (1.0e-18 J)
c     Temperature = K
c
c    Define number of components
c
      ncomp = 2
      allocate (sigvec(1:ncomp), epsvec(1:ncomp),
     & sig(1:ncomp,1:ncomp), eps(1:ncomp,1:ncomp),
     & MW(1:ncomp), xmass(1:ncomp), xmassi(1:ncomp),
     & Nvec(0:ncomp), sig6(1:ncomp,1:ncomp),
     & sig12(1:ncomp,1:ncomp), ulong(1:ncomp),
     & vlong(1:ncomp), xmf(1:ncomp), tfac(1:ncomp) )
c
c    define number of properties
c
      nprop_per_comp = 8
      nprop = nprop_per_comp
      allocate (props(1:nprop_per_comp,1:6,0:ncomp) )
c
c    define number of molecules
c
      Ntotal = 125
      Nvec(1) = 100
      if (ncomp .eq. 2) then
            Nvec(2) = Ntotal - Nvec(1)
      endif
      Nvec(0) = sum(Nvec(1:ncomp))
      N = Nvec(0)
      allocate( r(1:N,1:3), v(1:N,1:3), a(1:N,1:3),
     & d3(1:N,1:3), d4(1:N,1:3), d5(1:N,1:3),
     & f(1:N,1:3), rwopbc(1:N,1:3), ntype(1:N) )
```

```
c
c     define mole fractions
c
              xmf(1:ncomp) = dfloat(Nvec(1:ncomp))/dfloat(N)
c
c     define maximum number of neighbors
c
              maxnbr = N*N/2
              allocate( Nnbrlist(1:maxnbr,1:2) )
c
c     define molecule types
c
              Nstop = 0
              Nstart = 1
              do i = 1, ncomp, 1
                            Nstop = Nstop + Nvec(i)
                            do j = Nstart, Nstop, 1
                                          ntype(j) = i
                            enddo
                            Nstart = Nstart + Nvec(i)
              enddo
c

c             Specify thermodynamic state
c
              T = 300.0d0               ! Temperature (K)
c             Vn = 4.052d+4             ! Angstroms cubed / molecule (gas at 298 K & 1 atm)
c             Vn = 2.020d+4             ! Angstroms cubed / molecule (gas at 150 K & 1 atm)
              Vn = 1.1323d+2 ! Ang^3/molecule (liq at 150 K & 1 atm)
c      Vn = 1.107d+2 ! Ang^3/molecule (liq at 150 K & 10 atm)
c      Vn = 8.8303d+1 ! Ang^3/molecule (liq at 150 K & 100 atm)
c      Vn = 8.8303d+1 ! Ang^3/molecule (liq at 150 K & 100 atm)
c
c             Specify Numerical Algorithm Parameters
c
              maxeqb = 10000 ! Number of time steps during equilibration
              maxstp = 50000 ! Number of time steps during data production
              dt = 2.0d0    ! size of time step (fs)
c
c             Specify pairwise potential parameters
c
              sigvec(1) = 3.822d0    ! collision diameter (Angstroms)
              epsvec(1) = 137.d0     ! well depth (K)
              MW(1) = 16.0420d0   ! molecular weight (grams/mole)
              if (ncomp .eq. 2) then
                            sigvec(2) = 4.418d0    ! collision diameter (Angstroms)
                            epsvec(2) = 230.d0     ! well depth (K)
                            MW(2) = 30.0680d0   ! molecular weight (grams/mole)
              endif
              rcut = 15.d0                          ! cut-off distance for potential (Angstroms)
c
c             Specify sampling intervals
c
              ksamp = 1                 ! sampling interval
              knbr = 10                 ! neighbor list update interval
              kwrite = 10000            ! writing interval
              kmsd = 1000                           ! position save for mean square displacement
              rnbr = rcut + 3.d0
c
c     Logical Variables
c
              lmsd = .true.                         ! logical variable for mean square displacement
              lscale = .true.                       ! logical variable for temperature scaling
              lselfd = .true.                       ! logical variable for self diffusion coefficient
              if (lselfd) lmsd = .true.
c
c             Character Variables
c
```

```
        cmsd = 'md_msd.out'
        cout = 'md_sum.out'
        open(unit=1,file=cout,form='formatted',status='unknown')
        if (lmsd) then
                open(unit=2,file=cmsd,form='formatted',status='unknown')
        endif
c
c       props
c       first index is property
c       property 1:  total kinetic energy
c       property 2:  total potential energy
c       property 3:  total energy
c       property 4:  temperature
c       property 5:  total x-momentum
c       property 6:  total y-momentum
c       property 7:  total z-momentum
c       property 8:  pressure
c
c       second index is
c       1:  instantaneous value
c       2:  sum
c       3:  sum of squares
c       4:  average
c       5:  variance
c       6:  standard deviation
c
        props(1:nprop_per_comp,1:6,0:ncomp) = 0.d0
c
c  Initialize vectors
c
c   first index of r is over molecules
c   second index of r is over dimensionality (x,y,z)
        r(1:N,1:3) = 0.d0               ! position
        v(1:N,1:3) = 0.d0               ! velocity
        a(1:N,1:3) = 0.d0               ! acceleration
        d3(1:N,1:3) = 0.d0              ! third derivative
        d4(1:N,1:3) = 0.d0              ! fourth derivative
        d5(1:N,1:3) = 0.d0              ! fifth derivative
        f(1:N,1:3) = 0.d0               ! force
        rwopbc(1:N,1:3) = 0.d0          ! position w/o pbc
c
c**********************************************************
c  INITIALIZATION PART TWO
c**********************************************************
c
c       compute a few parameters
c
        dt2 = dt*dt
        dt2h = 0.5d0*dt2
        Vol = dfloat(N)*Vn              ! total volume (Angstroms**3)
        side = Vol**(1.d0/3.d0)         ! length of side of simulation volume (Angstrom)
        sideh = 0.5d0*side              ! half of the side
        density = 1.d0/Vn               ! molar density
        print *, ' side ', side, ' rcut ', rcut, ' rnbr ', rnbr
c
c       Use Lorenz-Barthelot Mixing Rules for mixed interaction parameters
c
        do i = 1, ncomp, 1
                do j = 1, ncomp, 1
                        sig(i,j) = 0.5d0*(sigvec(i) + sigvec(j))
                        eps(i,j) = dsqrt(epsvec(i)*epsvec(j))
                enddo
        enddo
        sig6(1:ncomp,1:ncomp) = sig(1:ncomp,1:ncomp)**6.d0
        sig12(1:ncomp,1:ncomp) = sig(1:ncomp,1:ncomp)**12.d0
        rcut2 = rcut*rcut
        rnbr2 = rnbr*rnbr
c       stuff for long range energy correction
```

```
            rcut3 = rcut**3.d0
            rcut9 = rcut**9.d0
            kb = 1.380660d-5 ! Boltzmann's constant (aJ/molecule/K)
            eps(1:ncomp,1:ncomp) = eps(1:ncomp,1:ncomp)*kb
            pi = 2.d0*dasin(1.d0)
            ulong(1:ncomp) = 0.0
            vlong(1:ncomp) = 0.0
      do ic = 1, ncomp, 1
                  do jc = 1, ncomp, 1
                        ulongpre = dfloat(N)*8.d0*eps(ic,jc)*pi*density*xmf(jc)
                        ulong(ic) = ulong(ic) + ulongpre*(
     &                             sig12(ic,jc)/(9.d0*rcut9)
     &       - sig6(ic,jc)/(3.d0*rcut3) )
                        vlongpre = 96.d0*eps(ic,jc)*pi*density*xmf(jc)
                        vlong(ic) = vlong(ic) -vlongpre*(
     &         sig12(ic,jc)/(9.d0*rcut9)
     &        - sig6(ic,jc)/(6.d0*rcut3) )
                  enddo
            enddo
c           temperature factor for velocity scaling
            xNav = 6.0220d+23 ! Avogadro's Number
            convertMWtomass = 1.0d+28/(xNav*1000.d0)
            xmass(1:ncomp) = MW(1:ncomp)*convertMWtomass ! (1e-28*kg/molecule)
            xmassi(1:ncomp) = 1.d0/xmass(1:ncomp)
            tfac(1:ncomp) = 3.d0*float(Nvec(1:ncomp))*kb*T*xmassi(1:ncomp)  ! (Angstrom/fs)**2
c           correction factors for numerical algorithm
            dtv = dt;
            do i = 2, 5, 1
                  dtv(i) = dtv(i-1)*dt/dfloat(i)
            enddo
            alpha(0) = 3.d0/20.d0
            alpha(1) = 251.d0/360.d0
            alpha(2) = 1.d0
            alpha(3) = 11.d0/18.d0
            alpha(4) = 1.d0/6.d0
            alpha(5) = 1.d0/60.d0
            fact = 1.d0
            do i = 1, 5, 1
                  fact = fact*dfloat(i)
                  alpha(i) = alpha(i)*dt**(-dfloat(i))*fact
            enddo
            alpha = alpha*dt2*0.5d0
c
c           assign initial positions of  molecules in FCC crystal structure
c
            call funk_ipos(N, side, r, rwopbc)
c
c           assign initial velocities
c
            call funk_ivel(N,v,T,tfac,ncomp,Nvec)
c
c           create neighbor list
c
            call funk_mknbr(N,r,rnbr2,side,sideh, Nnbr, Nnbrlist, maxnbr)
            print *, ' initially we have ', Nnbr, ' neighbor pairs'
c
c           evaluate initial forces and potential energy
c
            call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,eps,
     & f, U, virial, maxnbr, ntype, ncomp)
            do i = 1, N, 1
                  a(i,1:3) = f(i,1:3)*xmassi(ntype(i))  ! initial acceleration
            enddo
            call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     & virial,ulong,vlong,ncomp,xmf,Nvec)
            write (6,1001) 0, props(1:4,1,0)
            write (1,1001) 0, props(1:4,1,0)
 1001 format(i7,' KE',e16.8,' PE',e16.8,' E',e16.8,' T',e14.7)
```

```
        props(1:nprop_per_comp,1:6,0:ncomp) = 0.d0
c***********************************************************
c  EQUILIBRATION
c***********************************************************
        do istep = 1, maxeqb, 1
c               predict new positions
                call predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
c               evaluate forces and potential energy
                call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,
   &    eps, f, U, virial, maxnbr, ntype, ncomp)
c               correct new positions
                call corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
c               apply periodic boundary conditions
                call pbc(N,r,side)
c               scale velocities
                if (lscale) then
                        call funk_scalev(N,v,T,tfac,ncomp,Nvec)
                endif
c               update neighbor list
                if (mod(istep,knbr) .eq. 0) then
                        call funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
                endif
c               sample properties
                if (mod(istep,ksamp) .eq. 0) then
                  call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
   &   virial,ulong,vlong,ncomp,xmf,Nvec)
                endif
c               write periodic results
                if (mod(istep,kwrite) .eq. 0) then
                        write (6,1001) istep, props(1:4,1,0)
                        write (1,1001) istep, props(1:4,1,0)
                endif
        enddo
c
c       write equilibration results
c
        if (maxeqb .gt. ksamp) then
                call funk_report(N,props(1:nprop_per_comp,1:6,0),
   & nprop,maxeqb,ksamp,'equilibration')
        endif
c***********************************************************
c  PRODUCTION
c***********************************************************
        props(1:nprop_per_comp,1:6,0:ncomp) = 0.d0
        lscale = .false.
        if (lmsd) then
                call funk_msd(N,rwopbc)
        endif
        do istep = 1, maxstp, 1
c               predict new positions
                call predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
c               evaluate forces and potential energy
                call funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,sig12,
   &    eps, f, U, virial, maxnbr, ntype, ncomp)
c               correct new positions
                call corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
c               apply periodic boundary conditions
                call pbc(N,r,side)
c               scale velocities
                if (lscale) then
                        call funk_scalev(N,v,T,tfac,ncomp,Nvec)
                endif
c               update neighbor list
                if (mod(istep,knbr) .eq. 0) then
                        call funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
                endif
c               sample properties
                if (mod(istep,ksamp) .eq. 0) then
```

```
                        call funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
   &      virial,ulong,vlong,ncomp,xmf,Nvec)
                      endif
c                     write periodic results
                      if (mod(istep,kwrite) .eq. 0) then
                              write (6,1001) istep, props(1:4,1,0)
                              write (1,1001) istep, props(1:4,1,0)
                      endif
c                     save positions for mean square displacement
                      if (lmsd) then
                              if (mod(istep,kmsd) .eq. 0) then
                                      call funk_msd(N,rwopbc)
                              endif
                      endif
              enddo

c
c             write equilibration results
c
              if (maxstp .gt. ksamp) then
                      call funk_report(N,props(1:nprop_per_comp,1:6,0),
   & nprop,maxstp,ksamp,'production   ')
              endif
c
              ttot = ETIME(TA)
              write(*,*) 'Program has used', ttot, 'seconds of CPU time.'
              write(*,*) '  This includes', TA(1), 'seconds of user time and',
   & TA(2), 'seconds of system time.'
              write(1,*) 'Program has used', ttot, 'seconds of CPU time.'
              write(1,*) '  This includes', TA(1), 'seconds of user time and',
   & TA(2), 'seconds of system time.'
c
              close(unit=1,status='keep')
              if (lmsd) then
                      close(unit=2,status='keep')
              endif
c
c    compute self-diffusivities
c
              if (lselfd) then
                      call getd_multi_sub(maxstp, kmsd, N, dt, ncomp, Nvec, cmsd)
              endif
c
              stop
              end




c**********************************************************
c  SUBROUTINES
c**********************************************************

c
c  funk_ipos:  assigns initial positions
c
              subroutine funk_ipos(N,side,r,rwopbc)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N
              double precision, intent(in) :: side
              double precision, intent(out), dimension(1:N,1:3) :: r, rwopbc
              xi = dfloat(N)**(1.d0/3.d0)
              ni = int(xi)
              if (xi - dfloat(ni) .gt. 1.d-14) then
                      ni = ni + 1
              endif
              print *, 'funkipos: N = ', N, ' ni = ', ni
```

```
        ncount = 0
        dx = side/dfloat(ni)
        do ix = 1, ni, 1
                do iy = 1, ni, 1
                        do iz = 1, ni, 1
                                ncount = ncount + 1
                                if (ncount .le. N) then
                                        r(ncount,1) = dx*dfloat(ix)
                                        r(ncount,2) = dx*dfloat(iy)
                                        r(ncount,3) = dx*dfloat(iz)
                                endif
                        enddo
                enddo
        enddo
        rwopbc = r
        return
        end


c
c  funk_ivel:  assigns initial velocities
c
        subroutine funk_ivel(N,v,T,tfac,ncomp,Nvec)
        implicit double precision (a-h, o-z)
        integer, intent(in) :: N, ncomp
        double precision, intent(in) :: T
        double precision, intent(in), dimension(1:ncomp) :: tfac
        integer, intent(in), dimension(0:ncomp) :: Nvec
        double precision, intent(out), dimension(1:N,1:3) :: v
        double precision, dimension(1:3) :: sumv
c
        call random_number(v)           ! random velocities from 0 to 1
        v = 2.d0*v - 1.d0               ! random velocities from -1 to 1
c       enforce zero net momentum for each component
        do k = 1, ncomp, 1
                if (k .eq. 1) then
                        jstart = 1
                else
                        jstart = Nvec(k-1) + 1
                endif
                jend = jstart + Nvec(k) - 1
                do i = 1, 3, 1
                        sumv(i) = 0.d0
                        do j = jstart, jend, 1
                                sumv(i) = sumv(i) + v(j,i)
                        enddo
                        do j = jstart, jend, 1
                                v(j,i) = v(j,i) - sumv(i)/dfloat(Nvec(k))
                        enddo
                enddo
c       scale initial velocities to set point temperature
                sumvsq = sum(sum(v(jstart:jend,1:3)*v(jstart:jend,1:3),1) )
                fac = dsqrt(tfac(k)/sumvsq)
                v(jstart:jend,1:3) = v(jstart:jend,1:3)*fac
        enddo
        return
        end



c
c  funk_mknbr:  create neighbor list
c
        subroutine funk_mknbr(N,r,rnbr2,side,sideh,Nnbr,Nnbrlist,maxnbr)
        implicit double precision (a-h, o-z)
        integer, intent(in) :: N, maxnbr
        double precision, intent(in) :: side, sideh, rnbr2
        double precision, intent(in), dimension(1:N,1:3) :: r
```

```
      integer, intent(out) :: Nnbr
      integer, intent(out), dimension(1:maxnbr,1:2) :: Nnbrlist
      double precision, dimension(1:3) :: dis
      Nnbr = 0
      do i = 1, N, 1
            do j = i+1, N, 1
                  dis(1:3) = r(i,1:3) - r(j,1:3)
                  do k = 1, 3, 1
                        if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
                  if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
                  enddo
                  dis2 = sum(dis*dis)
                  if (dis2 .le. rnbr2) then
                        Nnbr = Nnbr + 1
                        Nnbrlist(Nnbr,1) = i
                        Nnbrlist(Nnbr,2) = j
                  endif
            enddo
      enddo
      return
      end



c
c  funk_force:  evaluate forces
c
      subroutine funk_force(N,r,rcut2,side,sideh,Nnbr,Nnbrlist,sig6,
     & sig12, eps, f, U, virial, maxnbr, ntype, ncomp)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N, maxnbr, Nnbr, ncomp
      double precision, intent(in) :: side, sideh, rcut2
      double precision, intent(in), dimension(1:ncomp,1:ncomp) :: sig6,
     & sig12, eps
      double precision, intent(in), dimension(1:N,1:3) :: r
      integer, intent(in), dimension(1:maxnbr,1:2) :: Nnbrlist
      double precision, intent(out), dimension(1:N,1:3) :: f
      double precision, intent(out) :: U, virial
      integer, intent(in), dimension(1:N) :: ntype
      double precision, dimension(1:3) :: dis
      f = 0.d0     ! forces
      U = 0.d0     ! potential energy
      virial = 0.d0 ! virial coefficient
      do m = 1, Nnbr, 1
            i = Nnbrlist(m,1)
            j = Nnbrlist(m,2)
            dis(1:3) = r(i,1:3) - r(j,1:3)
            do k = 1, 3, 1
                  if (dis(k) .gt.  sideh) dis(k) = dis(k) - side
                  if (dis(k) .lt. -sideh) dis(k) = dis(k) + side
            enddo
            dis2 = sum(dis*dis)
            if (dis2 .le. rcut2) then
                  dis2i = 1.d0/dis2
                  dis6i = dis2i*dis2i*dis2i
                  dis12i = dis6i*dis6i
                  sig12t = sig12(ntype(i),ntype(j))
                  sig6t = sig6(ntype(i),ntype(j))
                  epst = eps(ntype(i),ntype(j))
                  U = U + epst*( sig12t*dis12i - sig6t*dis6i )
                  fterm = epst*(2.d0*sig12t*dis12i - sig6t*dis6i )*dis2i
                  f(i,1:3) = f(i,1:3) + fterm*dis(1:3)
                  f(j,1:3) = f(j,1:3) - fterm*dis(1:3)
                  virial = virial - fterm*dis2
            endif
      enddo
      f = f*24.d0
      U = U*4.d0
```

```
              virial=virial*24.d0
              return
              end




c
c predict new positions
c
              subroutine predictor(N,r,rwopbc,v,a,d3,d4,d5,dtv)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N
              double precision, intent(inout), dimension(1:N,1:3) ::
     & r,rwopbc, v, a, d3, d4, d5
      double precision, intent(in), dimension(1:5) :: dtv
              rwopbc = rwopbc + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 +
     &                                    dtv(5)*d5
              r      = r      + v *dtv(1) + dtv(2)*a  + dtv(3)*d3 + dtv(4)*d4 +
     &                                    dtv(5)*d5
              v      = v      + a *dtv(1) + dtv(2)*d3 + dtv(3)*d4 + dtv(4)*d5
              a      = a      + d3*dtv(1) + dtv(2)*d4 + dtv(3)*d5
              d3     = d3     + d4*dtv(1) + dtv(2)*d5
              d4     = d4     + d5*dtv(1)
              return
              end
c
c correct new positions
c
              subroutine corrector(N,r,rwopbc,v,a,d3,d4,d5,f,dt2h,alpha,xmassi)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N
              double precision, intent(inout), dimension(1:N,1:3) ::
     & r,rwopbc, v, a, d3, d4, d5
              double precision, intent(in), dimension(1:N,1:3) :: f
              double precision, intent(in) :: dt2h, xmassi
      double precision, intent(in), dimension(0:5) :: alpha
              double precision, dimension(1:3) :: errvec
              do i = 1, N, 1
                     errvec(1:3) = ( f(i,1:3)*xmassi - a(i,1:3) )
                     rwopbc(i,1:3) = rwopbc(i,1:3) + errvec(1:3)*alpha(0)
                     r(i,1:3)  = r(i,1:3)  + errvec(1:3)*alpha(0)
                     v(i,1:3)  = v(i,1:3)  + errvec(1:3)*alpha(1)
                     a(i,1:3)  = a(i,1:3)  + errvec(1:3)*alpha(2)
                     d3(i,1:3) = d3(i,1:3) + errvec(1:3)*alpha(3)
                     d4(i,1:3) = d4(i,1:3) + errvec(1:3)*alpha(4)
                     d5(i,1:3) = d5(i,1:3) + errvec(1:3)*alpha(5)
              enddo
              return
              end




c
c apply periodic boundary conditions
c
              subroutine pbc(N,r,side)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N
              double precision, intent(inout), dimension(1:N,1:3) :: r
              double precision, intent(in) :: side
              do i = 1, N, 1
                     do j = 1, 3, 1
                            if (r(i,j) .gt.  side) r(i,j) = r(i,j) - side
                            if (r(i,j) .lt.  0.0)  r(i,j) = r(i,j) + side
                     enddo
              enddo
              return
              end



c
c  funk_scalev:  scale velocities
c
              subroutine funk_scalev(N,v,T,tfac,ncomp,Nvec)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N, ncomp
              double precision, intent(inout), dimension(1:N,1:3) :: v
              double precision, intent(in) :: T
              double precision, intent(in), dimension(1:ncomp) :: tfac
              integer, intent(in), dimension(0:ncomp) :: Nvec
              double precision, dimension(1:3) :: sumv
c     scale velocities to set point temperature
              do k = 1, ncomp, 1
                     if (k .eq. 1) then
                            jstart = 1
                     else
                            jstart = Nvec(k-1) + 1
                     endif
                     jend = jstart + Nvec(k) - 1
                     sumvsq = sum(sum(v(jstart:jend,1:3)*v(jstart:jend,1:3),1) )
                     fac = dsqrt(tfac(k)/sumvsq)
                     v(jstart:jend,1:3) = v(jstart:jend,1:3)*fac
              enddo
              return
              end


c
c calculate properties for sampling
c
              subroutine funk_getprops(N,v,xmass,T,kb,U,props,nprop,density,
     & virial,ulong,vlong,ncomp,xmf,Nvec)
              implicit double precision (a-h, o-z)
              integer, intent(in) :: N, nprop
              double precision, intent(in), dimension(1:N,1:3) :: v
              double precision, intent(in) :: T, kb, U, density, virial
              double precision, intent(in), dimension(1:ncomp) :: ulong, vlong,
     & xmf, xmass
              integer, intent(in), dimension(0:ncomp) :: Nvec
              double precision, intent(inout), dimension(1:nprop,1:6,0:ncomp) ::
     & props
c     props
c     first index is property
c     property 1:  total kinetic energy
c     property 2:  total potential energy
c     property 3:  total energy
c     property 4:  temperature
c     property 5:  total x-momentum
c     property 6:  total y-momentum
c     property 7:  total z-momentum
c   property 8:  pressure
c
c     second index is
c     1:  instantaneous value
c     2:  sum
c     3:  sum of squares
c     4:  average
c     5:  variance
c     6:  standard deviation
c
c   third index is for component, 0 for total
c
              sumvsq = 0.d0
              summvx = 0.d0
```

```
            summvy = 0.d0
            summvz = 0.d0
            do k = 1, ncomp, 1
                  xmasst = xmass(k)
                  if (k .eq. 1) then
                        jstart = 1
                  else
                        jstart = Nvec(k-1) + 1
                  endif
                  jend = jstart + Nvec(k) - 1
                  do j = jstart, jend, 1
                        sumvsq = sumvsq + xmasst*sum(v(j,1:3)*v(j,1:3))
                        summvx = summvx + xmasst*v(j,1)
                        summvy = summvy + xmasst*v(j,2)
                        summvz = summvz + xmasst*v(j,3)
                  enddo
            enddo
            xKE = 0.5d0*sumvsq ! (aJ)
            Ti = 2.d0/(3.d0*dfloat(N)*kb)*xKE
c
c           get instantaneous values
c
            props(1,1,0) = xKE
            props(2,1,0) = U + sum(xmf(1:ncomp)*ulong(1:ncomp))
            props(3,1,0) = xKE + props(2,1,0)
            props(4,1,0) = Ti
            props(5,1,0) = summvx
            props(6,1,0) = summvy
            props(7,1,0) = summvz
            vlongt = sum(xmf(1:ncomp)*vlong(1:ncomp))
            props(8,1,0) = density*(kb*Ti - virial/(3.D0*dfloat(N))
     & -vlongt/3.d0)
c
c     get the cumulative sum and the cumulative sum of the squares
c
            props(1:nprop,2,:) = props(1:nprop,2,:) + props(1:nprop,1,:)
            props(1:nprop,3,:) = props(1:nprop,3,:) +
     &          props(1:nprop,1,:)*props(1:nprop,1,:)
            return
            end

c
c calculate and report simulation statistics
c
            subroutine funk_report(N,props,nprop,maxeqb,ksamp,csect)
            implicit double precision (a-h, o-z)
            integer, intent(in) :: N, nprop, maxeqb, ksamp
            double precision, intent(inout), dimension(1:nprop,1:6) :: props
            character*13, intent(in) :: csect
            character*22, dimension(1:nprop) :: propname
            den = dfloat(maxeqb/ksamp)
            props(1:nprop,4) = props(1:nprop,2)/den
            props(1:nprop,5) = props(1:nprop,3)/den - props(1:nprop,4)**2.d0
            do i = 1, nprop, 1
                  if (props(i,5) .gt. 0.d0) then
                        props(i,6) = dsqrt(props(i,5))
                  else
                        props(i,6) = 0.d0
                  endif
            enddo
            propname(1) = 'Kinetic Energy (aJ)   '
            propname(2) = 'Potential Energy (aJ) '
            propname(3) = 'Total Energy (aJ)     '
            propname(4) = 'Temperature (K)       '
            propname(5) = 'x-Momentum            '
            propname(6) = 'y-Momentum            '
            propname(7) = 'z-Momentum            '
            propname(8) = 'Pressure aJ/Angstrom^3'
```

```
      write(6,1002) csect
      write(1,1002) csect
 1002 format ('********** ', a22 ' Completed **********')
      write(6,1003)
      write(1,1003)
 1003 format ('property              instant       average       s',
     & 'tandard deviation')
      do i = 1, nprop, 1
         write(6,1004) propname(i), props(i,1),props(i,4),props(i,6)
         write(1,1004) propname(i), props(i,1),props(i,4),props(i,6)
      enddo
 1004 format(a22,3(1x,e16.8))
      return
      end
c
c  save positions for mean square displacement calculations
c
      subroutine funk_msd(N,rwopbc)
      implicit double precision (a-h, o-z)
      integer, intent(in) :: N
      double precision, intent(in), dimension(1:N,1:3) :: rwopbc
      do i = 1, N, 1
         write(2,1005) rwopbc(i,1:3)
      enddo
 1005 format(3(e16.8,1x))
      return
      end

      subroutine getd_multi_sub(maxstp, kmsd, N, dt, ncomp, Nvec, cmsd)
c
c     This program will calculate diffusivities
c  from mean square displacement data
c  for multicomponent systems
c
c     author  David Keffer
c     Department of Chemical Engineering
c     University of Tennessee, Knoxville
c     last updated  January 9, 2002
c
      integer, intent(in) :: maxstp, kmsd, N, ncomp
      integer, intent(in), dimension(0:ncomp) :: Nvec
      double precision, intent(in) :: dt
      character*12, intent(in) :: cmsd
      integer, dimension(1:ncomp) :: Nstart, Nend
      character*12 :: cout                  ! character variables
      character*3, dimension(1:4) :: cname
      double precision, dimension(1:4) :: Dav, Dsd
      double precision, dimension(1:3) :: slope,slopesd,yinter,yintersd
      double precision, allocatable :: md_msd(:,:), time_vec(:),
     & xmsd(:,:,:)
c
      do i = 1, ncomp, 1
         if (i .eq. 1) then
            Nstart(i) = 1
         else
            Nstart(i) = Nend(i-1) + 1
         endif
         Nend(i) = Nstart(i) + Nvec(i) - 1
      enddo
c
      cout = 'get_diff.out'
c     number of times represented in data
      ntime = maxstp/kmsd + 1
c     number of rows of data
      ndata = N*ntime
      allocate (md_msd(1:ndata,1:3))
      open(unit=1,file=cout,form='formatted',status='unknown')
      open(unit=2,file=cmsd,form='formatted',status='old')
```

```
            print *, ' ntime = ', ntime, ' ndata = ', ndata
            do i = 1, ndata, 1
                  read(2,*) md_msd(i,1:3)
            enddo
            print *, ' read all the data'
c           number of origins is half number of time steps
            norigin = (ntime-1)/2
c           minimum number of intervals to contribute to diffusivity
            nmin = (ntime-1)/4
c           maximum number of intervals to contribute to diffusivity
            nmax = norigin
c
            if (nmin .gt. nmax) then
                  print *, ' We have a problem. '
                  print *, ' nmin = ', nmin, ' nmax = ', nmax
                  stop
            endif
c           store mean square displacements in xmsd
            allocate (time_vec(1:norigin), xmsd(1:norigin,1:3,1:ncomp))
            do i = 1, norigin, 1
                  time_vec(i) = dfloat(i*kmsd)*dt
            enddo
            xmsd = 0.d0
            do ic = 1, ncomp, 1
                  do i = Nstart(ic), Nend(ic), 1
                        do j = 1, norigin, 1
                              jstart = (j-1)*N + i
                              do k = nmin, nmax, 1
                                    kend = jstart + k*N
                                    xmsd(k,1:3,ic) = xmsd(k,1:3,ic) +
     &                        (md_msd(kend,1:3) - md_msd(jstart,1:3) )**2
                              enddo
                        enddo
                  enddo
            enddo
            do ic = 1, ncomp, 1
                  fact = 1.d0/dfloat(Nvec(ic)*norigin)
                  xmsd(:,:,ic) = xmsd(:,:,ic)*fact
            enddo
c
            do ic = 1, ncomp, 1
c
c                 perform a linear least squares regression
c
                  do i = 1, 3, 1
                        call dllsr(slope(i), slopesd(i), yinter(i), yintersd(i),
     &            nmax-nmin+1, time_vec(nmin:nmax), xmsd(nmin:nmax,i,ic))
                  enddo
c
c                 report results
c
                  cname(1) = 'x  '
                  cname(2) = 'y  '
                  cname(3) = 'z  '
                  cname(4) = 'avg'
                  write(6,*) 'Component ', ic
                  write(1,*) 'Component ', ic
                  do i = 1, 3, 1
                        write(6,1007) cname(i), slope(i), yinter(i)
                        write(1,1007) cname(i), slope(i), yinter(i)
                  enddo
 1007 format(a3, ' slope = ', e16.8, ' y-intercept = ', e16.8,
     & ' A^2/fs')
                  Dav(1:3) = 0.5d0*slope(1:3)*1.0d-5 ! convert to m^2/sec
                  Dsd(1:3) = 0.5d0*slopesd(1:3)*1.0d-5 ! convert to m^2/sec
                  Dav(4) = sum(Dav(1:3))/3.d0
c     standard deviation of average diffusivity
                  term1 = 3.d0*(Dav(1)*Dav(1) + Dav(2)*Dav(2) + Dav(3)*Dav(3) )
```

```
                Dsd(4) = sqrt( (term1 - Dav(4)*Dav(4)*9.d0) /6.d0 )
                do i = 1, 4, 1
                        write(6,1006) cname(i), Dav(i), Dsd(i)
                        write(1,1006) cname(i), Dav(i), Dsd(i)
                enddo
 1006 format(a3, ' diffusivity avg = ', e16.8, ' stand dev = ', e16.8,
     & ' m^2/sec ')
c
c       write xmsd vs time data for later plotting
c
                do i = 1, norigin, 1
                        write(1,1008) time_vec(i), xmsd(i,1:3, ic)
                enddo
 1008 format(4(e16.8,1x))
        enddo
        close (unit=1,status='keep')
        close (unit=2,status='keep')
        return
        end


        subroutine dllsr(slope, slopesd, yinter, yintersd, n, x, y)
        implicit double precision (a-h, o-z)
        double precision, intent(out) :: slope, slopesd, yinter, yintersd
        integer, intent(in) :: n
        double precision, intent(in), dimension(1:n) :: x, y
        xn = dfloat(n)
        xavg = sum(x)/xn
        yavg = sum(y)/xn
        sumxy = 0.d0
        sumxx = 0.d0
        sumx2 = 0.d0
        do i = 1, n, 1
                sumxy = sumxy + (x(i) - xavg)*(y(i) - yavg)
                sumxx = sumxx + (x(i) - xavg)*(x(i) - xavg)
                sumx2 = sumx2 + x(i)*x(i)
        enddo
        slope = sumxy/sumxx
        yinter = yavg - slope*xavg
        sse = 0.d0
        do i = 1, n, 1
                sse = sse + (y(i) - slope*x(i) - yinter)**2.d0
        enddo
        sig2 = sse/dfloat(n-2)
        slopesd = dsqrt(sig2/sumxx)
        yintersd = dsqrt(sig2/dfloat(n)*sumx2/sumxx)
        return
        end
```