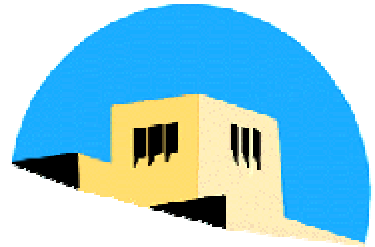


Workshop On Scientific Problem Solving in Fortran 90/95 - Applications I : Integration and MD

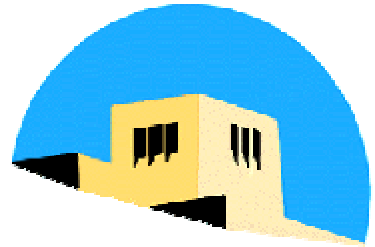
Spring, 1999

Richard C. Allen, SNL

Paul M. Alsing, UNM/AHPCC

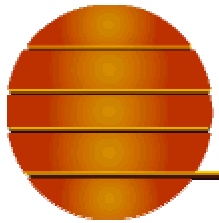
Andrew C. Pineda, UNM/AHPCC

Brian T. Smith, UNM/AHPCC/CS

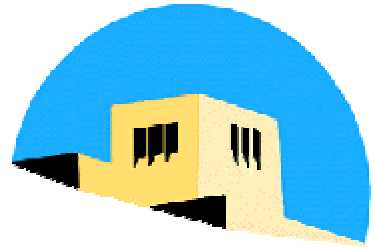


Topics To Be Covered

- Last 4 workshops of this series
  - described new language features added since Fortran 77
    - Examples: array syntax, pointers, derived types, precision control
  - described Fortran 90/95 syntax
- Next 4 workshops of this series
  - illustrate these new features in “real” scientific applications

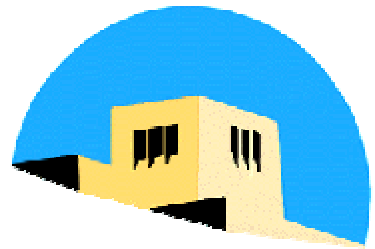


ALBUQUERQUE
High Performance Computing Center



Remaining Workshops

Applications	Fortran 90/95 Features Illustrated
Integration and Molecular Dynamics	Optional arguments, derived types (structures), array syntax and efficient use of pointers
Multigrid	Recursion, pointers
Eigenvalues/Eigenvectors in QM	Arrays, External libraries
FFT	?

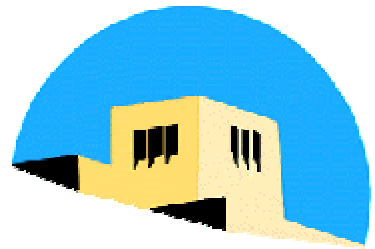


Applications To Be Covered

- VECGAUSS - Gaussian Quadrature Example
  - History
    - ACP needed to integrate a large set of functions which all depended on a set of functions (spherical Bessel functions) that were most easily obtained by a recursive algorithm that provided the set all at once.

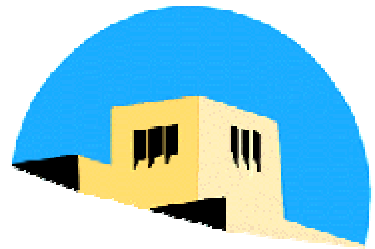
$$j_n(x), h_n(x), \forall n \in (0,100)$$

    - Original Fortran 77 code based on a single function Gaussian integrator used with much success in our research group.
    - Ported to Fortran 90 for instructional purposes.



Applications To Be Covered

- VECGAUSS - Gaussian Quadrature
  - F77 version features
    - Required user to provide work arrays, arrays of error tolerances, smallest domain over which integration region was to be sub-divided.
    - Swapped work arrays using indexed multi-dimensional arrays.
    - Required tabulated abscissas and weights, precision fixed at double precision.
  - F90 version features
    - Need for work arrays eliminated, other arguments made optional.
    - Work arrays swapped with pointers.
    - Abscissas and weights are generated by function call. Precision set in parameter statement.
    - Module

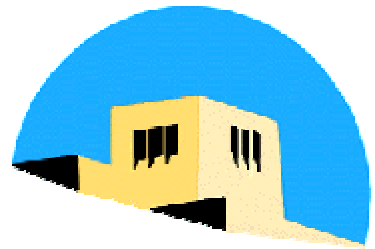


Applications To Be Covered

- LJ - A 3D Lennard-Jones Molecular Dynamics Simulation
  - History
    - Lennard-Jones potential - one of the first “realistic” potentials used in atomic simulation. Very successful in predicting properties of noble gasses such as Ar.

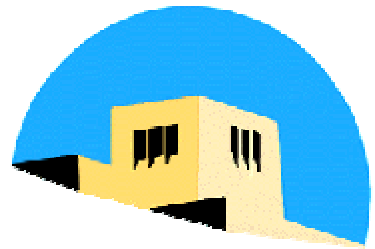
$$U(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]$$

    - Original Fortran 77 version authored by Steve Plimpton and Laura Frink of Sandia National Laboratories. Used for CS 471 class in Fall 1997.
    - Ported to Fortran 90 by ACP.



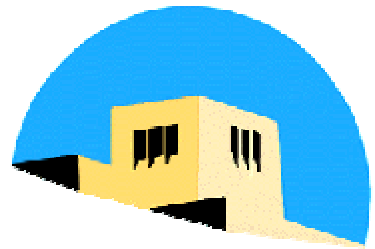
Applications To Be Covered

- LJ - A 3D Lennard-Jones Molecular Dynamics Simulation
  - Fortran 77 Version Features
    - Particle properties (position, velocity, force/acceleration) spread across several arrays.
    - Does not implement features such as cell or neighbor lists that are used to speed calculations for large numbers of particles.
    - Doesn't use F77 intrinsic functions to simplify code.
    - Ergodic hypothesis (time-averaged properties = ensemble-averaged properties) allow computation of thermodynamic properties such as temperature, pressure.



Applications To Be Covered

- LJ - A 3D Lennard-Jones Molecular Dynamics Simulation
  - Fortran 90 Version Features.
    - Particle properties (position, velocity, force/acceleration) stored in derived types (structures).
    - Array syntax - simplifies code and removes dimension of space from most of code
    - F90 intrinsics used to simplify code
    - Implements cell lists and neighbor lists to speed calculations for large numbers of particles. These are implemented with structures.
      - Cell lists - uses a doubly linked list built into the particle structure
      - Neighbor lists - implemented as a singly linked list attached to each particle



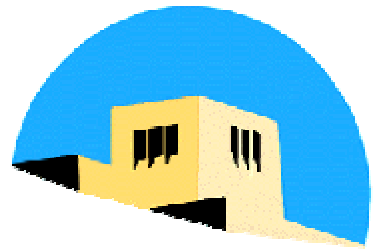
Where to get today's codes

- From the Web, go to main page

  http://www.arc.unm.edu/~bsmith/ScPrSolving/f90workshop.html

  - Navigate to codes page, you will find links to directories for f77 and f90 versions of each code.
- Copy the directory tree directly (if you have an AHPCC account)

  ~bsmith/public_html/ScPrSolving/Codes/Source/md_and_integration/

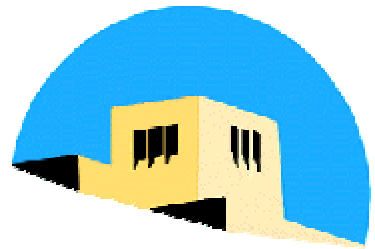


VECGAUSS: Theory/Algorithm

- Find the integral

$$\int_a^b f(x)w(x)dx$$

- The function *w(x)* is introduced to handle infinite intervals (*a*,*b*) and functions *f(x)* that have singularities in the interval of integration.

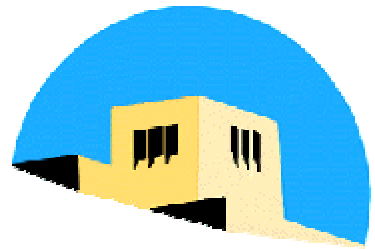


Suppose we know *f(x)* at *N* distinct points $\{x_1, x_2, \ldots, x_N\}$ and consider an interpolating polynomial that approximates *f(x)* at these points

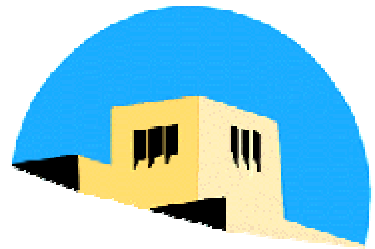
- the interpolating polynomial is never formed explicitly but integrals of the polynomials are determined and yield various rules (Simpson's, trapezoidal, midpoint, …)

$$\int_a^b f(x)w(x)dx \approx \int_a^b \sum_{i=1}^{N} f(x_i)L_i(x)w(x)dx =$$

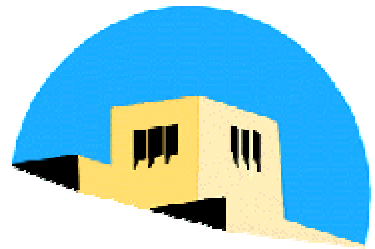
$$\sum_{i=1}^{N} f(x_i)\int_a^b L_i(x)w(x)dx = \sum_{i=1}^{N} A_i f(x_i)$$



- The A_i are called the weights
- The formula $\sum_{i=1}^{N} A_i f(x_i)$ is called the quadrature rule
- The rule integrate exactly all polynomials *f(x)* up to a certain degree, say degree *d*
  - *d* is called the degree of precision of the rule
  - a careful selection of the points x_i can lead to a rule whose degree of precision exceeds *N-1*, *Gaussian rules have d=2N-1*



- The modern theory of Gaussian integration for arbitrary *w(x)*, is largely due to Jacobi and Christoffel, who used the theory of orthogonal polynomials.
- They showed that the x_i for an *N*-point Gaussian rule with weight function *w(x)* are the zeros of the *N*th orthogonal polynomial determined by *w(x)* and the interval.



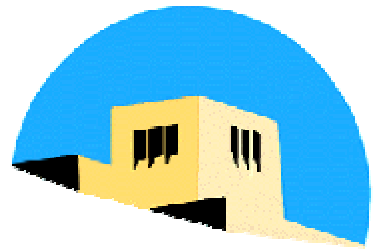
- The weights, A_i, corresponding to the x_i have a nice representation:

$$A_i = \frac{\left\langle p_{N-1} \middle| p_{N-1} \right\rangle}{p_{N-1}(x_i)\, p'_N(x_i)}$$

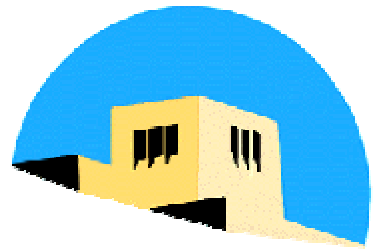
- We will use the Gauss-Legendre rules,

$$w(x) = 1, -1 < x < 1, \; p_N(x) = P_N(x)$$

- The even rules are symmetric about 0. So store only x>0. These are packed into arrays t, w in vecgauss for rules 2, 4, 8, 16, 32, 64, 128, 256. Req's and array of 255 elements.



- VECGAUSS uses the even power of 2 Gauss-Legendre (2 to 256 point) fixed rules to construct an adaptive integrator.
  - Adaptive schemes choose the points at which *f(x)* is evaluated depending on the behavior of *f(x)*.
  - In an adaptive scheme, the user inputs a desired accuracy (relative & absolute error).
  - A set of fixed rules is then applied to the integration range.
  - The accuracy is tested by comparing the results of the rules. If the test fails, the integration range is split (typically in half) and the rules applied to each piece. This process is repeated until convergence, or until the range gets too small.

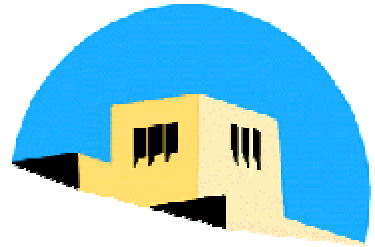


Changes to the argument list

F77: subroutine vecgauss(fcn, a, b, ans,work,n, aer,rer,del, ier, flag, nstart)

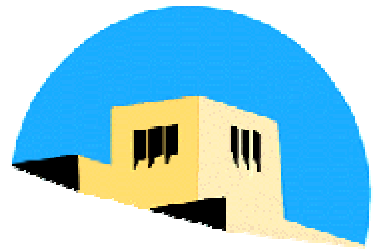
F90: subroutine vecgauss(fcn, a, b, ans, n, ier, aer, rer, del, nstart, rer_fudge, nfunccalls)

- real*8 work(n,4) removed; real(WP), dimension(size(ans)), target :: work1, work2 allocated on stack, work3, work4 not targets.
- logical flag(n) used to track convergence now allocated on stack too.
- Red items - optional in f90 version. Default values set using F90 intrinsics.
- Blue items - new optional variables (input/output).



Optional Arguments Example

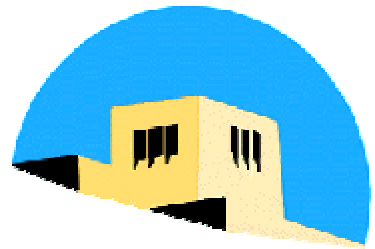
```
real(WP), dimension(:), target, intent(inout), optional :: aer
real(WP), dimension(size(ans)), target :: aerlocal
real(WP), dimension(:), pointer :: ptr_aer
if(present(aer) then
      ptr_aer=>aer
else
      aerlocal=0.0e0_WP
      ptr_aer=>aer
endif
```



Use of pointers

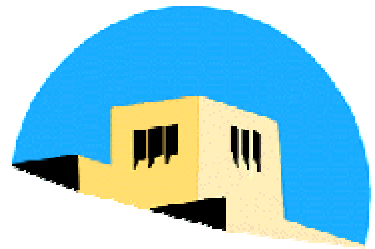
```
pwork1=>work1
pwork2=>work2

do  i=nterms,2*nterms-1,1
  call fcn(c1+c2*t(i),work3,flag)
  call fcn(c1-c2*t(i),work4,flag)
  where (.not.flag(1:nlocal))
    pwork1(1:nlocal)=pwork1(1:nlocal)+ &
              w(i)*(work3(1:nlocal)+work4(1:nlocal))
  end where
end do
```

```
if(nterms.lt.ntmax)then
    pswap=>pwork1
    pwork1=>pwork2
    pwork2=>pswap
    nterms=nterms*2
  else
    ! chop interval
    exit ! Exit 1 level of do loops
  endif
```



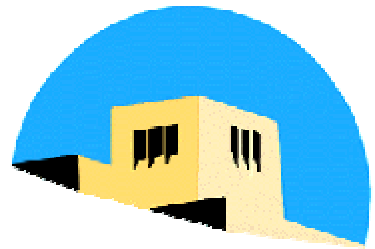
Testing for convergence using where

```
newflag(1:nlocal)=.FALSE.
where (.not.flag(1:nlocal))
  pwork1(1:nlocal)=pwork1(1:nlocal)*c2
  newflag(1:nlocal)=abs(pwork1(1:nlocal)-pwork2(1:nlocal)) &
      .le.(ptr_aer(1:nlocal)+ &
      ptr_rer(1:nlocal)*abs(pwork1(1:nlocal)))
end where
where(newflag(1:nlocal))
  flag(1:nlocal)=newflag(1:nlocal)
end where
 ...
conv=all(flag(1:nlocal)) ! See if we are done on this interval
```



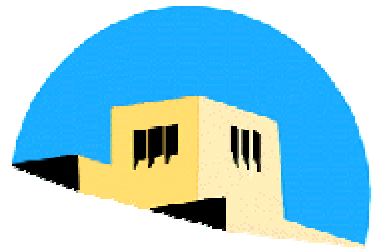
Molecular Dynamics

- Simulation of the motion of molecules to gain an understanding of their interaction
  - chemical reactions
  - fluid flow
  - phase transitions
  - …..



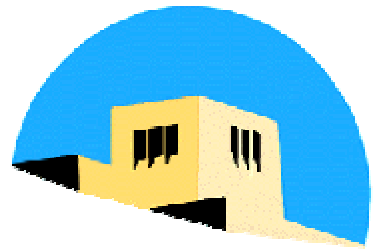
MD Introduction

- Uses classical Newtonian mechanics
  - forces depend on distance between objects
  - conservation of energy
- Utilizes the speed of computers and their ability to repeat simple computations millions of times
- An early use of computers and continues to consume many cycles
  - initially 100's of molecules simulated
  - now millions of cycles, many time steps (10^5)



MD Introduction

- Discrete models for potentials and forces
  - attractive and repulsive forces
  - N particles yield in general $O(N^2)$ operations
  - large simulations can take advantage of forces tailing off with larger distances (beyond a cutoff, forces are assumed to be zero) - corrections need to be added to conserve energy though.



Relationship between MD simulation and thermodynamics

- Let A be some thermodynamic property, <*A*> be the time average:

$$< A >= \lim_{t\to\infty} \frac{1}{t} \int_0^t dt' A(t') = \lim_{t\to\infty} \frac{1}{t} \int_0^t dt' A(r^N(t'), p^N(t'))$$

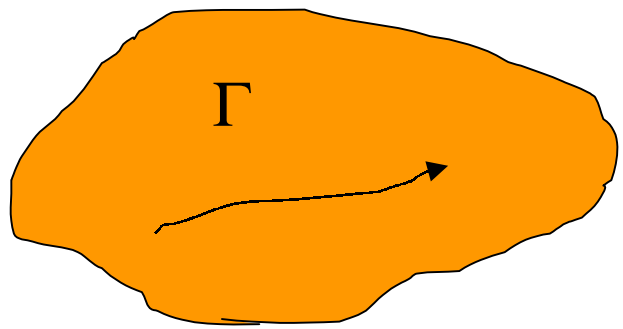
where $\{r^N(0), p^N(0)\}$ are arbitrary initial conditions consistent with N,V,E.

- Ergodic hypothesis, *not true in general*, says the above is equivalent to a microcanonical phase space average (constant N,V,E).

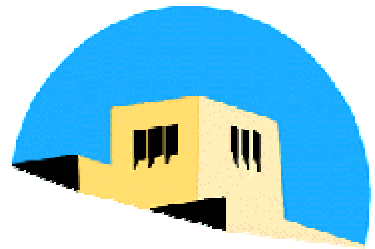
$$\langle A \rangle = \int dr^N dp^N A(r^N, p^N) p_E(r^N, p^N), p_E(r^N, p^N) \propto \delta(H(r^N, p^N) - E)$$

Ergodic hypothesis

- The idea behind the ergodic hypothesis is that trajectories in phase space Γ pass by every point in phase space if you wait long enough.

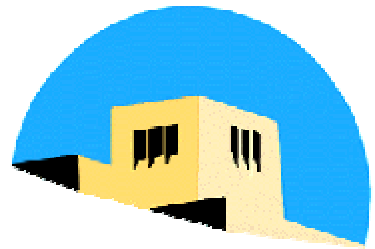


- The time they spend in a region is proportional to its ensemble weight.



Molecular Dynamics Method

- What do we need to do?
  - Need to integrate equations of motion for N particles.
  - Need an integration scheme that allows us to compute positions and velocities at a sequence of time steps. Since each iteration requires the computation of the forces on all particles, which is an $O(N^2)$ problem, we want to make Δt as large as possible.



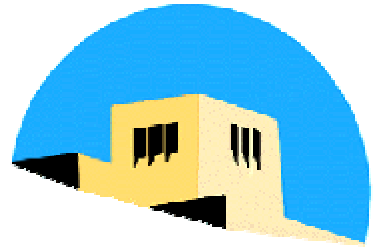
MD Integration Method

- Want a scheme that conserves energy and is time-reversible.
- Some of the oldest and best methods for this is the Verlet algorithm, and its variants. LJ code uses the "velocity-Verlet" algorithm.

$$v(t+\frac{1}{2}\Delta t) \approx v(t) + \frac{1}{2}\Delta t a(t)$$

$$r(t+\Delta t) \approx r(t) + \Delta t v(t+\frac{1}{2}\Delta t)$$

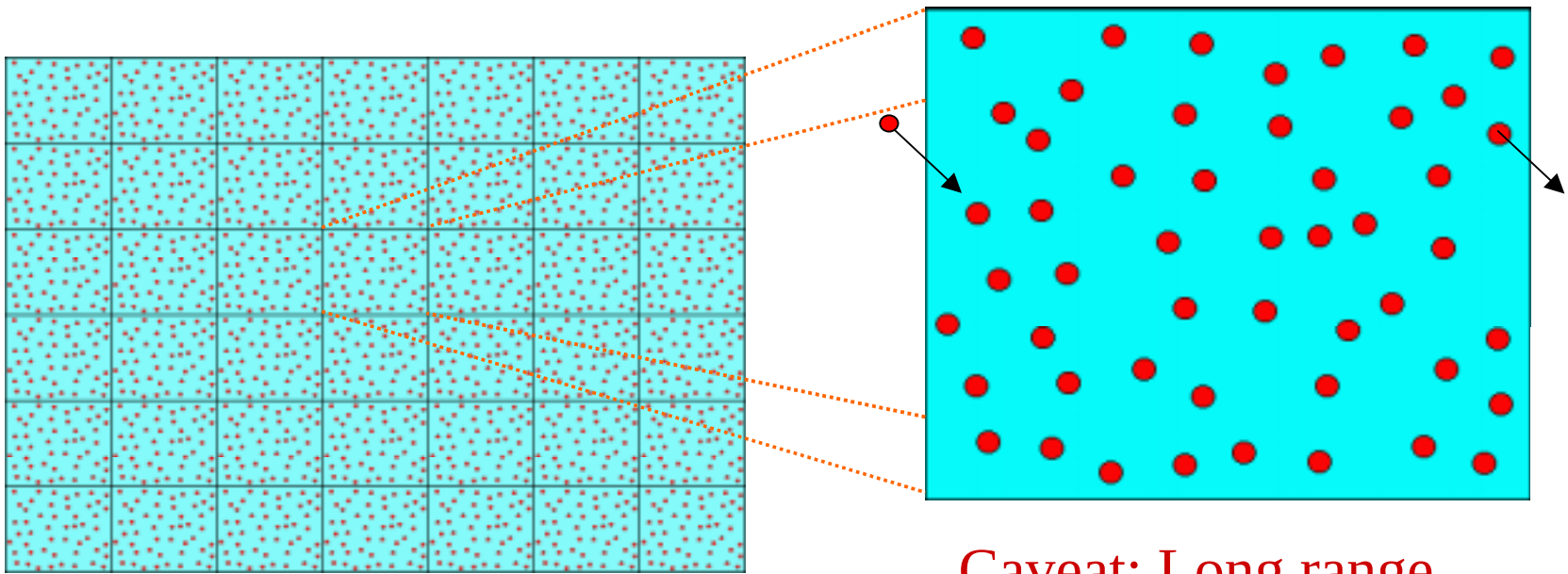
$$v(t+\Delta t) \approx v(t+\frac{1}{2}\Delta t) + \frac{1}{2}\Delta t a(t+\Delta t)$$



Basic Structure of a Molecular Dynamics Program

```
program molecular_dynamics
  call do_input_and_initialize_particles
  t=0
  do while (t<tmax)
    call   compute_forces
    call   integrate_equations_of_motion
    t=t+delta_t
    call   sample_properties
  end do
end program molecular_dynamics
```

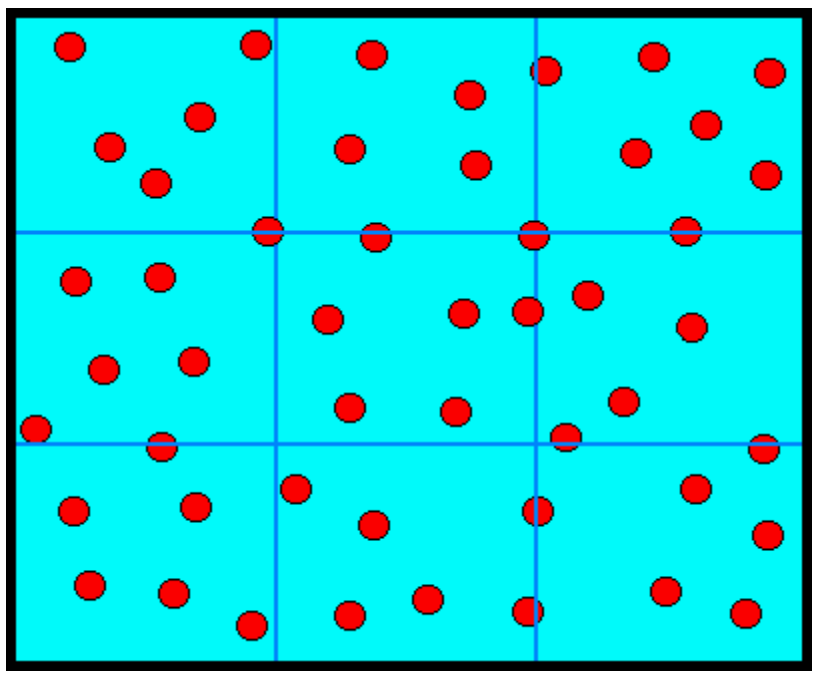
Simulating large systems



Use Periodic Boundary Conditions!

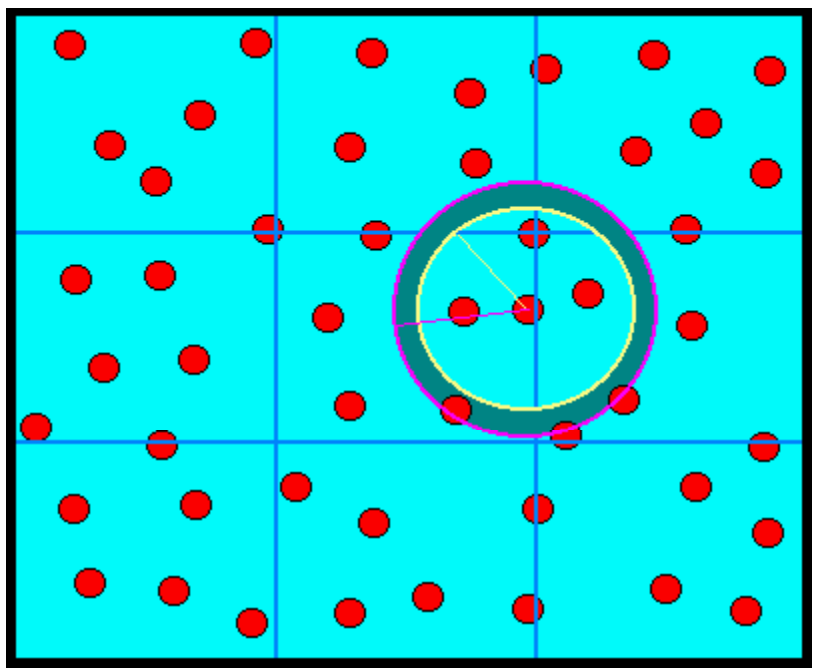
Caveat: Long range forces require tricks!

Cell Lists



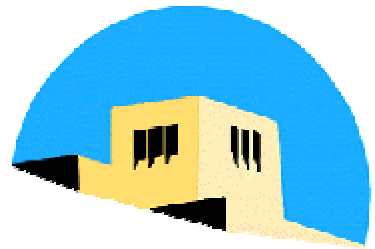
Divide region into cells of dimension comparable to potential cutoff. Only have to calculate forces within cell and nearest neighbors. Pure cell method is method of choice for N>=1000. Cost - overhead in maintaining cell lists, O(N).

Neighbor Lists



Reduce computational overhead in force loop from $O(N^2)$ to $O(N\rho\sigma_n^3)$.

Cost - overhead involved in periodically rebuilding neighbor list, storage for list. Trade off - size of nn list, time between rebuilds.



Particle structure

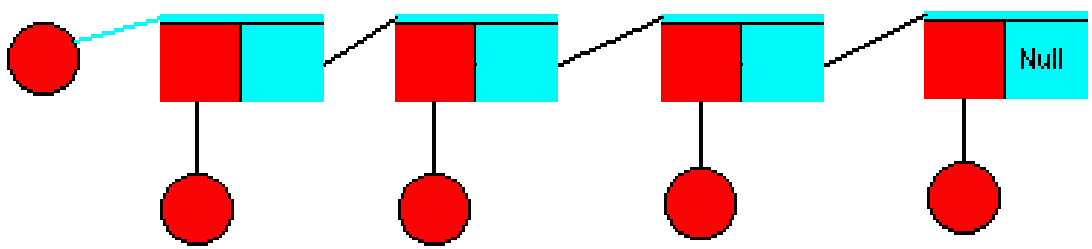
```
type particle
   integer :: label
   real(WP), dimension(idim) :: pos, vel, frc    ! data
   type(cell), pointer :: containing_cell
   ! Doubly linked list for cell list
   type(particle), pointer :: prev_atom_ptr, next_atom_ptr
   ! Singly linked list used for neighbor list
   type(particle_stack_node), pointer :: neighbor_list_ptr
end type particle
```

Cell list can be organized in a circle because cell membership is transitive.

Neighbors of a particle not same as neighbors of a neighbor. Need another structure.

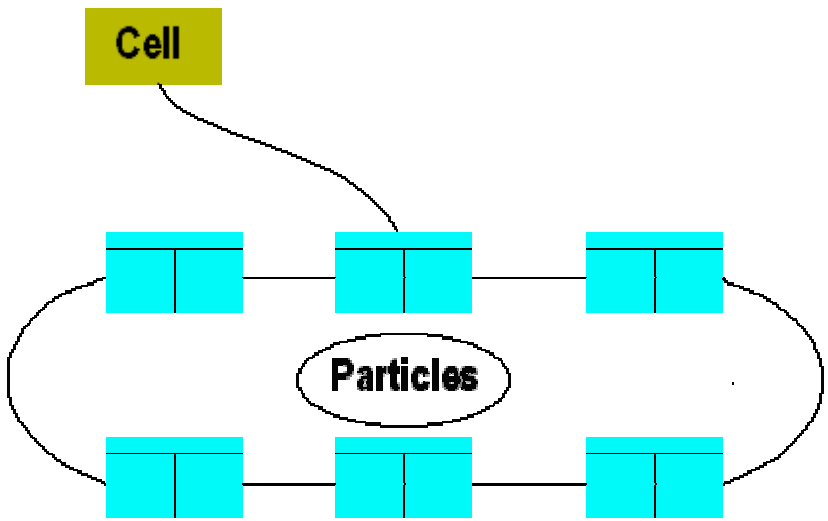
Singly linked list

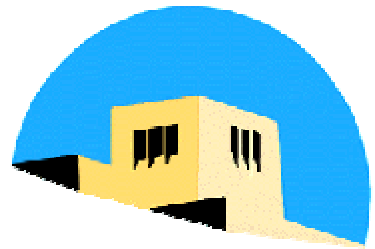
```
type particle_stack_node
   type(particle), pointer :: neighbor
   type(particle_stack_node), pointer :: next
end type particle_stack_node
```



Cell List Implementation

(Doubly-linked list)

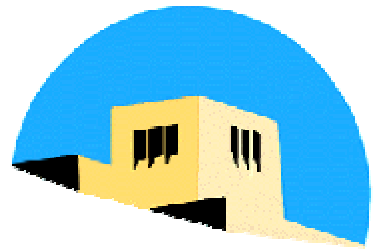




Cell Implementation

```
type cell
   integer, dimension(idimension) :: label
   type(particle), pointer :: particle_list_ptr
   type(cell_pointer), dimension(max_neighbor_cells) ::  my_neighbor_cells
end type cell
type cell_pointer
  type(cell), pointer :: ptr
end type cell_pointer
```

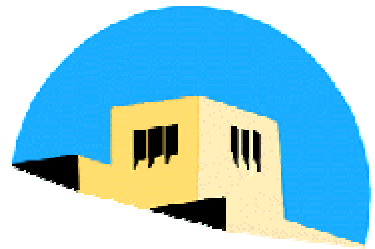
Need the cell_pointer type to implement an array of pointers since unlike C/C++, F90 pointers do not allow this directly.



```
subroutine force ...
  do i = 1,natoms
    atom(i)%frc = 0.0_WP  ! array assignment
  enddo
  do i = 1,natoms-1
    this_atom=>atom(i);  tmppos=this_atom%pos; my_list=>this_atom%neighbor_list_ptr
    do while(associated(my_list))
      if (my_list%neighbor%label.gt.i) then
        delpos=tmppos-my_list%neighbor%pos
        delpos=delpos-box_size*anint(delpos/box_size)
        rsq = dot_product(delpos,delpos)
        if (rsq<cutforcesq) then  ...
          this_atom%frc = this_atom%frc + delpos*frc
          my_list%neighbor%frc = my_list%neighbor%frc - delpos*frc
        endif
      endif
      my_list=>my_list%next
    enddo
  enddo
```

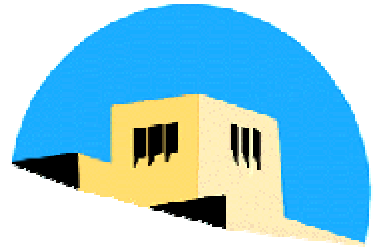
Interact with closest image

Use Newton's 3rd Law



Updating Cell Lists

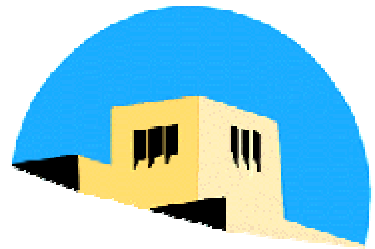
```
subroutine update_cell_lists
  implicit none
  integer :: iatom
  type(particle), pointer :: this_atom
  type(cell), pointer :: this_cell
  integer, dimension(idimension) :: cell_index
  do iatom=1,natoms
    this_atom=>atom(iatom)
    this_cell=>this_atom%containing_cell
    cell_index=floor(this_atom%pos*ncells_per_row/box_size)+1
    if(all(this_cell%label==cell_index)) cycle
    call remove_atom_from_cell(this_atom,this_cell)
    call add_atom_to_cell(this_atom,cell_index)
  end do
 end subroutine update_cell_lists
```



```
subroutine add_atom_to_cell(an_atom,ind)
  implicit none
  type(particle), intent(inout), target :: an_atom
  type(particle), pointer :: tmp_atom
  integer, dimension(idimension), intent(in) :: ind
  ! Change line below for 2D code
  type(cell), pointer :: this_cell
  this_cell=>cell_array(ind(1),ind(2),ind(3))
  if(associated(this_cell%particle_list_ptr)) then
    tmp_atom=>this_cell%particle_list_ptr%next_atom_ptr
    this_cell%particle_list_ptr%next_atom_ptr=>an_atom
    tmp_atom%prev_atom_ptr=>an_atom
    an_atom%prev_atom_ptr=>this_cell%particle_list_ptr
    an_atom%next_atom_ptr=>tmp_atom
    an_atom%containing_cell=>this_cell
  else
    this_cell%particle_list_ptr=>an_atom
    an_atom%next_atom_ptr=>an_atom
    an_atom%prev_atom_ptr=>an_atom
    an_atom%containing_cell=>this_cell
  endif
end subroutine add_atom_to_cell
```

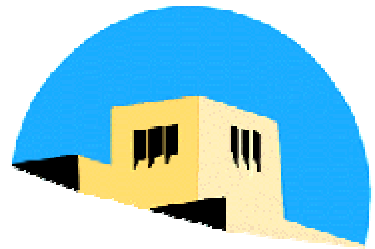
Adding a particle to a cell involves pointing the particle pointers in the particle structure to 0,1, or 2 of my neighbors and vice versa. Point cell to ring of particles. Point particle to containing cell.

First atom in list



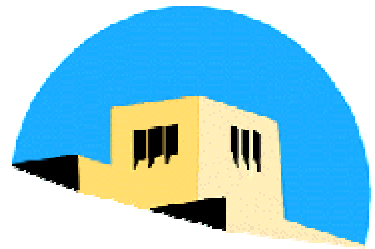
```
subroutine remove_atom_from_cell(an_atom,a_cell)
  implicit none
  type(particle), intent(inout), target :: an_atom
  type(cell), intent(inout) :: a_cell
  type(particle), pointer :: tmpnext, tmpprev
  tmpnext=>an_atom%next_atom_ptr
  tmpprev=>an_atom%prev_atom_ptr
  if (associated(tmpnext,an_atom)) then
    nullify(a_cell%particle_list_ptr)
  else
    if(associated(a_cell%particle_list_ptr,an_atom)) then
      a_cell%particle_list_ptr=>tmpnext
    endif
    ! Join the 2 particles on  either side of an_atom.
    tmpnext%prev_atom_ptr=>tmpprev
    tmpprev%next_atom_ptr=>tmpnext
  endif
 end subroutine remove_atom_from_cell
```

To remove the atom from the cell, first determine if we are the only atom in the list. If not, we need to connect the remaining particles and perhaps change the cell's pointer to the ring. Note I don't reset all the pointers in an_atom as they will be updated when it is added to a new cell.



```
subroutine energy(eng)
  ...
  eng = 0.0_WP
  do i = 1,natoms-1
    this_atom=>atom(i);  tmppos=this_atom%pos
    my_list=>this_atom%neighbor_list_ptr
    do while (associated(my_list))
      if(my_list%neighbor%label.gt.i) then
        delpos=tmppos-my_list%neighbor%pos
        delpos=delpos-box_size*anint(delpos/box_size)
        rsq = dot_product(delpos,delpos)
        if (rsq<cutforcesq) eng = eng + phi(rsq) - eng_cutoff
      endif
      my_list=>my_list%next
    enddo
  enddo
  eng = eng/natoms
  return
end subroutine energy
```

The term in red is a correction for using cutoff potentials.



Nearest Image Convention - F77 versus F90

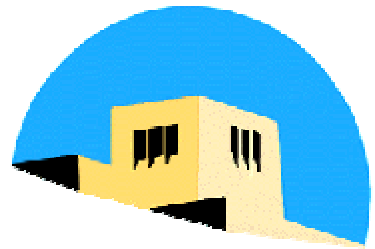
F77

```
delx = xtmp - x(1,j)
if (delx.gt.xprd2) then
        delx = delx - xprd
else if (delx.lt.-xprd2) then
        delx = delx + xprd
endif
…
rsq=delx*delx+...
```

Notice that using anint to rewrite the nearest image expression in a way that is independent of the dimension of space.

F90

```
delpos=current_atom%pos-next_atom%pos
delpos=delpos-box_size*anint(delpos/box_size)
rsq = dot_product(delpos,delpos)
```

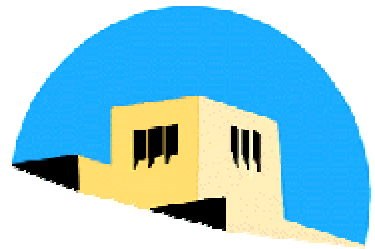


Other things to notice

To convert the code to 2D requires changing 1 parameter, 1 line in add_atom_to_cell, and 10 executable lines in setup_cells. The rest of the code is completely independent of the dimension of space.

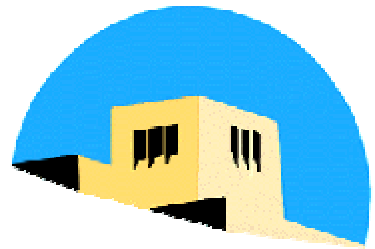
If you run the code on the Turing cluster, you will find the f90 code is slower for the given input file. Part of this because the f95 compiler is not an optimizing compiler, part may also be due to the allocation/deallocation of neighbor list pointers.

Exercise: Try removing the neighbor list code in favor of using cell lists exclusively. Note that the cell lists will have to be updated before every call to force. Notice any difference?



References

- Numerical Integration
  - “Methods of Numerical Integration”, Philip J. Davis and Philip Rabinowitz, Academic Press, New York, 1984.
  - “Numerical Recipes in C”, Press, Vetterling, Teukolsky, and Flannery, Cambridge University Press, 1992.



References

- Molecular Dynamics
  - “Computer Simulation of Liquids”, M.P. Allen and D.J. Tildesley, Oxford University Press, 1996.
  - “Understanding Molecular Simulation”, Daan Frenkel and Berend Smit, Academic Press, 1996.
  - “The Art of Molecular Dynamics Simulation”, D.C. Rapaport, Cambridge University Press, 1995.