Molecular Dynamics with Verlet Neighbor Lists

CD61004: Group Project (Autumn 2025)

- Aditya Raj
- Amaan Akhtar
- Challa Sai Ram Reddy
- Vemulapalli Maanasa

1. Introduction

Molecular Dynamics (MD) simulations are a powerful computational tool for studying the physical movements of atoms and molecules. The core of an MD simulation is the calculation of forces between all pairs of atoms to determine their trajectories over time. A naive implementation checks every possible pair of atoms, resulting in a computational complexity of $O(N^2)$, where N is the number of atoms. As system sizes increase, this becomes prohibitively expensive.

The **Verlet neighbor list** is a standard algorithm used to accelerate these calculations. It works by maintaining a list of "nearby" atoms for each atom in the system. Instead of iterating over all $N-1$ other atoms during every force calculation step, the algorithm only iterates over the much smaller set of atoms in the neighbor list. This reduces the complexity of the force calculation to approximately $O(N)$.

The neighbor list is built by defining two cutoff radii: 1. r_{cut} **(Potential Cutoff):** The distance beyond which interatomic forces are assumed to be zero. 2. r_{list} **(Neighbor List Cutoff):** A slightly larger radius, $r_{list} = r_{cut} + r_{skin}$, where r_{skin} is a "skin" depth.

All atoms within r_{list} of atom i are stored in its neighbor list. During subsequent time steps, as long as no atom has moved more than $r_{skin}/2$, the neighbor list remains valid. This allows the expensive list-building operation to be performed infrequently (e.g., every 10-20 steps), while the frequent force calculations use the efficient, pre-calculated list.

2. Implementation Details

The Verlet neighbor list has been implemented in the provided Fortran code with the following key components:

- **Data Structures:**
  - `nlist(:,:)`: A 2D array where `nlist(i, k)` stores the index of the k-th neighbor of atom i.
  - `n_neigh(:)`: A 1D array where `n_neigh(i)` stores the total number of neighbors for atom i.

  - `old_r(:,:)`: Stores the atomic positions at the time the last neighbor list was built, used to track atom displacement.
- **Key Parameters:**
  - `Rskin`: The skin thickness. It is dynamically estimated based on the root-mean-square velocity (v_{rms}) of the system to target a specific update frequency (`Nupdate_target = 20` steps). It is clamped between `0.2 Å` and `1.0 Å`.

$$R_{skin} \approx 2 \cdot v_{rms} \cdot \Delta t \cdot N_{update}$$

  - **Auto-rebuild Trigger:** The list is rebuilt if any atom has moved more than half the skin distance since the last build: $2 \cdot \max(|\vec{r}_i(t) - \vec{r}_i(t_{build})|) > R_{skin}$. This check is performed every 5 steps (`check_interval`) for efficiency.
- **Algorithm Flow:**
  1. **Initialization:** An initial neighbor list is built, and memory is allocated based on an estimated maximum number of neighbors.
  2. **Integration Loop:**
     - Positions are updated.
     - Every few steps, the maximum atomic displacement is checked against R_{skin}. If the threshold is exceeded, `build_neighbor_list` is called.
     - `force_calc` is called, which now iterates only over `nlist` instead of all atom pairs.
  3. **Force Calculation:** The inner loop for atom i now runs from `1` to `n_neigh(i)`, checking only atoms `j = nlist(i, n)`.

3. Performance Analysis

The implementation was tested on three Argon systems of increasing size (1600, 5400, and 12800 atoms) for 600 MD steps.

System Size (Atoms)	Wall Time (Naive) [s]	Wall Time (Verlet) [s]	Speedup Factor
1600	~17.7	~2.1	**~8.4x**
5400	~126.2	~10.3	**~12.25x**
12800	>1000	~69.6	**~14.36x**

4. Discussion

Choosing Reasonable r_{skin}

Choosing r_{skin} is a trade-off between memory/force-calculation time and list-building time. * **Too small r_{skin}:** Requires very frequent (expensive) list rebuilds. * **Too large r_{skin}:** The neighbor list becomes huge, wasting memory

and time checking pairs that are within r_{list} but outside r_{cut} (where force is zero).

A reasonable r_{skin} is typically chosen so the list is updated every 10–20 time steps. It can be estimated using the maximum particle velocity (v_{max}) in the system and the simulation time step (Δt):

$$r_{skin} \approx 2 \cdot v_{max} \cdot \Delta t \cdot N_{update}$$

where N_{update} is the desired number of steps between list rebuilds.

Criterion for rebuild

Track the **displacement** of each atom since the last list update:

$$\Delta r_i = |\mathbf{r}_i(t) - \mathbf{r}_i(t_{\text{last list}})|$$

Then compute

$$\text{max\_disp} = \max_i \Delta r_i$$

You **rebuild the list** when:

$$2 \times \text{max\_disp} > R_{\text{skin}}$$

The factor of 2 ensures that two atoms moving toward each other won't get closer than (R_cut) without being in each other's list.

Constraints under PBC in a Cubic Box

Under Periodic Boundary Conditions (PBC) in a cubic box of side length L, the Minimum Image Convention requires that a particle does not interact with its own image or multiple images of another particle.

For a standard Verlet list implementation, strictly maintain:

$$r_{cut} + r_{skin} < \frac{L}{2}$$

If $r_{cut} + r_{skin} \geq L/2$, a particle i might incorrectly list the same particle j (or its own image) twice from different periodic directions, leading to incorrect forces or requiring complex, slower checks during the force loop.

5. Code

Neighbor List with PBC constraint

```
subroutine build_neighbor_list(TotAtom, Box, Rcut, Rskin, r, nlist, n_neigh)
  use general, only: dp
  implicit none
  integer, intent(in) :: TotAtom
  real(kind=dp), intent(in) :: Box, Rcut, Rskin
  real(kind=dp), intent(in) :: r(TotAtom,3)
  integer, intent(inout) :: nlist(TotAtom, *), n_neigh(TotAtom)

  integer :: i, j
  real(kind=dp) :: dx, dy, dz, r2, Rlist2

  Rlist2 = (Rcut + Rskin)**2
  n_neigh = 0

  do i = 1, TotAtom - 1
     do j = i + 1, TotAtom
        dx = r(i,1) - r(j,1); dy = r(i,2) - r(j,2); dz = r(i,3) - r(j,3)

        ! minimum image
        if (dx > 0.5d0*Box) dx = dx - Box
        if (dx < -0.5d0*Box) dx = dx + Box
        if (dy > 0.5d0*Box) dy = dy - Box
        if (dy < -0.5d0*Box) dy = dy + Box
        if (dz > 0.5d0*Box) dz = dz - Box
        if (dz < -0.5d0*Box) dz = dz + Box

        r2 = dx*dx + dy*dy + dz*dz
        if (r2 <= Rlist2) then
           n_neigh(i) = n_neigh(i) + 1
           nlist(i, n_neigh(i)) = j
           n_neigh(j) = n_neigh(j) + 1
           nlist(j, n_neigh(j)) = i
        end if
     end do
  end do
end subroutine build_neighbor_list
```

integrate.f90 Check to rebuild list

```
  ! --- Rebuild check every 'check_interval' timesteps to save cost ---
  if (mod(step, check_interval) == 0) then
     disp = r - old_r
     max_disp = maxval(sqrt(sum(disp**2, dim=2)))
     if (2.d0 * max_disp > Rskin) then
        call build_neighbor_list(TotAtom, Box, Rcut, Rskin, r, nlist, n_neigh)
        old_r = r
```

```
      disp = 0.d0
   end if
end if
```

force.f90 calc for neighbors only

```
do i = 1, TotAtom
   do n = 1, n_neigh(i)
      j = nlist(i,n)
      if (j <= i) cycle

      dx = r(i,1) - r(j,1)
      dy = r(i,2) - r(j,2)
      dz = r(i,3) - r(j,3)

      if (dx > 0.5d0*Box) dx = dx - Box
      if (dx < -0.5d0*Box) dx = dx + Box
      if (dy > 0.5d0*Box) dy = dy - Box
      if (dy < -0.5d0*Box) dy = dy + Box
      if (dz > 0.5d0*Box) dz = dz - Box
      if (dz < -0.5d0*Box) dz = dz + Box

      r2 = dx*dx + dy*dy + dz*dz
      if (r2 > rcut2) cycle

      r6 = (Sig**2 / r2)**3
      r12 = r6*r6
      ffac = 48.d0*Eps*(r12 - 0.5d0*r6)/r2
      f = ffac * (/dx, dy, dz/)

      Force(i,:) = Force(i,:) + f
      Force(j,:) = Force(j,:) - f
      PE = PE + 4.d0*Eps*(r12 - r6)
   enddo
enddo
```

6. Conclusion:

The Verlet neighbor list provides a dramatic performance improvement that scales with system size. While the brute-force $O(N^2)$ approach becomes unfeasible for large systems, the neighbor list maintains near-linear scaling $O(N)$, making simulations of tens or hundreds of thousands of atoms practical. The overhead of rebuilding the list is negligible compared to the time saved in force calculations.