Neighbor lists

July 26, 2017

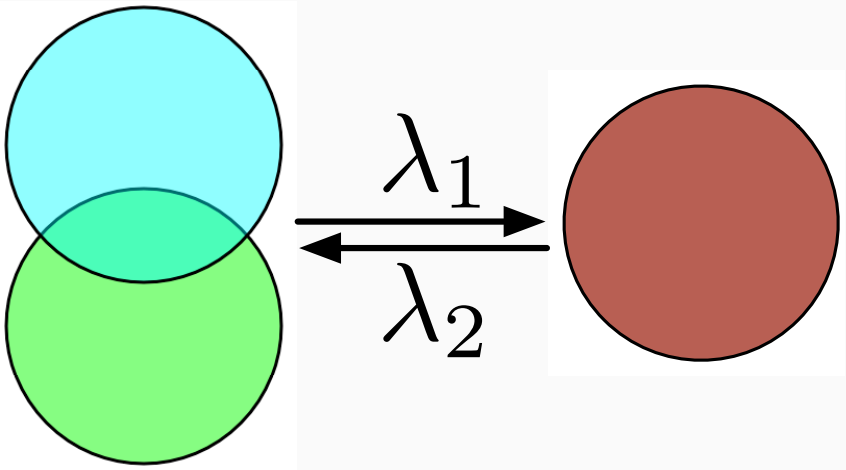
CMB Group, FU Berlin

Table of contents

Introduction

Introduction

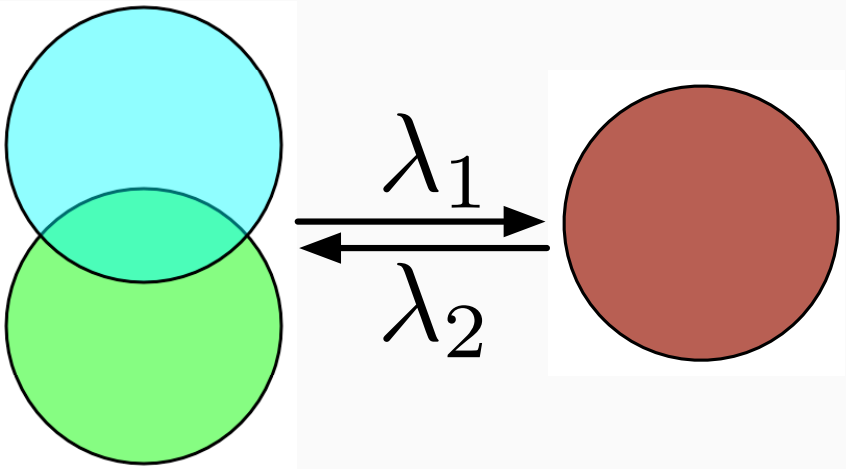
- Central operation in particle simulations: Calculation of distances between atoms.
- In MD: Energy and force calculations.
- In particle based Reaction–Diffusion: Energy and force calculations, handling of reactions.



Complexity?

Introduction

- Central operation in particle simulations: Calculation of distances between atoms.
- In MD: Energy and force calculations.
- In particle based Reaction–Diffusion: Energy and force calculations, handling of reactions.



Complexity $\mathcal{O}(N^2)$ (naively)

Example: The Lennard–Jones potential

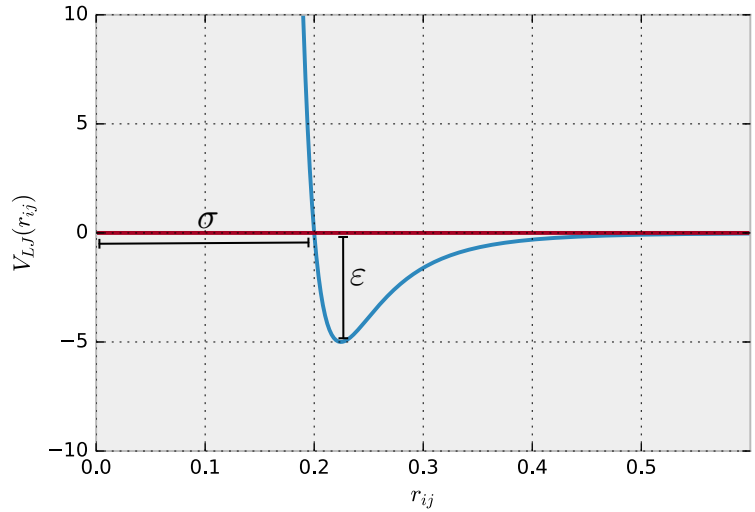
The Lennard–Jones potential is a common model of two-body interatomic interaction.

$$V_{\mathrm{LJ}}(r_{ij}) := 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], \qquad V_{\mathrm{tot}} = \frac{1}{2} \sum_{i=1}^{N} \sum_{j \neq i}^{N} V_{\mathrm{LJ}}(r_{ij}).$$

To compute the system's energy, the complexity is again $\mathcal{O}(N^2)$.

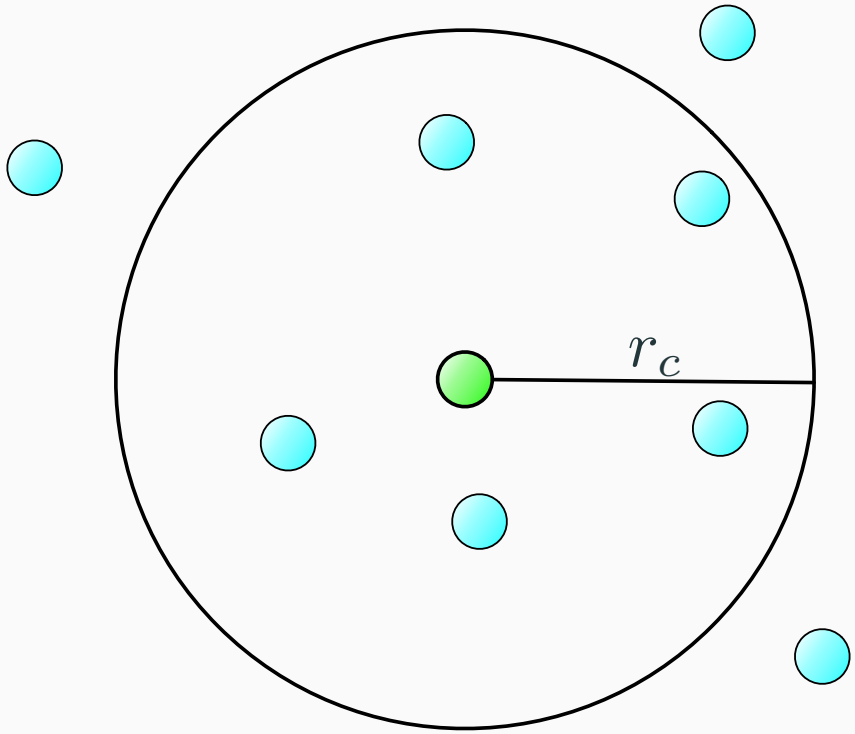
Applying Newton's third law, the number of computation steps can be reduced to

$$\frac{N(N-1)}{2}.$$



Locality

But: Everything is more or less local, so it makes sense to introduce a cutoff r_c and consider only the nearest neighbors within that radius.



For the Lennard–Jones potential r_c is usually chosen as

$$V_{\mathrm{LJ}}(r_c) = V_{\mathrm{LJ}}(2.5\sigma) \approx -\frac{\varepsilon}{60}.$$

A first approach

- Want: A data structure containing the neighboring particles, so that the complexity of, e.g., force and energy calculations reduces.
- Idea: Loop over all pairs of particles and see if they are close enough:

```
neighbor_list = []

for i in range(0, n_particles):
    for j in range(0, n_particles):
        if i != j and d(i, j) < r_cutoff:
            neighbor_list.append( (i, j) )
```

A first approach

- Want: A data structure containing the neighboring particles, so that the complexity of, e.g., force and energy calculations reduces.
- Idea: Loop over all pairs of particles and see if they are close enough:

```
neighbor_list = []

for i in range(0, n_particles):
    for j in range(0, n_particles):
        if i != j and d(i, j) < r_cutoff:
            neighbor_list.append( (i, j) )
```

Using this list, the complexity of force and energy calculations reduced to $\mathcal{O}(N)$, but the creation is still in $\mathcal{O}(N^2)$.

A first approach, cont.

To avoid having (i, j) and (j, i) in the created list:

```
neighbor_list = []

for i in range(0, n_particles):
    for j in range(i+1, n_particles):
        if d(i, j) < r_cutoff:
            neighbor_list.append( (i, j) )
```

The construction of the list is now only half as expensive, but still is in $\mathcal{O}(N^2)$ and therefore not of great use for large particle numbers.

Also, the data structure is bad for parallelization over particles or particle pairs. How can we gain more efficiency?

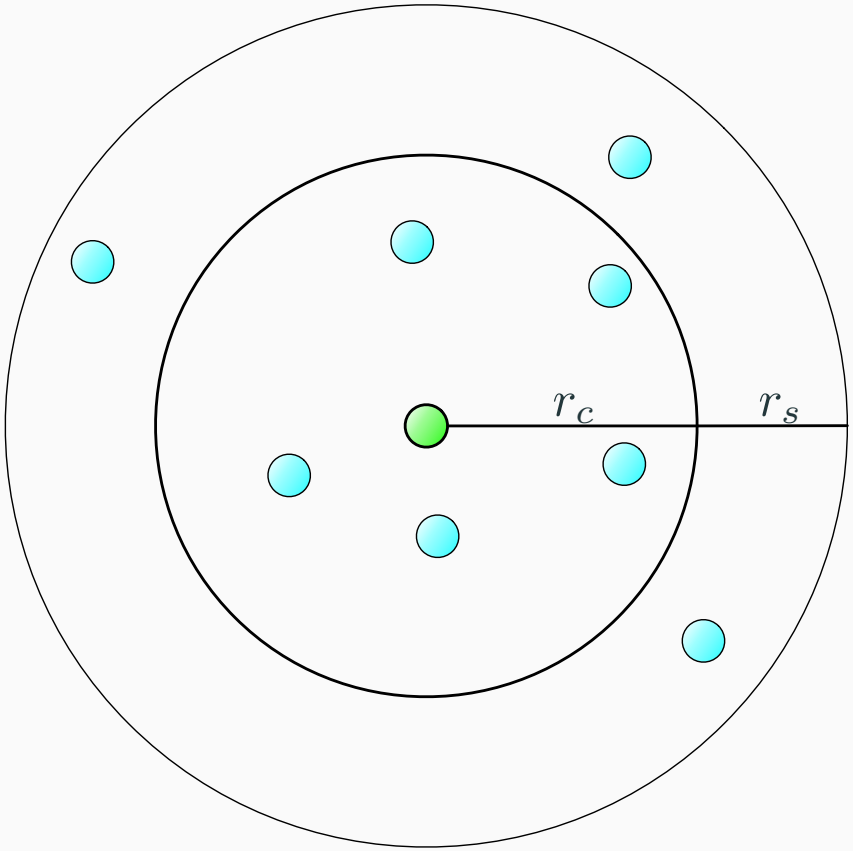
Verlet neighbor lists

Verlet neighbor lists

- Introduce a skin region r_s and construct a list that contains for each particle i the indices of all particles j whose distance is smaller than $r_c + r_s$.
- Update the neighbor list every N_s time steps.
- N_s and r_s are chosen such that

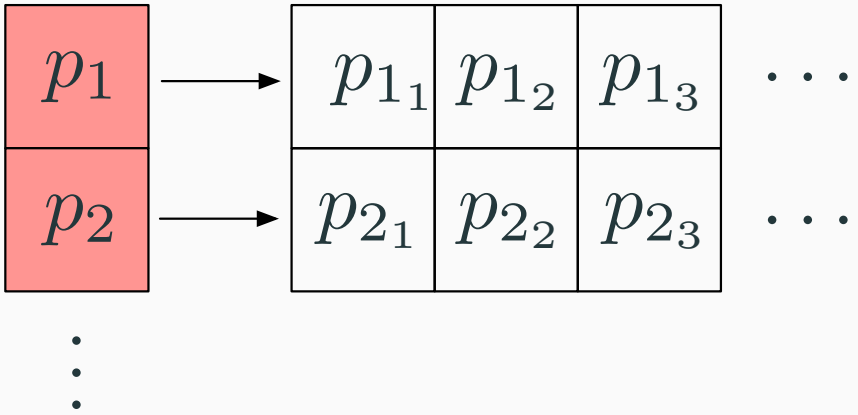
$$r_s > N_s v_{\text{typ}} \delta t,$$

where v_{typ} is the typical speed of a particle.

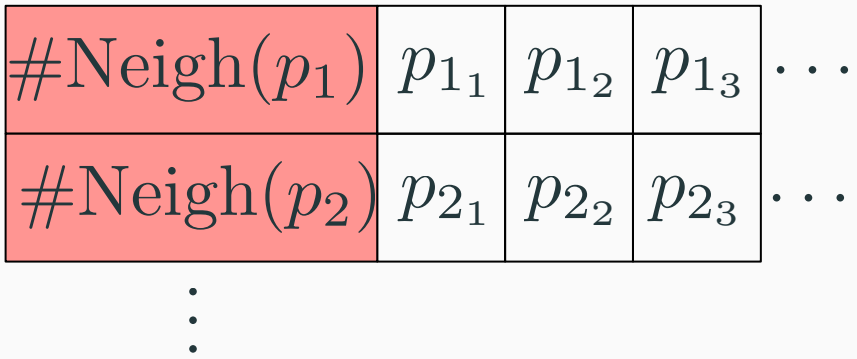


Verlet neighbor lists in practice

Data structure hash table:



Data structure array:



- Mapping from particle index to particle indices of neighbors.
- Can be implemented using a hash table with higher memory requirements.
- Can be implemented using an array without random-access but cheaper to create.

Example implementation (contiguous list)

```
neighbor_list = [] # empty list
start_of_neighbor_list = 0
for i in range(0, n_particles):
    # count the neighbors
    n_neighbors = 0
    # placeholder for n_neighbors
    neighbor_list.append(0)
    for j in range(0, n_particles):
        if j != i and d(i, j) < r_cutoff + r_s:
            n_neighbors += 1
            neighbor_list.append(j)
    neighbor_list[start_of_neighbor_list] = n_neighbors
    # start for the next particle is current length
    start_of_neighbor_list = len(neighbor_list)
```

Conclusion for Verlet lists

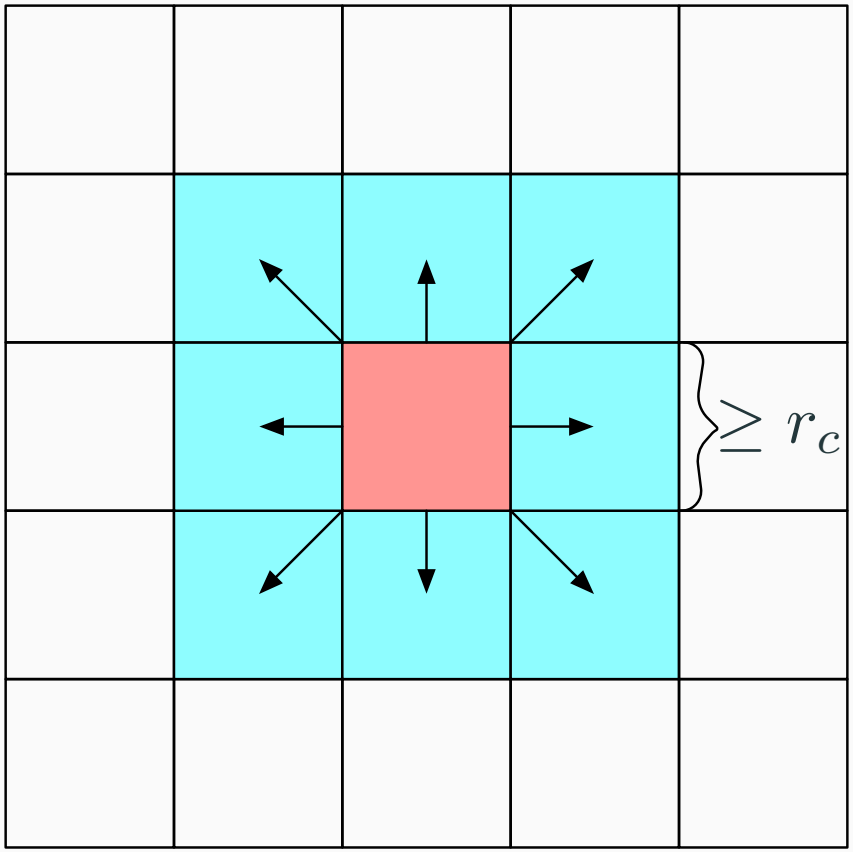
- The complexity to create a Verlet list is still $\mathcal{O}(N^2)$ but got reduced by N_s, the number of time steps until which it gets re-created.
- If random access of the particles is needed, one should invest into a hash table structure or provide padding for the neighbors in the contiguous list, which is more expensive to create (and in the padding case requires knowledge about the maximum number of neighboring particles) but provides faster lookup.

Cell linked-lists

Cell linked-lists

- The Verlet list still has a creation complexity of $\mathcal{O}(N^2)$, so it is unsuitable for large particle numbers.
- With cell linked-lists, one can get the complexity down to $\mathcal{O}(N)$.
- The domain is decomposed into cells whose edge length is larger or equal than r_c.
- All particles are assigned to these cells by their position ($\mathcal{O}(N)$).
- The neighboring particles can only be in the own cell or in one of the neighboring cells.

Cell linked-lists



The complexity for, e.g., energy and force calculations in 3 dimensions is $\mathcal{O}(27N_{\text{sub}}N) \in \mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$, where N_{sub} is the average number of particles in each cell.

Cell linked-lists: A practical implementation

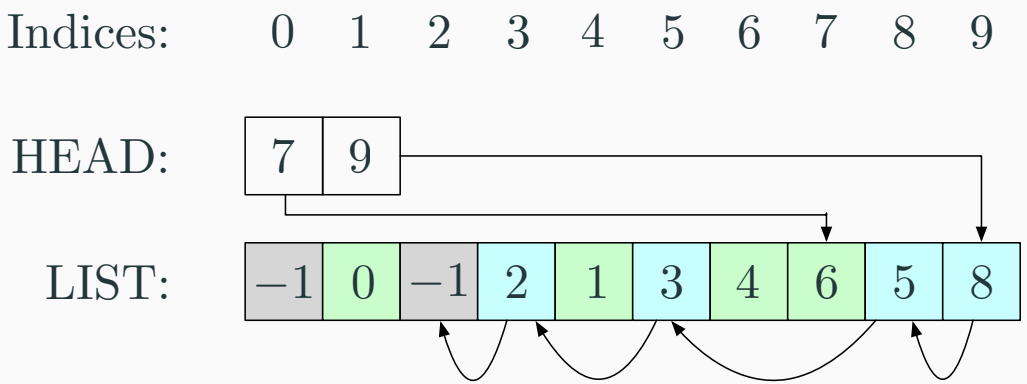
Array HEAD:

- Size is #(cells)
- Contains pointers to the LIST
- Tells, where neighbors of the current cell start

Example:

Array LIST:

- Size is N
- Element j points to where the next particle index of particles in this cell is



Cell linked-lists: A practical implementation

```
# initialize the HEAD array
head = [-1] * n_cells
# initialize the LIST array
list = [-1] * n_particles
for i in range(0, n_particles):
    # calculate the cell's index
    # by the particle's position
    cell_index = get_cell_index(i)
    # the current particle points
    # to the old head of the cell
    list[i] = head[cell_index]
    # the new head of the cell
    # is the current particle
    head[cell_index] = i
```

Cell linked-lists: A practical implementation

Assuming that the simulation box $L \times L \times L$ is centered around the origin, the cells have an edge width of M and are enumerated first by z, then y, then x-axis, the cell index of position $\mathbf{p} = (p_0, p_1, p_2)$ can be computed by

```
i = floor((p0 + .5 * L) / M)
j = floor((p1 + .5 * L) / M)
k = floor((p2 + .5 * L) / M)
ax_n_c = floor(L / M)
cell_index = k + j * ax_n_c + i * ax_n_c * ax_n_c
```

Note that a two-dimensional array (in python list of lists or two-dimensional numpy array) could be used as well, but requires more memory or dynamic allocation, which is less efficient.

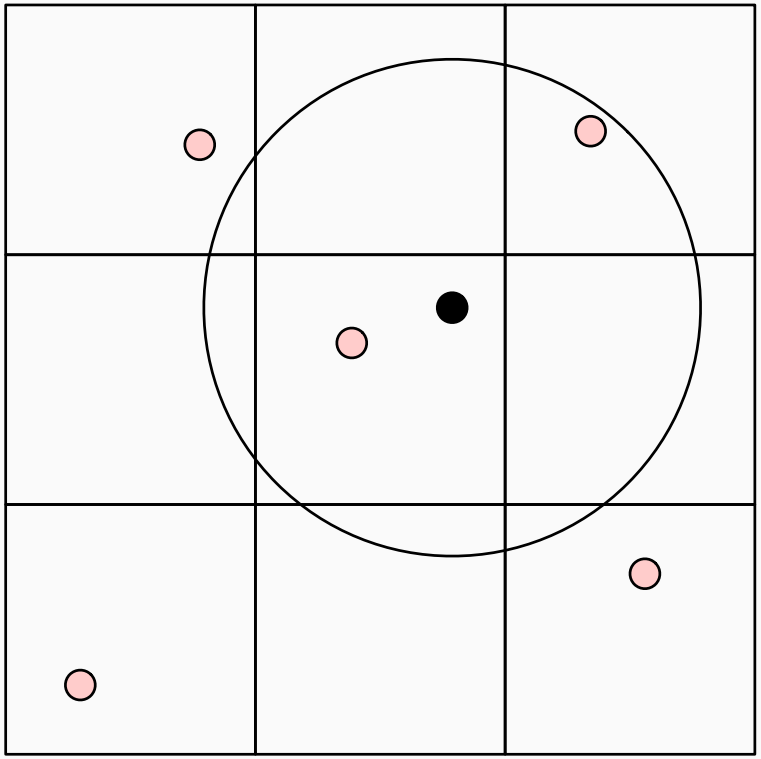
Cell linked-lists vs. Verlet lists

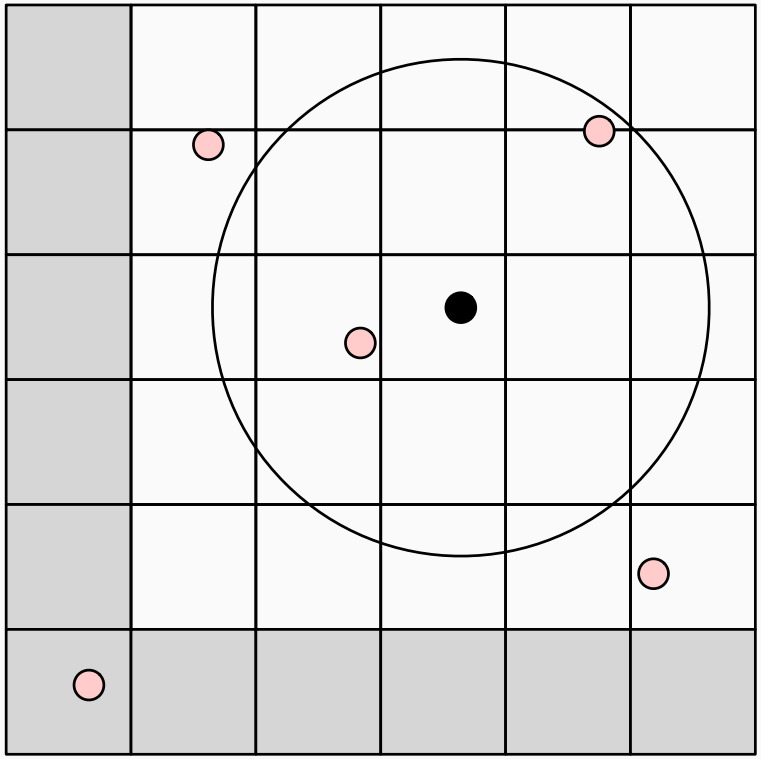
- Cell linked-lists are in $\mathcal{O}(n)$ for construction and iteration, but the list of potential neighbors for a particle is much larger than for a Verlet list.
- Common choice for cell edge size: r_c, thus for each atom a volume of $27r_c^3$ is checked in the force and energy calculations.
- Optimally, one would only consider a volume of $\frac{4}{3}\pi r_c^3 \approx 4.1 r_c^3$, i.e., about 15% of $27r_c^3$.

How to improve?

Cell linked-lists with smaller edge length

If the cell edge is $\frac{1}{2}r_c$, the checked volume will be $125\left(\frac{1}{2}\right)^3 r_c^3$, only 57.87% of the previous volume.





One can go even further and make the cell edges so small, that at most one particle fits into a box. This however has been reported to be slower than Verlet lists when the particle number is relatively small.

Cell decomposition approach

Cell decomposition approach

The cell decomposition approach combines cell linked-lists and Verlet lists:

1. Partition the domain into cells and build a cell linked-list.
2. Construct a Verlet list by searching only in neighboring cells instead of considering all particle pairs.

The overall process is now $\mathcal{O}(C \cdot N)$, where $C \neq C(N)$ is a constant.

For systems with high atomic mobility, it was found that cell edges of $\frac{1}{2} r_c$ give best performance.

Partial updating of the Verlet list

The choice of the skin r_s is performance critical:

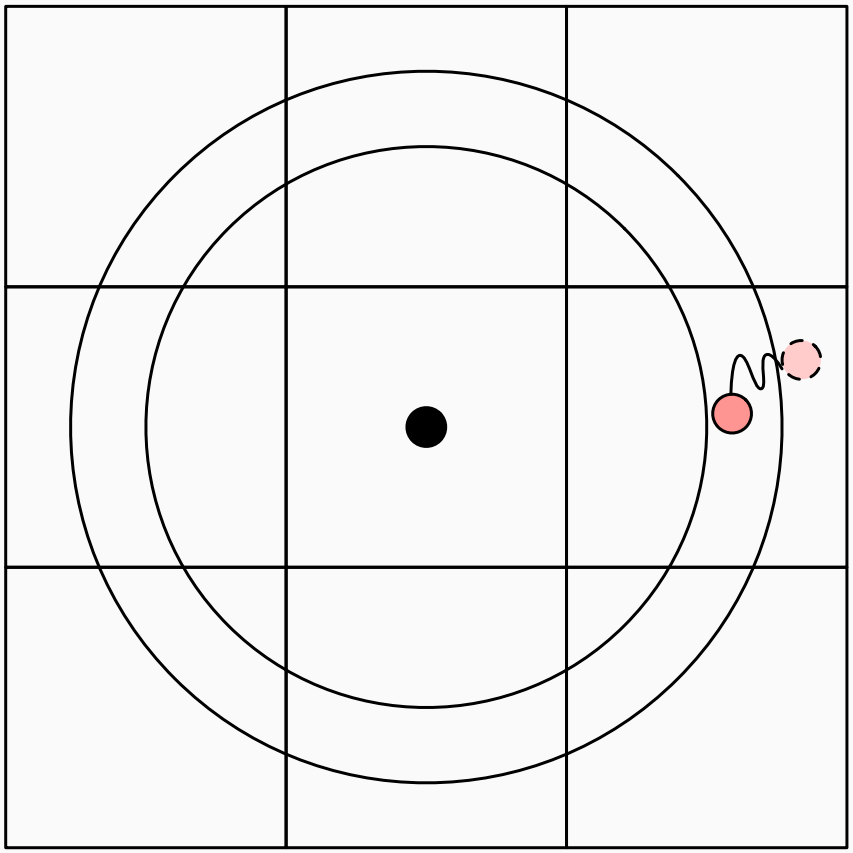
- If it is large, the neighbor list does not have to be updated for more time steps, but the forces evaluation efficiency is lowered.
- If it is small, the neighbor list has to be updated frequently, but the evaluation of forces becomes more efficient.

When the system is inhomogeneous in terms of density and particle mobility, it can be tough to find a good value.

Idea: Assign each cell a “dirty” flag, which denotes that the portion of the neighbor list corresponding to this cell and the neighboring cells should be updated.

The “dirty” flag

The cumulative displacement of each particle in the cell is tracked and as soon as the sum of the two largest displacements is greater than r_s, the cell is marked dirty.

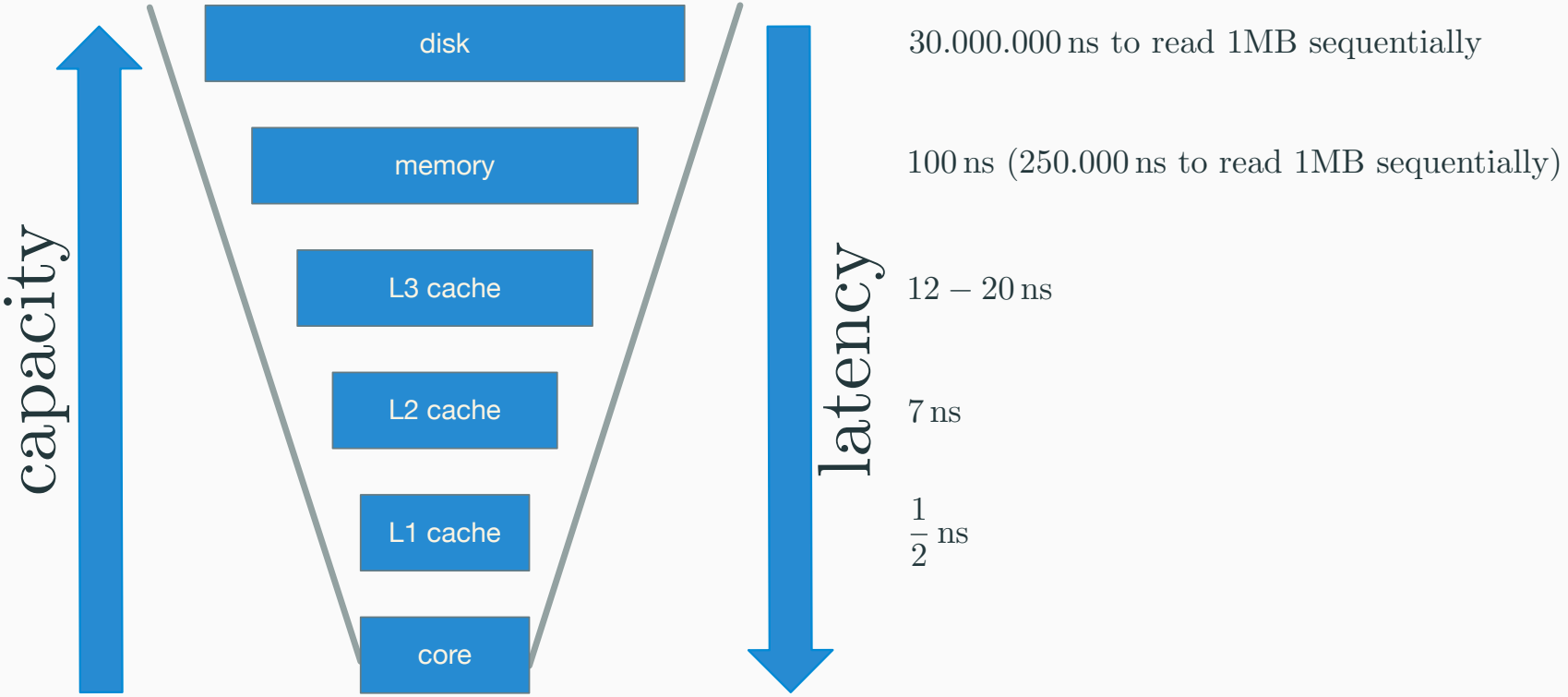


- This minimizes the time one needs to spend on updating the neighbor list and the skin can be chosen small enough so that the force evaluation efficiency is maximized.
- The neighbor lists of different atoms should now be stored separately (for example as hash table or as two-dimensional array with padding).

Optimization by spatial sorting

The principle of locality

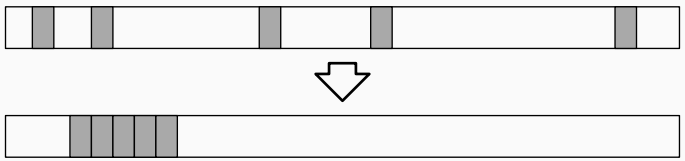
The principle of locality means: Reuse of data or use of data in relatively close storage locations.



If locality is given, the processor can attain better performance due to fewer cache misses.

The principle of locality and neighbor lists

- Particles which are in a cell and respective neighboring cells should optimally be close together.
- One obvious step to increase locality is to group the particles with respect to their cells.

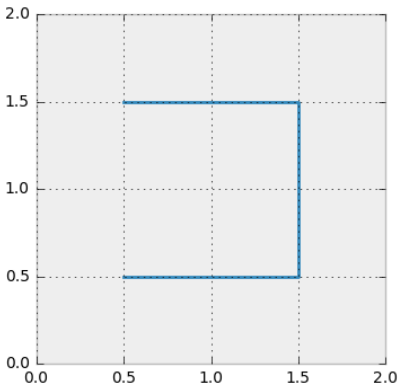


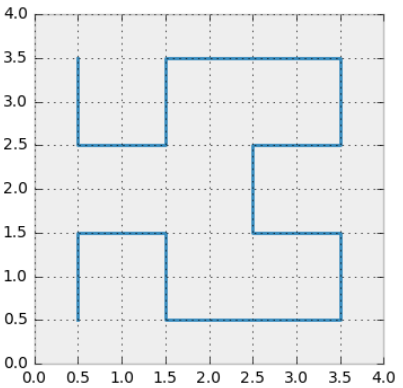
- Grouping particles into cells gives better locality within the cell, but not the neighboring cells.
- Additionally one can sort the particles along the axis that corresponds to the longest edge of the simulation box.
- One can further improve the locality by using space-filling curves.

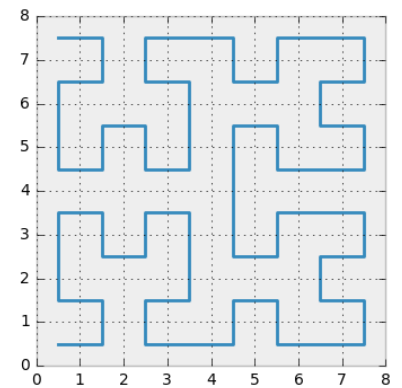
Neighbor lists using space-filling curves

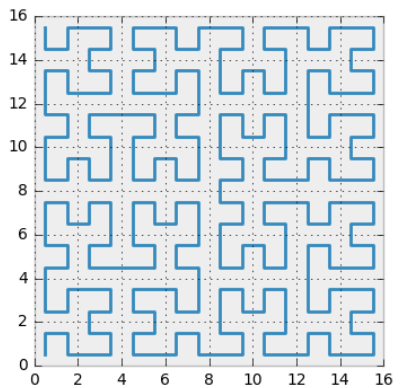
Space-filling curves are curves that pass through every point in a space. One curve that was reported to work well for neighbor lists: The Hilbert curve.

It is constructed recursively:



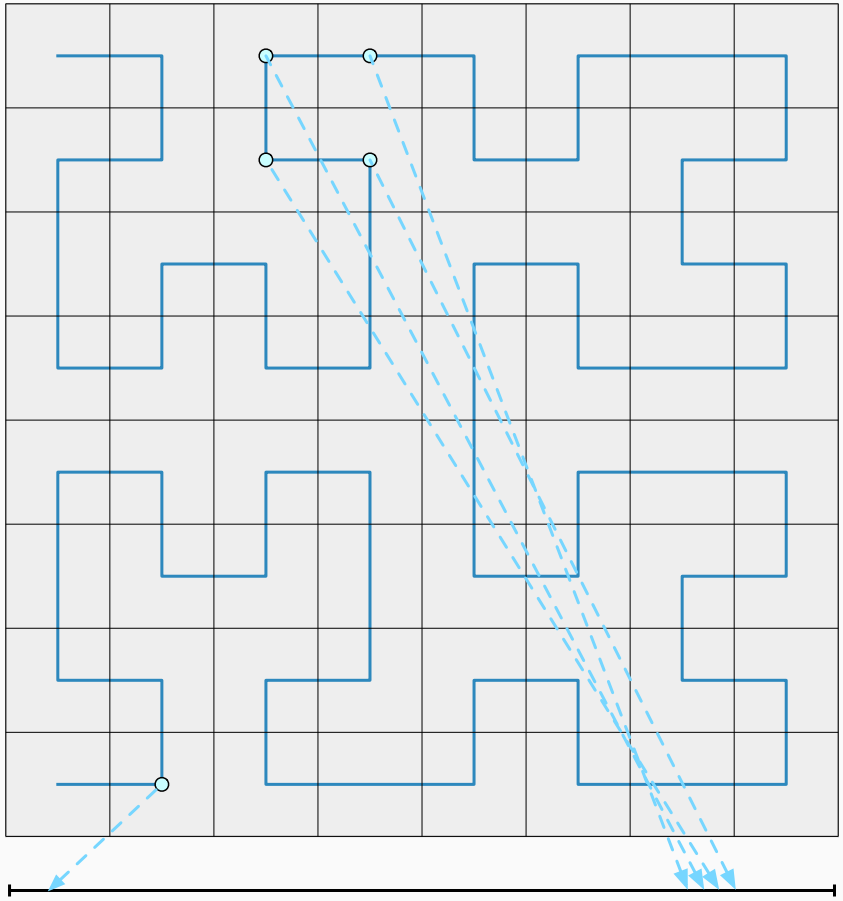




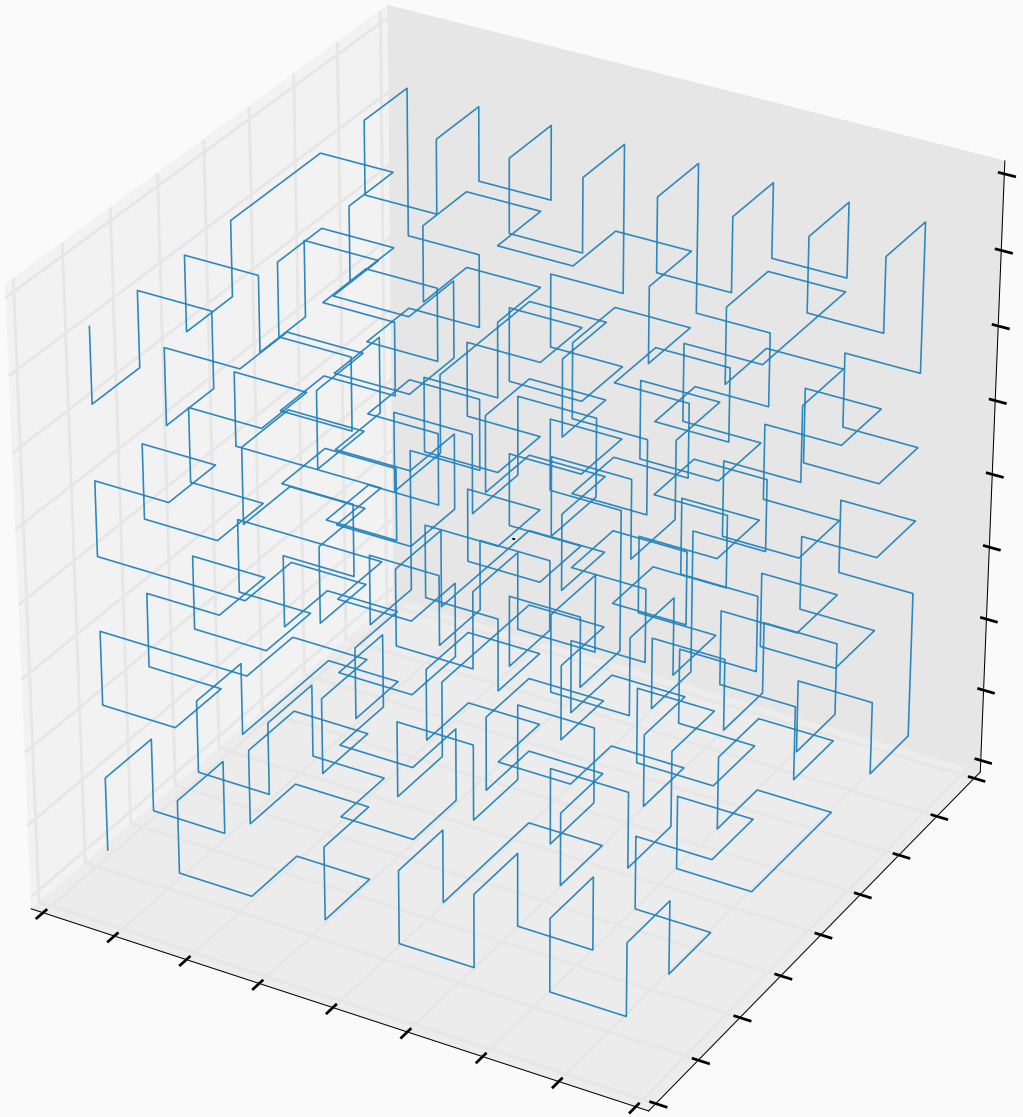


The Hilbert curve

The Hilbert curve has the good property that points that have a close index on the curve are also spatially close, while the converse is not always true. Thus it can be used to sort particles within the cells / the cells themselves.



The Hilbert curve in 3 dimensions



Summary

Summary

- The calculation of forces and energy are in $\mathcal{O}(N^2)$, but the computational efforts can be lowered by introducing a cutoff.
- Even with a Verlet list, the overall complexity is still $\mathcal{O}(N^2)$.
- Using cell linked-lists, one can reach an overall complexity of $\mathcal{O}(N)$, however for small systems, a Verlet list might still be more performant.
- Combining the cell linked-lists with a Verlet list even improves the performance.
- Partial updating of the neighbor list can improve performance, especially when systems are not homogeneous.
- Spatial sorting of the particle data improves locality and thus produces fewer cache misses.