8.808/8.308 IAP 2026 Recitation 8: Simulating many-body interactions off lattice

Jessica Metzger

jessmetz@mit.edu | Office hours: 1/9, 1/14, 1/20, 1/27 11am-12pm (8-320)

January 23, 2026

Let's consider an "off-lattice" system of many particles labeled $i = 1, 2, \ldots, N$ with coordinates $\mathbf{r}_i(t)$. The particles will be described by some Langevin dynamics, including interactions between the particles, which we would like to simulate. For example, consider active Brownian particles (ABPs) in 2 dimensions, which evolve as

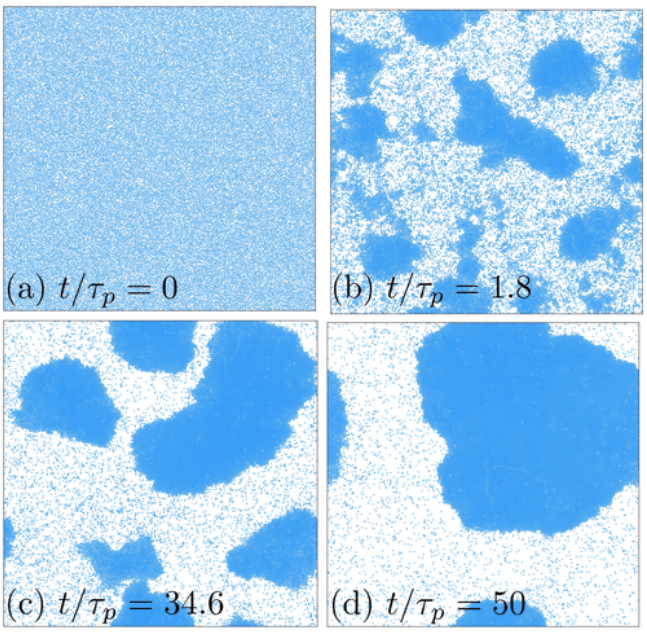
$$\dot{\mathbf{r}}_i(t) = v_0 \mathbf{u}[\theta_i(t)] - \sum_{j \neq i} \nabla U(\mathbf{r}_i - \mathbf{r}_j) \tag{1}$$

$$\dot{\theta}_i(t) = \sqrt{2D_r}\xi_i(t) \, , \tag{2}$$

where $\mathbf{u}[\theta_i] = \hat{x}\cos\theta_i + \hat{y}\sin\theta_i$ would be the orientation vector of particle i with orientation θ_i.

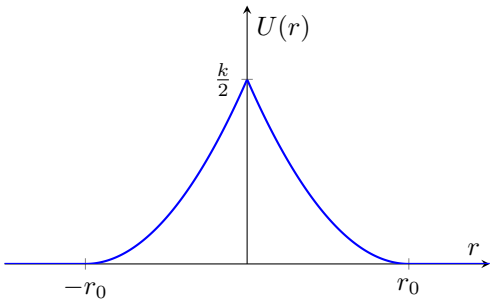
Simulating this system for a very large number of particles presents a problem. Because there are $N(N-1)/2$ pairs of particles, we may have to calculate the interaction force $N(N-1)/2$ times. This is too computationally expensive for the modern standard of simulating systems of $N \sim 10^5$. Even if each force calculation took only ~10 CPU cycles, or ~3 nanoseconds on average, calculating the force $N(N-1)/2$ times for a single timestep would take ~ 20 seconds. This makes studying the long-time behavior of the system impossible on reasonable timescales.

For example, the simulations shown on the right (from Hecht et al. arXiv:2102.13007) depict 39,200 ABPs with a timestep $dt = 5 \times 10^{-6}$, simulated up to a time $t = 50$. Thus, reaching the end required 10^7 timesteps. If each one took 20 seconds, the entire simulation would take over 6 years!



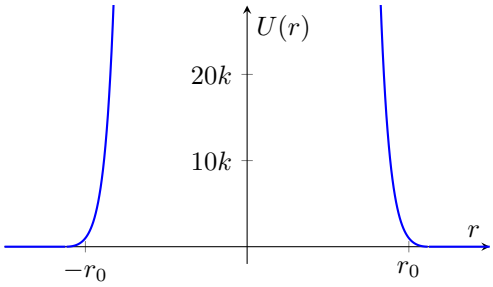
To remedy this, we can create a "lattice" that narrows down the number of pairs we have to calculate. In most realistic systems, the interaction forces have a finite radius. For example, we often use the purely repulsive harmonic potential

$$U(\mathbf{r}) = \begin{cases} \frac{k}{2}\left(1 - \frac{r}{r_0}\right)^2 , & |\mathbf{r}| < r_0 \\ 0 , & \text{otherwise} \end{cases} \tag{3}$$



along with the Weeks-Chandler-Anderson (WCA) potential

$$U(\mathbf{r}) = \begin{cases} 4k\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right] + k , & |\mathbf{r}| < 2^{1/6} r_0 \\ 0 , & \text{otherwise} \end{cases} \tag{4}$$



When $|\mathbf{r}|$ is larger than some cutoff (r_0 or $2^{1/6} r_0$), this is zero, and we don't need to worry about interactions between particles farther than this. We will now show how to simulate the dynamics accurately, without having to worry about the nonexistent interactions between faraway particles.

1 Spatial hashing setup

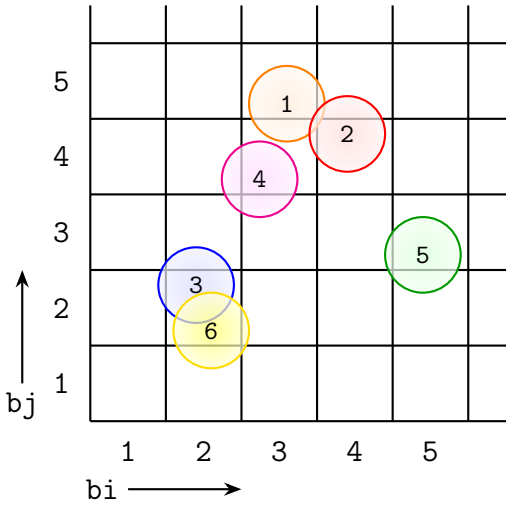
We will go over how to implement a "spatial hashing" algorithm. It creates a bookkeeping lattice that sorts particles and allows us to neglect faraway pairs, while leaving the physics of the simulation completely unchanged.

Start a simulation by initializing the particles and their states as usual. Once your particles have been placed, define a grid of boxes where the box width is the maximum interaction length, and do the following:

- Save the box indices `bi,bj` as attributes of each particle `i`
- Create a `boxes` array. In each entry `boxes[bi,bj]`, enter either
  - if empty, `boxes[bi,bj]=-1`
  - if not empty, `boxes[bi,bj]=i` for one particle `i` inside the box (doesn't matter which). This particle is now the first in the box.

  The particles in each box will be (arbitrarily) ordered, with `boxes[bi,bj]=i` being the first. This ordering will be stored in the `neighbors` list:
- Create a list `neighbors` which you will use to iterate the particles in each box. The entries of `neighbors` are as follows:
  - `neighbors[2i-1]` is the index of the particle before particle `i` in its box
  - `neighbors[2i]` is the index of the particle after particle `i` in its box
  - For a particle `i` which is the first in its box, `neighbors[2i-1]=-1`. (There is no one before it.)
  - For a particle `i` which is the last in its box, `neighbors[2i]=-1`. (There is no one after it.)

Here is an example configuration:



i	bi	bj	neighbors[2i-1]	neighbors[2i]
1	3	5	-1	-1
2	4	4	-1	-1
3	2	2	-1	6
4	3	4	-1	-1
5	5	3	-1	-1
6	2	2	3	-1

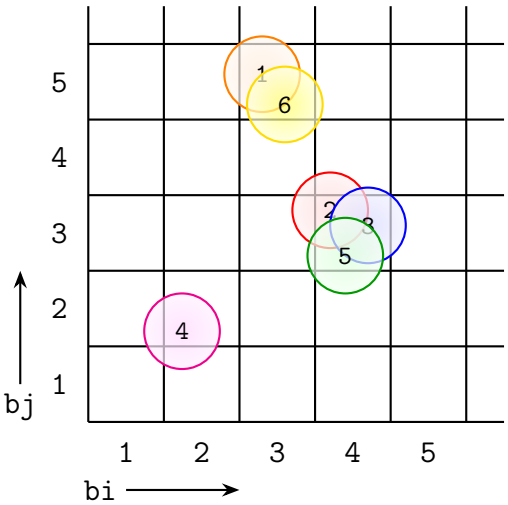
boxes

bj \ bi	1	2	3	4	5
5	-1	-1	1	-1	-1
4	-1	-1	4	2	-1
3	-1	-1	-1	-1	5
2	-1	3	-1	-1	-1
1	-1	-1	-1	-1	-1

neighbors

-1
-1
-1
-1
-1
6
-1
-1
-1
-1
3
-1

Another example:



i	bi	bj	neighbors[2i-1]	neighbors[2i]
1	3	5	6	-1
2	4	3	3	5
3	4	3	-1	2
4	2	2	-1	-1
5	4	3	2	-1
6	3	5	-1	1

neighbors

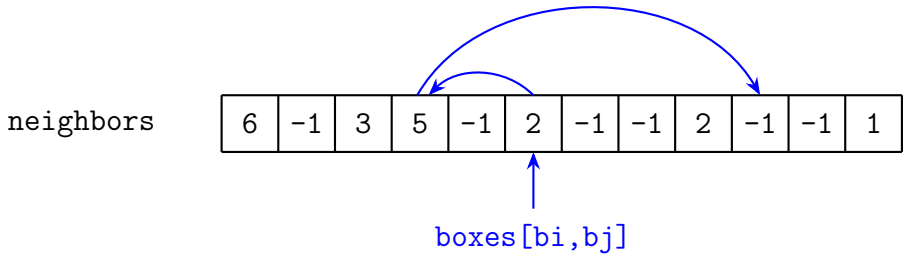
6
-1
3
5
-1
2
-1
-1
2
-1
-1
1

boxes

5	-1	-1	6	-1	-1
4	-1	-1	-1	-1	-1
3	-1	-1	-1	3	-1
2	-1	4	-1	-1	-1
1	-1	-1	-1	-1	-1

bj
1 2 3 4 5
bi

Iterating through the particles in a box is done through the `neighbors` list. Starting at some box, e.g. `(bi,bj)=(4,3)`, we will start with `boxes[bi,bj] = 3`, so particle 3. Next will be the particle `neighbors[2*3]=2`. Next will be the particle `neighbors[2*2]=5`. Finally, we will reach `neighbors[2*5]=-1`, which signifies we have reached the end of the box. This process looks like this:



We will now go over pseudocode that does this iteration, and uses it to calculate inter-particle forces.

2 Calculating forces

To calculate the forces, instead of iterating over particles, we iterate over boxes `[bi,bj]`. Here is pseudocode for how to calculate the forces:

```
for bi=1 to Nxbox, bj=1 to Nybox
    j = boxes[bi,bj]  // get first particle in (bi,bj)
    k = neighbors[2j] // get next particle after j in (bi,bj)

    // iterate through remaining particles in this box
    while k != -1

        // calculate forces. may be 0
        fx = force_x(particles[j].x, particles[k].x)
        fy = force_y(particles[j].y, particles[k].y)
        particles[j].force_x += fx
        particles[j].force_y += fy
        particles[k].force_x -= fx // action-reaction
        particles[k].force_y -= fy // action-reaction

        k = neighbors[2k] // move to next particle in this box
    end while

    // iterate through neighboring boxes (up, right, etc.)
    nbijs = [(bi,bj+1), (bi+1,bj+1), (bi+1,bj), (bi+1,bj-1)]
    for m=1 to 5
        nbi,nbj = nbijs[m]

        // start with first particle in box (nbi,nbj)
        k = boxes[nbi,nbj]

        while k != -1 // iterate through all particles in box (nbi,nbj)

            // calculate forces. May be 0
            fx = force_x(particles[j].x, particles[k].x)
            fy = force_y(particles[j].y, particles[k].y)
            particles[j].force_x += fx
            particles[j].force_y += fy
            particles[k].force_x -= fx // action-reaction
            particles[k].force_y -= fy // action-reaction

            k = neighbors[2k] // move to next particle in (nbi,nbj)
        end while
    end for

    j = neighbors[2j] // move to the next particle in box (bi,bj)
end for
```

We would also need to account for boundary conditions:

- If we have closed boundary conditions, some boxes won't have neighboring boxes. This requires some if/then statements in the `for l=1 to 5`... loop.
- If we have periodic boundary conditions, locating neighboring boxes is more complicated than simply doing `(bi,bj+1), (bi+1,bj+1),` etc. We will have to use modular arithmetic.
  Moreover, when calculating distances inside the `force_x` and `force_y` functions, we will have to adjust based on periodic boundary conditions.

3 Moving particles

After we have calculated forces between particles, we can simply increment the positions according to these forces

```
for i=1 to N
    particles[i].x += particles[i].force_x * dt
    particles[i].y += particles[i].force_y * dt

    // implement other parts of the dynamics (e.g. thermal noise, activity, ...)
end for
```

There is then a possibility that some particles have moved to other boxes. To take care of this, we check the box indices `(bi,bj)` of each particle and compare it to the previous one. If it is different, then we move the particle as follows:

```
for i=1 to N
    newbi = int(particles[i].x / box_width)
    newbj = int(particles[i].y / box_width)
    if newbi != bi or newbj != bj
        remove_from_box(i,bi,bj,boxes,neighbors)
        add_to_box(i,newbi,newbj,boxes,neighbors)
        particles[i].bi = newbi
        particles[i].bj = newbj
    end if
end for
```

where we must define functions `remove_from_box` and `add_to_box`, something like:

```
function remove_from_box(i, bi, bj, boxes, neighbors)
    next = neighbors[2i]        // who was "after" i in box (bi,bj)

    if boxes[bi,bj]==i          // if i was the first in its box
        boxes[bi,bj]=next       // now the particle after i is the first in the box
        if next != -1           // if "next" is a particle, there is no no one before it
            neighbors[2next-1] = -1
        end if
    else                        // if i wasn't the first particle in its box
        prev = neighbors[2i-1]  // who was "before" i
        neighbors[2prev] = next // link i's previous particle to its next one
        if next != -1           // if "next" is a particle, now "prev" is before it
            neighbors[2next-1] = prev
        end
    end
end
```

That is, we make sure the particle before and after `i` are linked together in `neighbors`, and that the first and last particles in the box appropriately map to `-1`.

```
function add_in_box(i, bi, bj, boxes, neighbors)
    k = boxes[bi,bj]       // k is currently the first particle in box (bi,bj)
    boxes[bi,bj] = i       // now, i is the first particle in box (bi,bj)
    neighbors[2i-1] = -1   // there is now nobody before i
    neighbors[2i] = k      // now, k is after i
    if k != -1             // if k is a particle, i is now before it
        neighbors[2k-1] = i
    end if
end function
```

In summary, we put particle `i` at the front of box `bi,bj`, and demote the previous first particle to be following it.

This is the end of the timestep. After incrementing time (and calculating any desired observables), you can repeat from the beginning ("calculating forces", Sec. 2).

4 Tasks

If you want to get an off-lattice code running, you can do the following tasks:

1. Convince yourself that the algorithm for calculating forces (Sec. 2) doesn't change the physics, relative to a naïve double-for-loop

2. Implement an off-lattice interacting system with a naïve double-for-loop scheme for calculating forces, i.e.

```
// naive double-for-loop algorithm
for j=1 to N
    for k=1 to j
        // calculate force (may be zero)
        fx = force_x(particles[j].x, particles[k].x)
        fy = force_y(particles[j].y, particles[k].y)
        particles[j].force_x += fx
        particles[j].force_y += fy
        particles[k].force_x -= fx  // action-reaction
        particles[k].force_y -= fy  // action-reaction
    end for
end for
```

   and see how slow it gets when N is large. (This code is much more simple and less prone to bugs than the spatial hashing one, and is good for validating your spatial hashing code, so this step is important.)

3. Implement the spatial hashing algorithm (Sec. 2-3). Simulate some small systems ($N \sim 10$ to 100) and make sure the dynamics look the same whether you use spatial hashing or the naïve double-for-loop algorithm. Check that some observables (e.g. spatial correlations, mean-squared-displacements, etc.) look the same for the two algorithms.