Hands-on: Molecular Dynamics

Content

- MD Visualization

- Analyse MD data
  - Property 1
  - Property 2
  - Property 3

- Produce MD data

MD Visualization

Run:

```
vmd trajectory.xyz
```

MD Visualization

Run:

```
vmd trajectory.xyz
```

Configuration:

Graphics/Representations…/Drawing Method/VDW

MD Visualization

Run:

vmd trajectory.xyz

Configuration:

Graphics/Representations…/Drawing Method/VDW

Edit:

Sphere scale

MD Visualization

Run:

vmd trajectory.xyz

Configuration:

Graphics/Representations…/Drawing Method/VDW

Edit:

Sphere scale

Movie:

- MD Visualization

- Analyse MD data
    - Property 1
    - Property 2
    - Property 3

- Produce MD data

- MD Visualization

- Analyse MD data
    - Property 1
    - Property 2
    - Property 3

- Produce MD data

Questions

- Is the material a solid, liquid, or gas?

Questions

- Is the material a solid, liquid, or gas?
- Given the positions of every atom at each time step, what analyses would you perform to corroborate it?

Mean square displacement & Diffusion

- MSD

$$MSD(t) = \langle (x(0) - x(t))^2 \rangle$$

Mean square displacement & Diffusion

- MSD

$$MSD(t) = \langle (x(0) - x(t))^2 \rangle$$

- Diffusion Coefficient

$$6 D = d\, MSD(t) / dt$$

Mean square displacement & Diffusion

- MSD

$$MSD(t) = \langle (x(0) - x(t))^2 \rangle$$

- Diffusion Coefficient

$$6\,D = d\,MSD(t) / dt$$

- Problem:

Mean square displacement & Diffusion

- MSD

$$MSD(t) = \langle (x(0) - x(t))^2 \rangle$$

- Diffusion Coefficient

$$6 D = d\, MSD(t) / dt$$

- Problem: PBC

Mean square displacement & Diffusion

- MSD

$$MSD(t) = \langle (x(0) - x(t))^2 \rangle$$

- Diffusion Coefficient

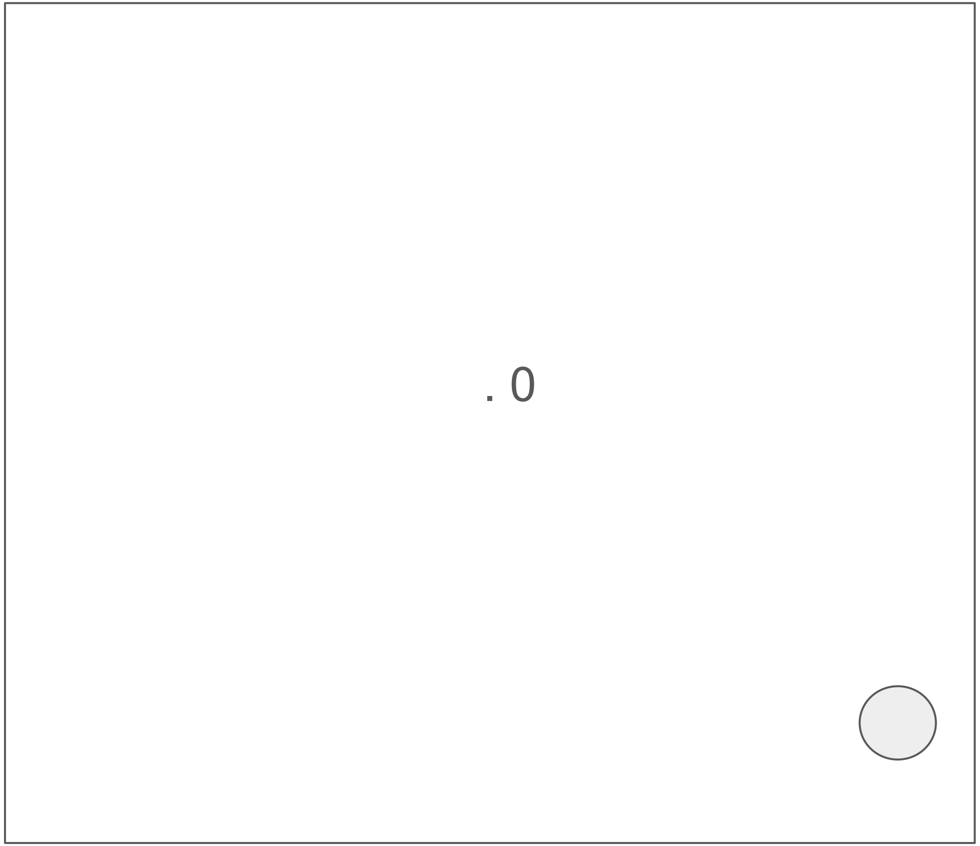
$$6 D = d\, MSD(t) / dt$$

- Problem: PBC

- Steps: 1) "unwrap" or "unfold" trajectories, 2) Compute MSD(t) of unfolded system

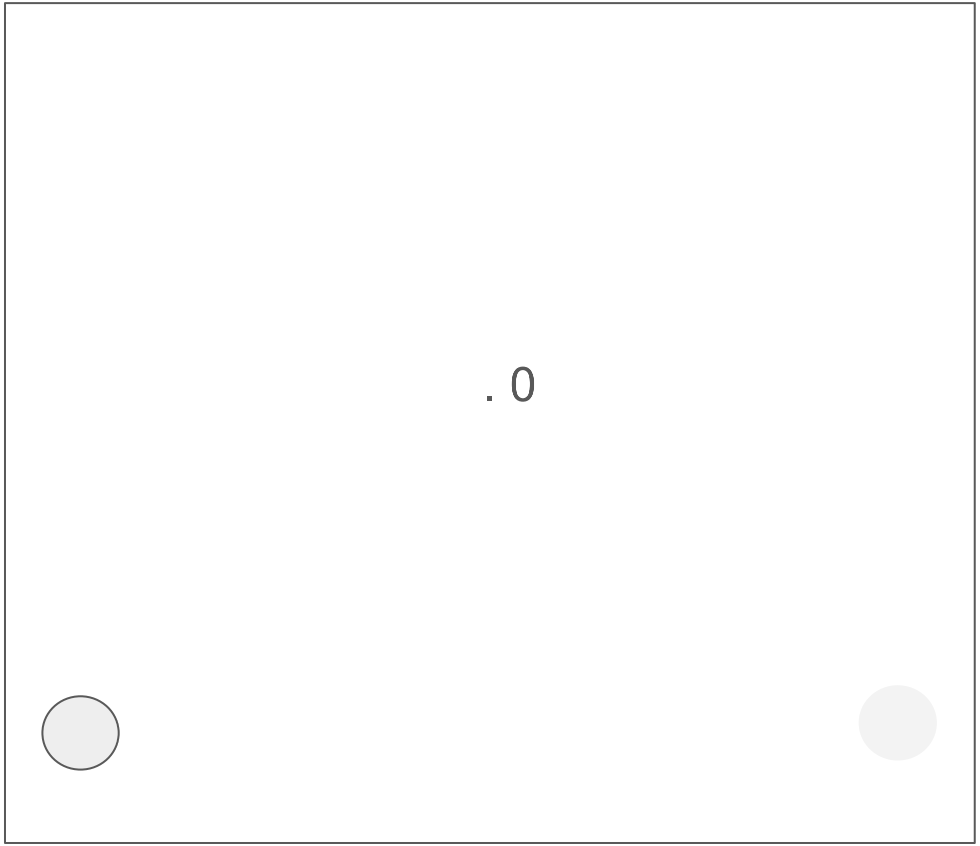
[0, 0]
jumps_cumulated

[0, 0]
current_jumps



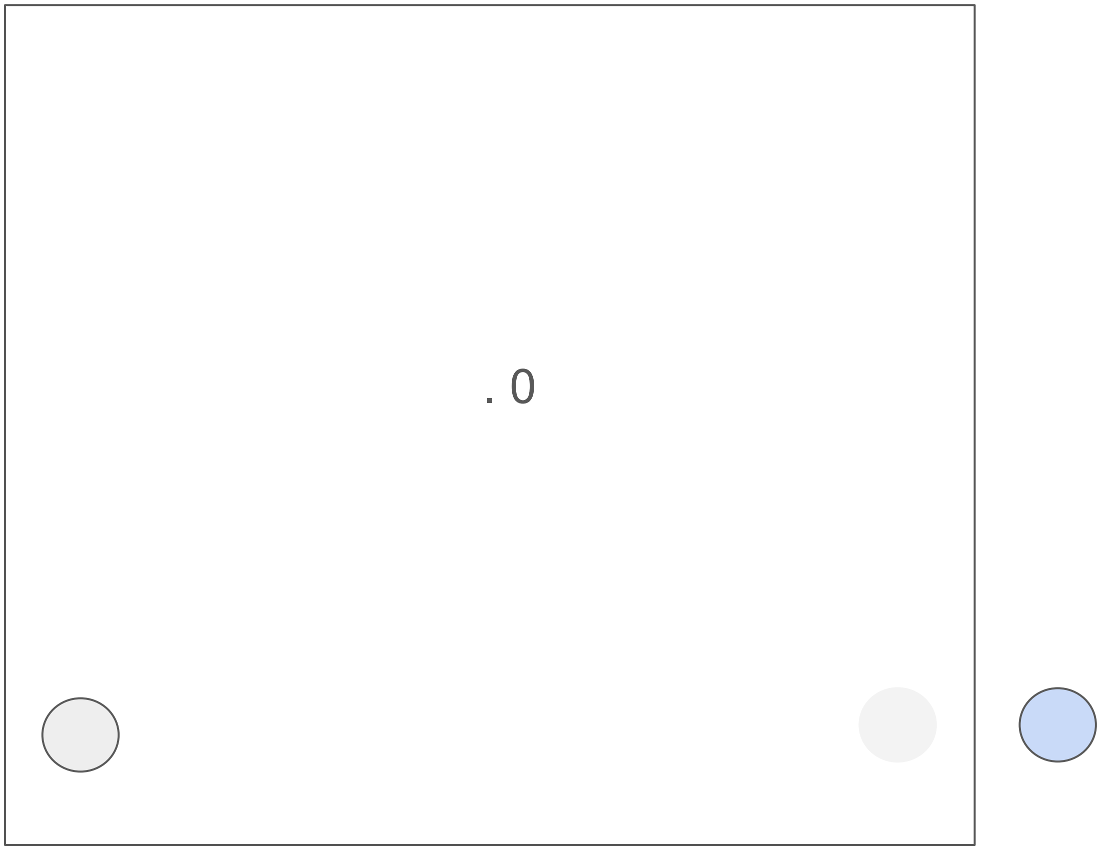
[1, 0]
jumps_cumulated

[1, 0]
current_jumps



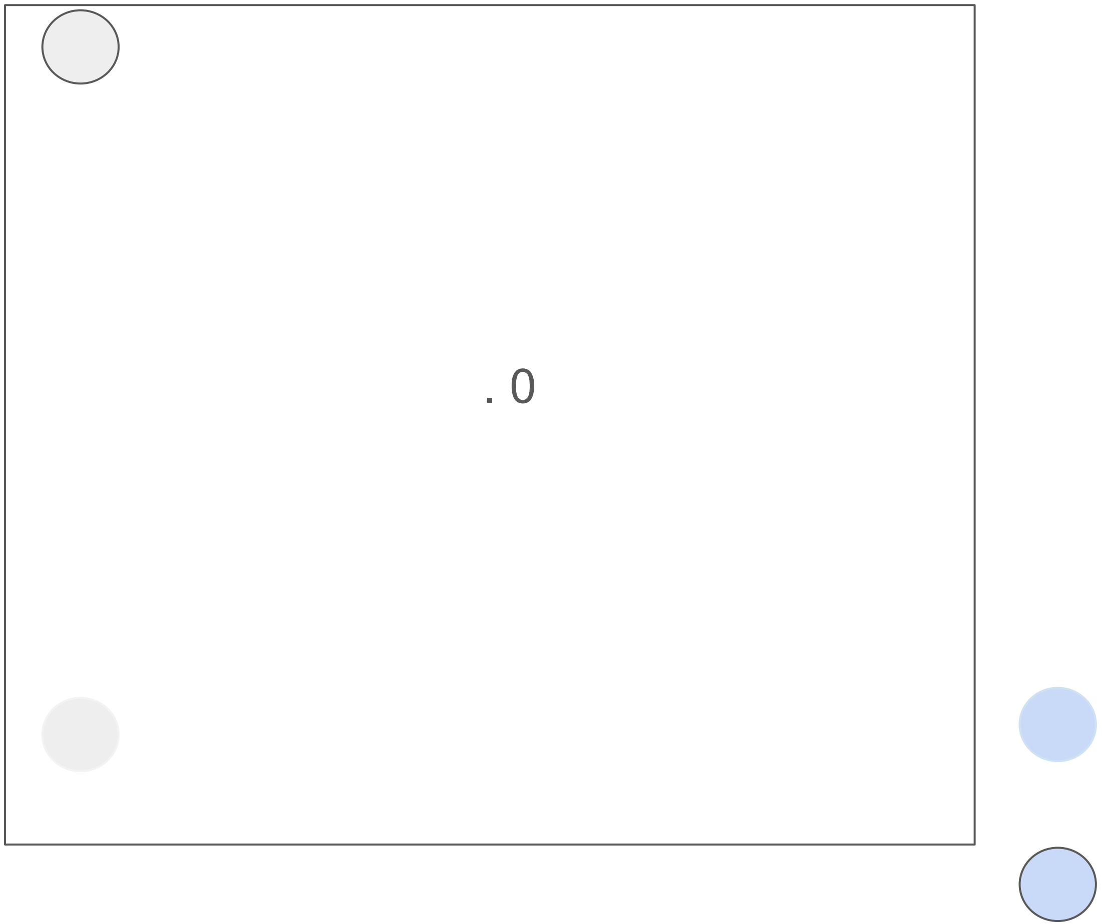
[1, 0]
jumps_cumulated

[1, 0]
current_jumps



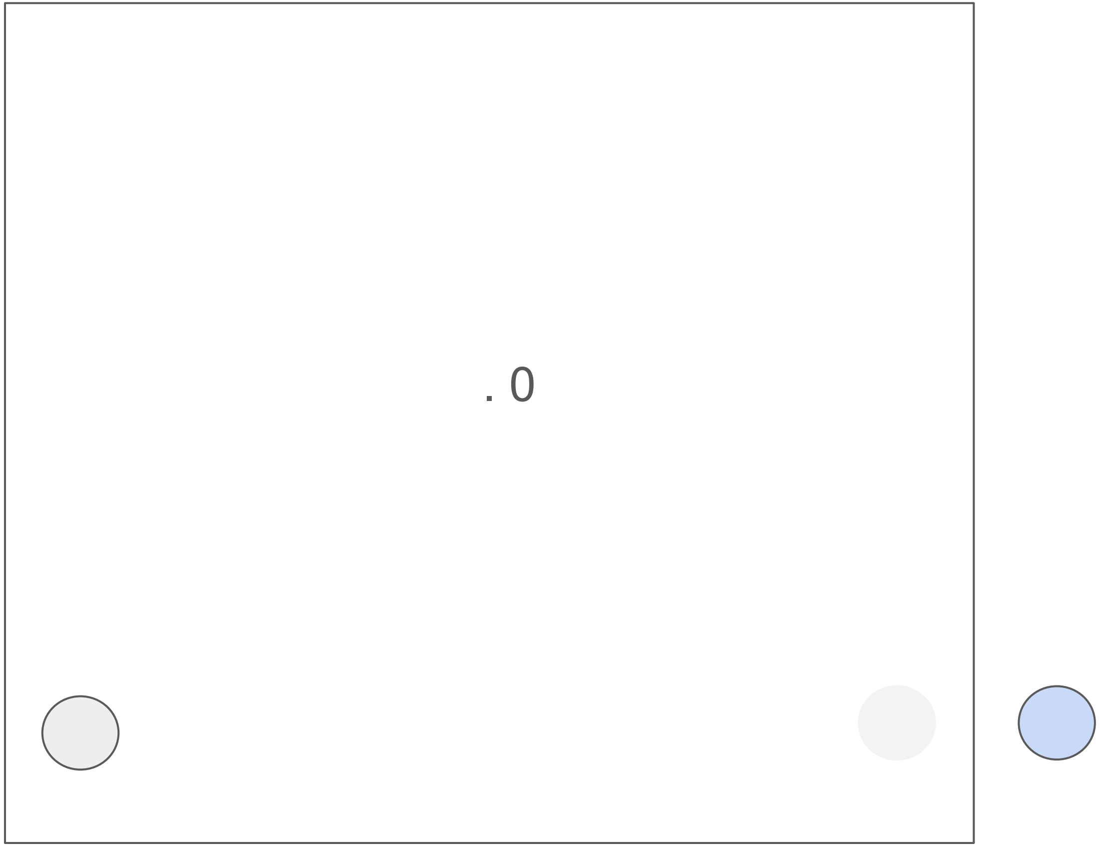
[1, -1]
jumps_cumulated

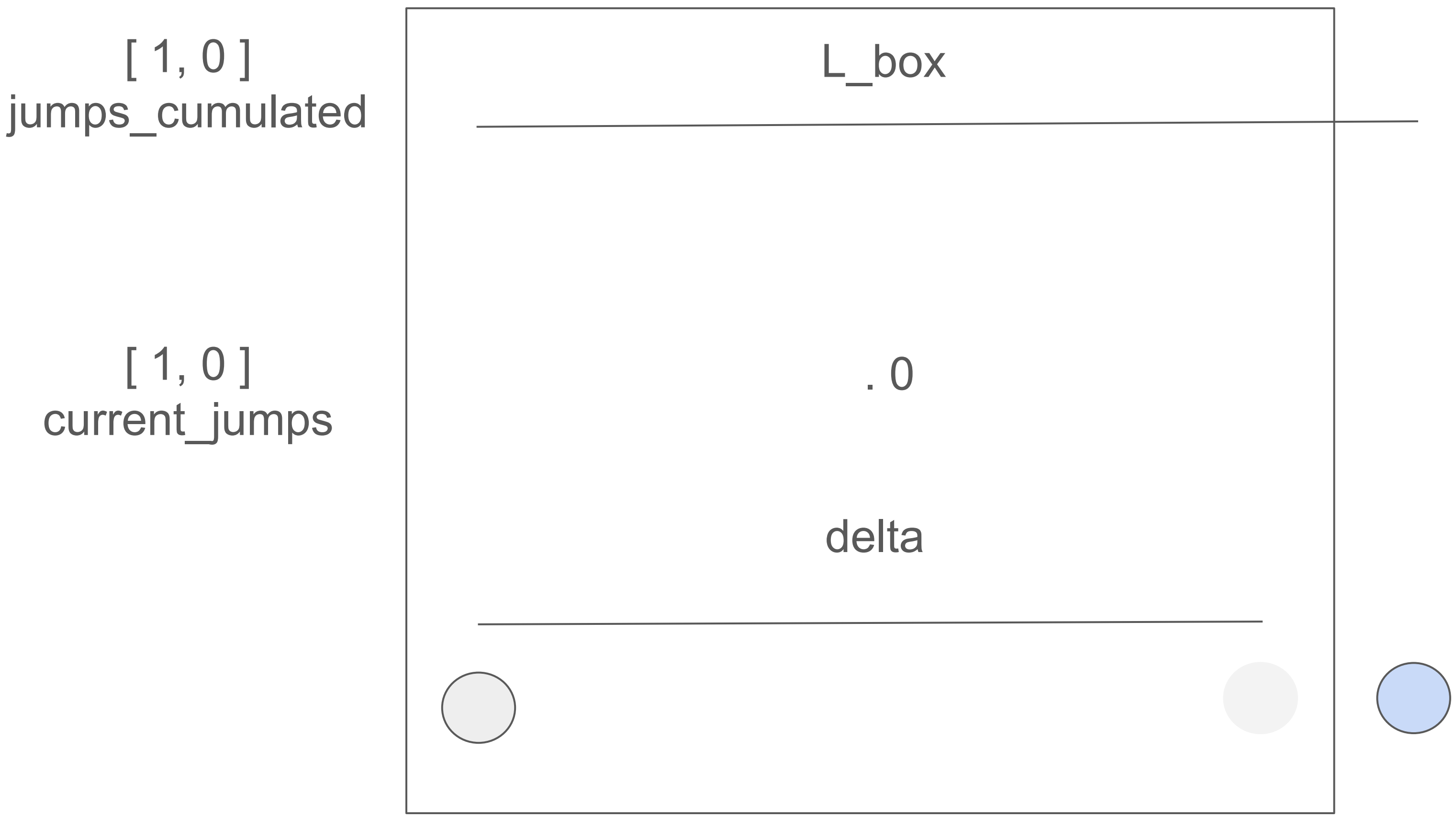
[0, -1]
current_jumps



[1, 0]
jumps_cumulated

[1, 0]
current_jumps

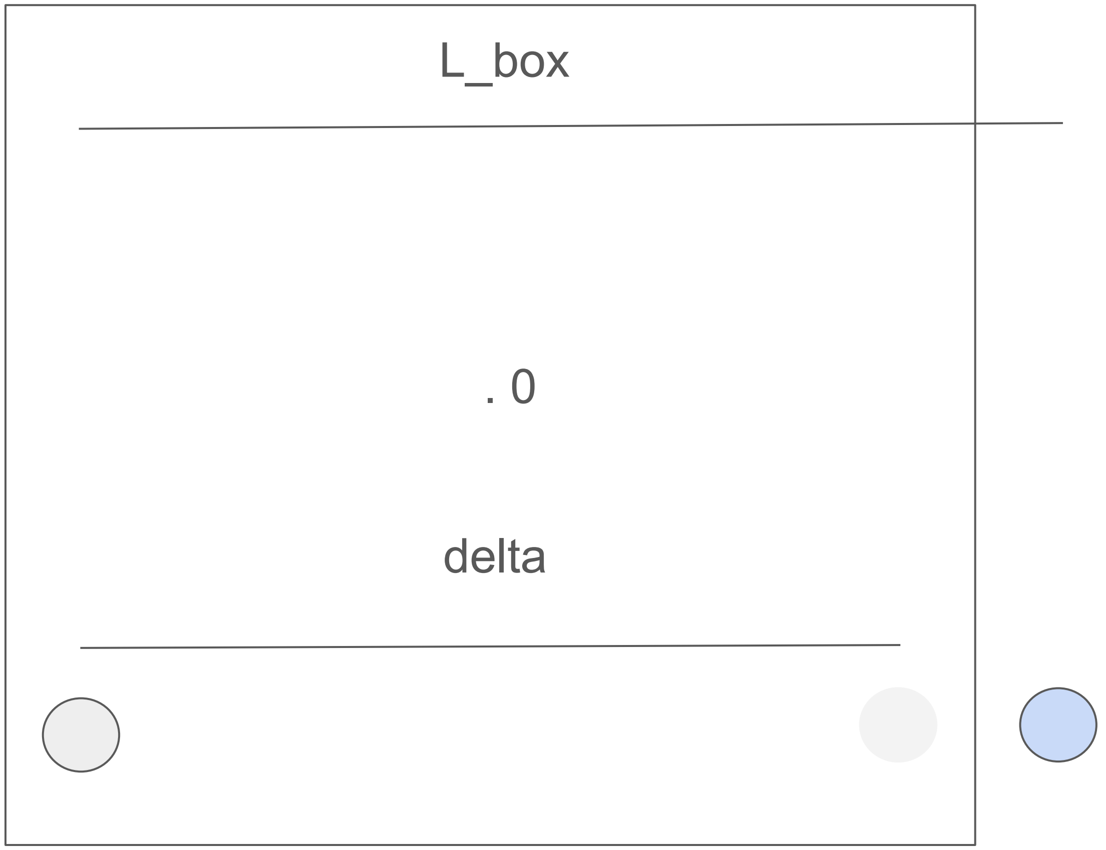



[1, 0]
jumps_cumulated
[1, 0]
current_jumps
L_box
. 0
delta

For all atoms!

[1, 0]
jumps_cumulated

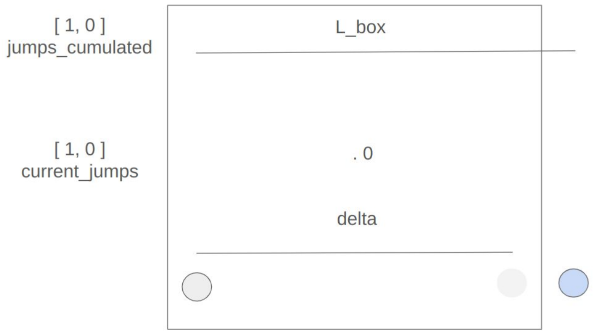
[1, 0]
current_jumps

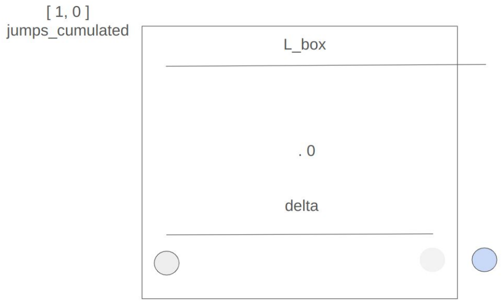


Unwrap trajectories

```
def unwrap_trajectory(trajectory_wrapped):
    '''
    Unwrap the trajectory!
    '''

    unwrapped = np.zeros_like(trajectory_wrapped)
    unwrapped[0] = trajectory_wrapped[0]

    jumps_cumulated = np.zeros((N_atoms, 3))  # counts boundary crossings

    for t in range(N_frames - 1):

        delta =                               # delta in real units

        current_jumps = np.rint(delta / L_box)


        # Update jumps_cumulated


        # Reconstruct next unwrapped position
        unwrapped[t + 1] =

    return unwrapped
```

For all atoms!





Unwrap trajectories: Solution

```
def unwrap_trajectory(trajectory_wrapped):
    '''
    Unwrap the trajectory!
    '''

    unwrapped = np.zeros_like(trajectory_wrapped)
    unwrapped[0] = trajectory_wrapped[0]

    jumps_cumulated = np.zeros((N_atoms, 3))  # counts boundary crossings

    for t in range(N_frames - 1):

        delta = trajectory_wrapped[t] - trajectory_wrapped[t+1] # delta in real units

        current_jumps = np.rint(delta / L_box)


        # Update jumps_cumulated
        jumps_cumulated += current_jumps

        # Reconstruct next unwrapped position
        unwrapped[t + 1] = trajectory_wrapped[t+1] + jumps_cumulated * L_box

    return unwrapped
```

Compute mean square displacement

- MSD

$$MSD(t) = < (x(0) - x(t)) \hat{} 2 >$$

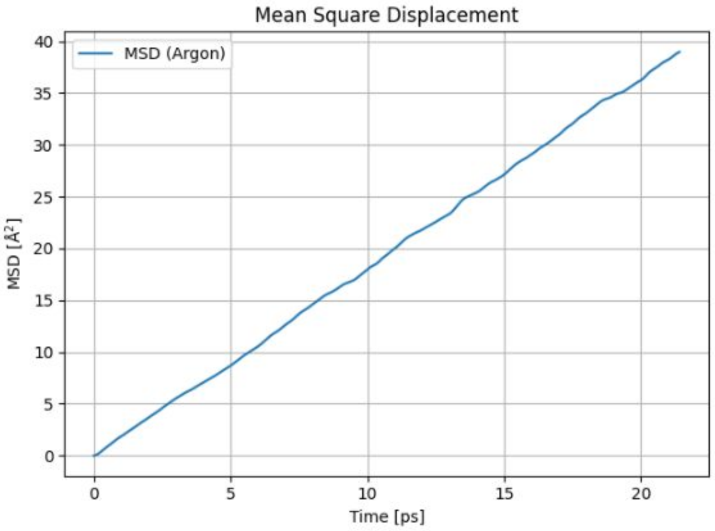
- For a given t:

$$\left.\begin{array}{l} (x\_1(0) - x\_1(t))\hat{}2 \\ (x\_2(0) - x\_2(t))\hat{}2 \\ (x\_3(0) - x\_3(t))\hat{}2 \\ (x\_4(0) - x\_4(t))\hat{}2 \end{array}\right\} \text{Average over particles}$$

```python
def compute_msd(unwrapped):
    '''
    Compute the mean squared displacement!
    '''
    r0 = unwrapped[0]  # initial positions

    displacements = unwrapped - r0  # displacement from initial position

    return msd
```

Compute mean square displacement: Solution

```
def compute_msd(unwrapped):

    r0 = unwrapped[0]  # initial positions

    displacements = unwrapped - r0  # displacement from initial position

    squared_displacements = np.sum(displacements**2, axis=2)  # shape: (n_frames, n_atoms)

    msd = np.mean(squared_displacements, axis=1)  # average over atoms

    return msd
```



- MD Visualization
- Analyse MD data
  - Property 1
  - Property 2
  - Property 3
- Produce MD data

- MD Visualization ✓

- Analyse MD data
  - Property 1: MSD & Diffusion
  - Property 2
  - Property 3

- Produce MD data

- MD Visualization
- Analyse MD data
  - Property 1: MSD & Diffusion
  - Property 2
  - Property 3
- Produce MD data

Question

- What if you only have the positions at a single time frame?

Radial Distribution Function

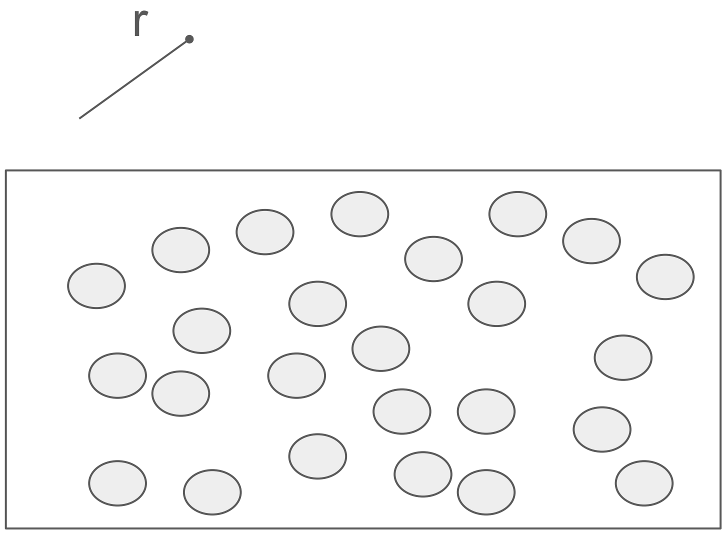
- Definition

$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$

Radial Distribution Function

- Definition

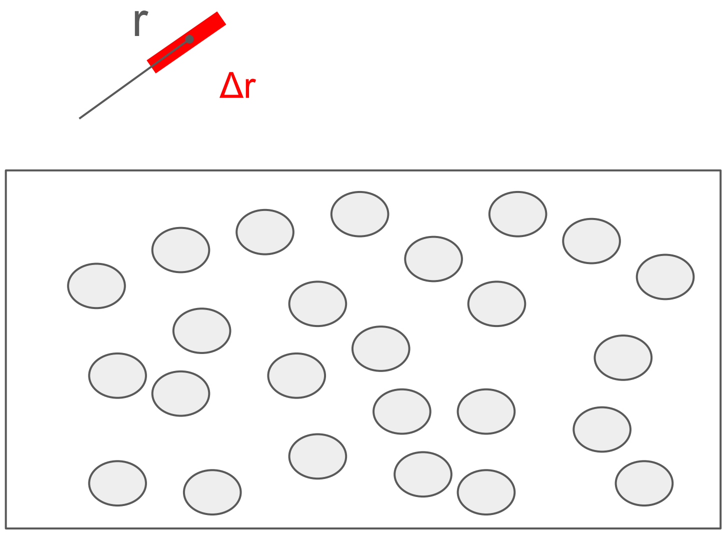
$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$



Radial Distribution Function

- Definition

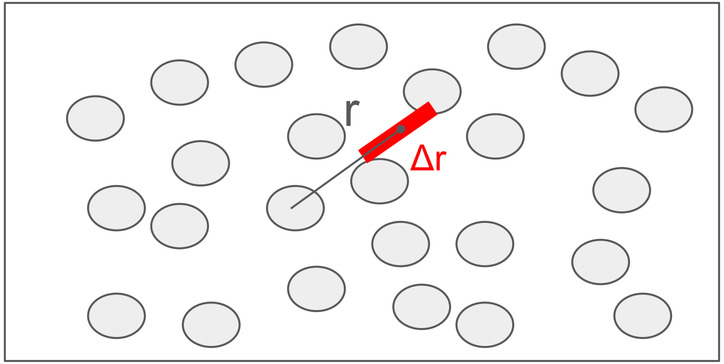
$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$



Radial Distribution Function

- Definition

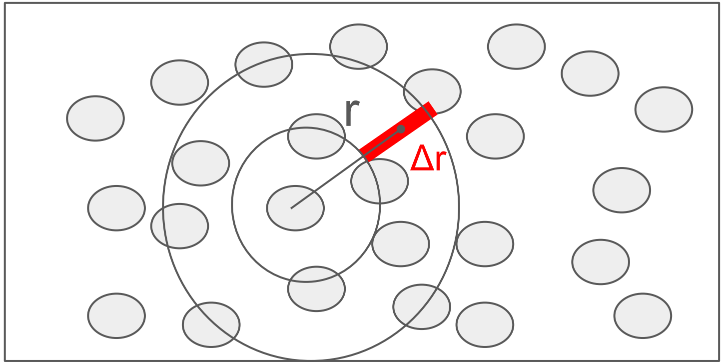
$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$



Radial Distribution Function

- Definition

$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$

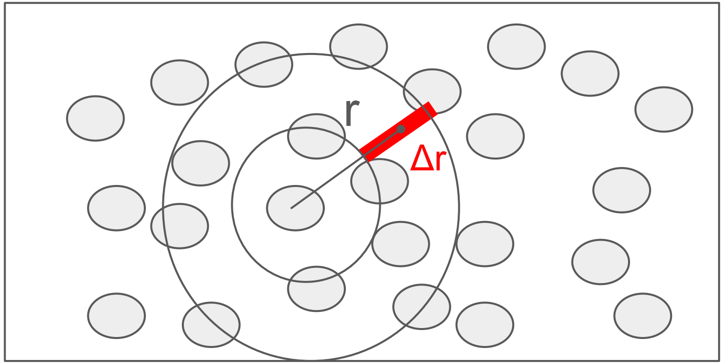


Radial Distribution Function

- Definition

$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$

- Steps
  1. Counting
  2. Normalization

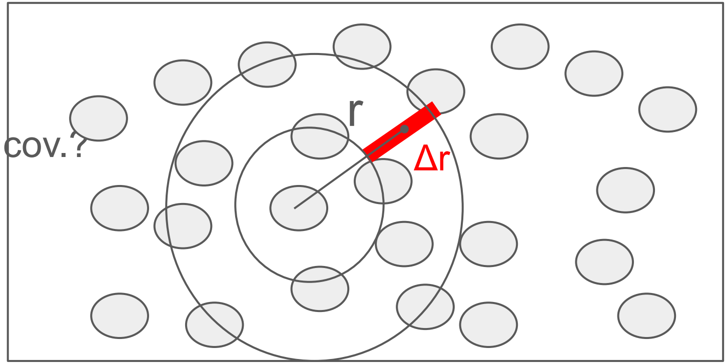


Radial Distribution Function

- Definition

$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$

- Steps
  1. Counting
     a. How to consider the min. image cov.?
  2. Normalization

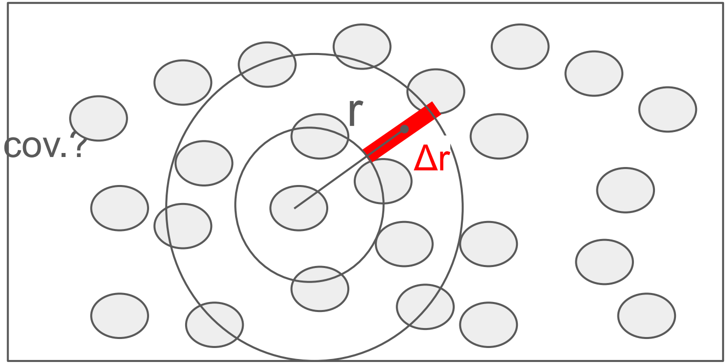


Radial Distribution Function

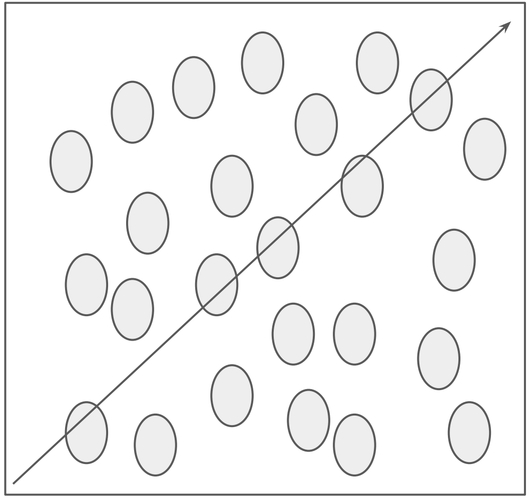
- Definition

$$\rho g(\mathbf{r}) = \frac{1}{N}\left\langle \sum_{i}^{N} \sum_{j \neq i}^{N} \delta[\mathbf{r} - \mathbf{r}_{ij}] \right\rangle$$

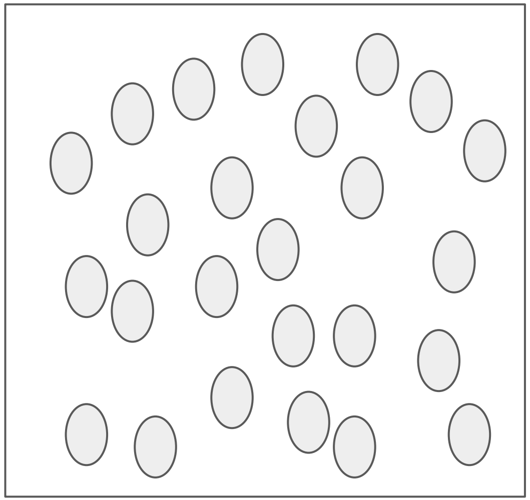
- Steps
  1. Counting
     a. How to consider the min. image cov.?
  2. Normalization
     a. How to consider Δr ?



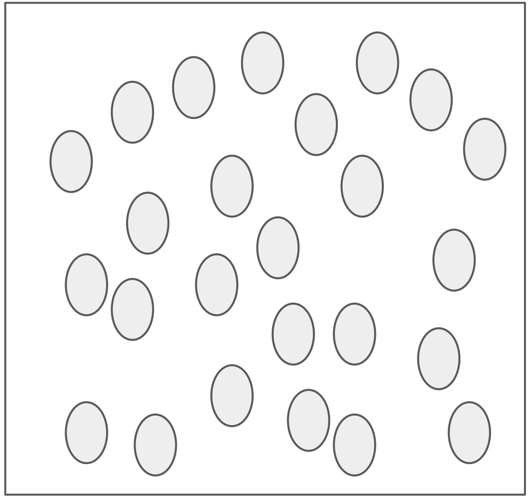
Counting



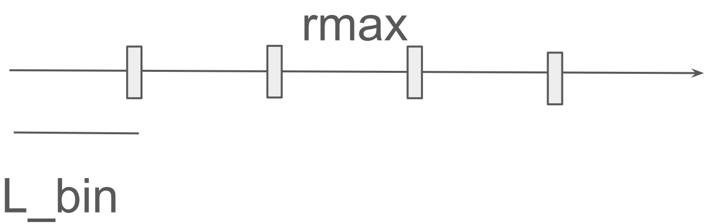
Counting



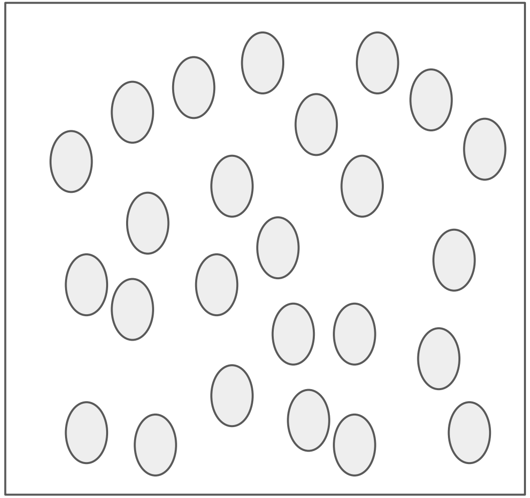
Counting



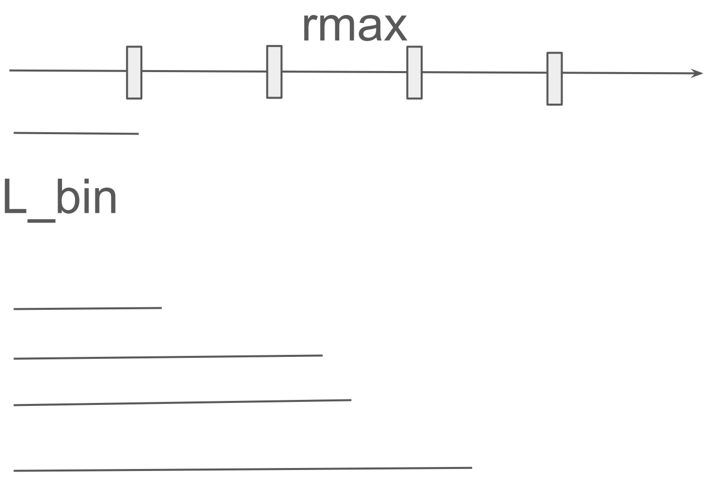
N_bins = 5



Counting

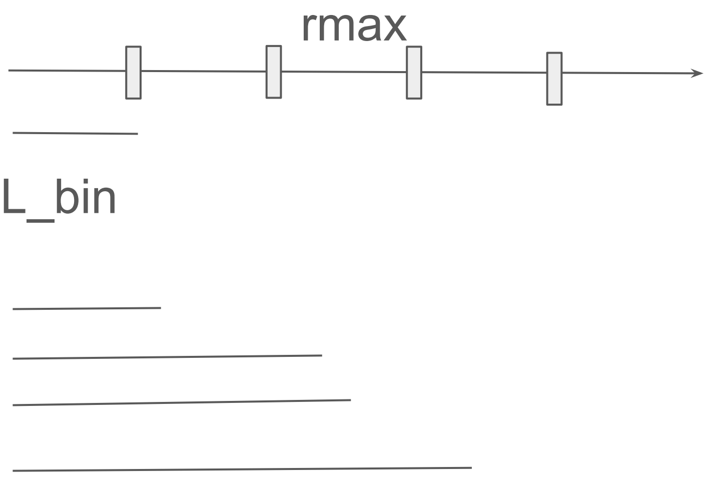


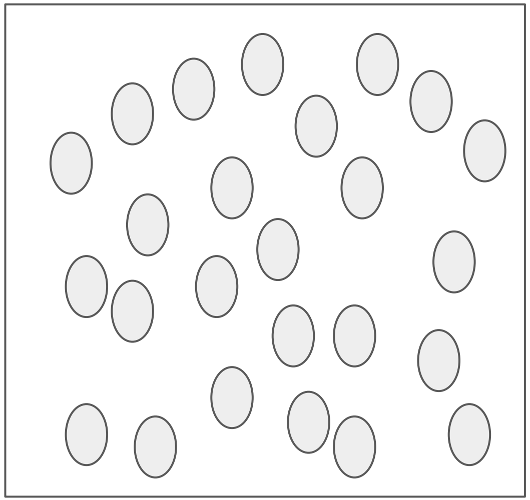
N_bins = 5



Counting

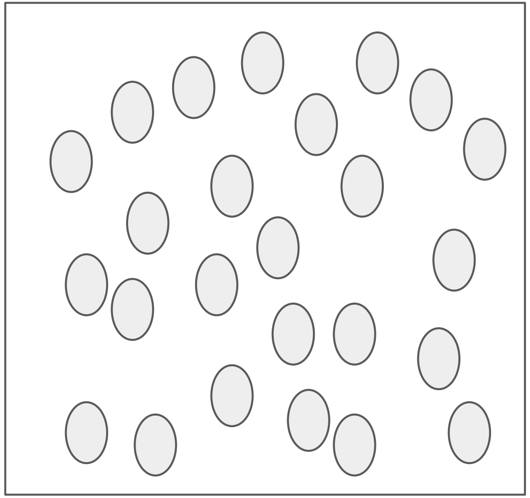
N_bins = 5



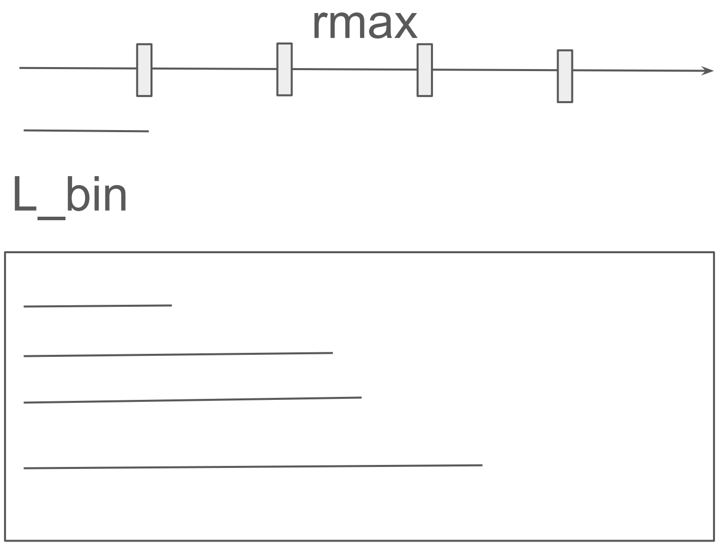


g_count = [0, 1, 2, 1, 0]

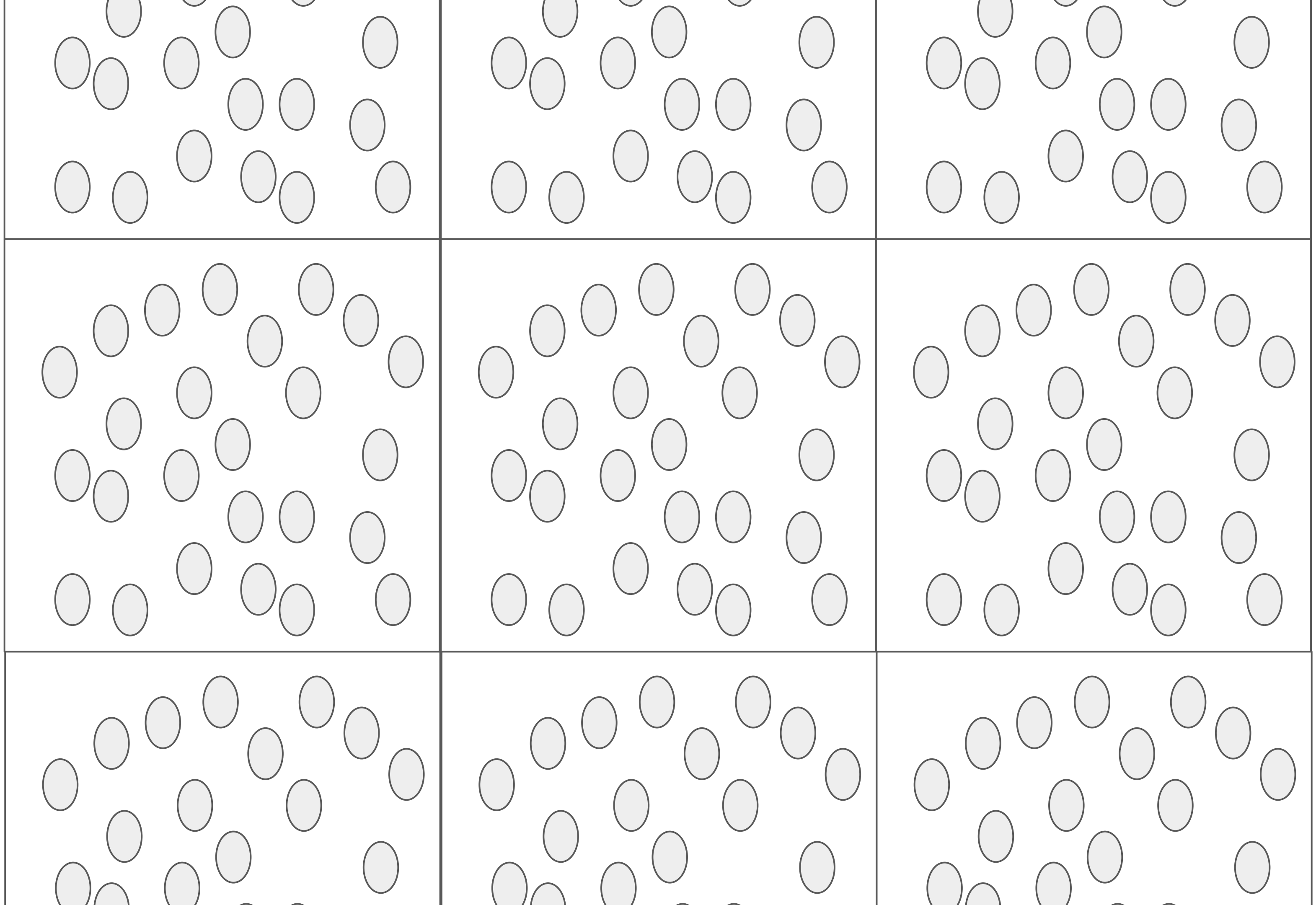
Counting

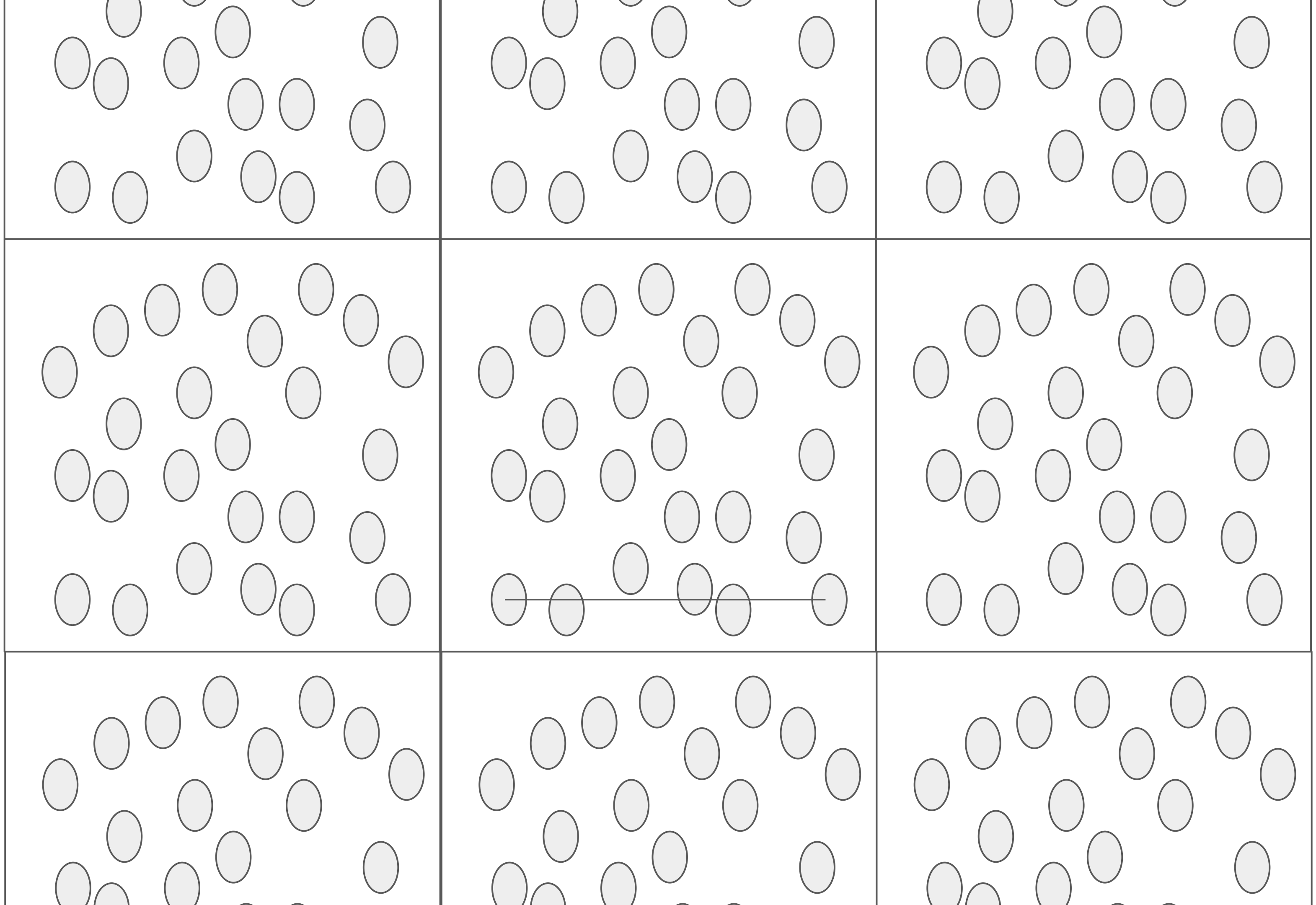


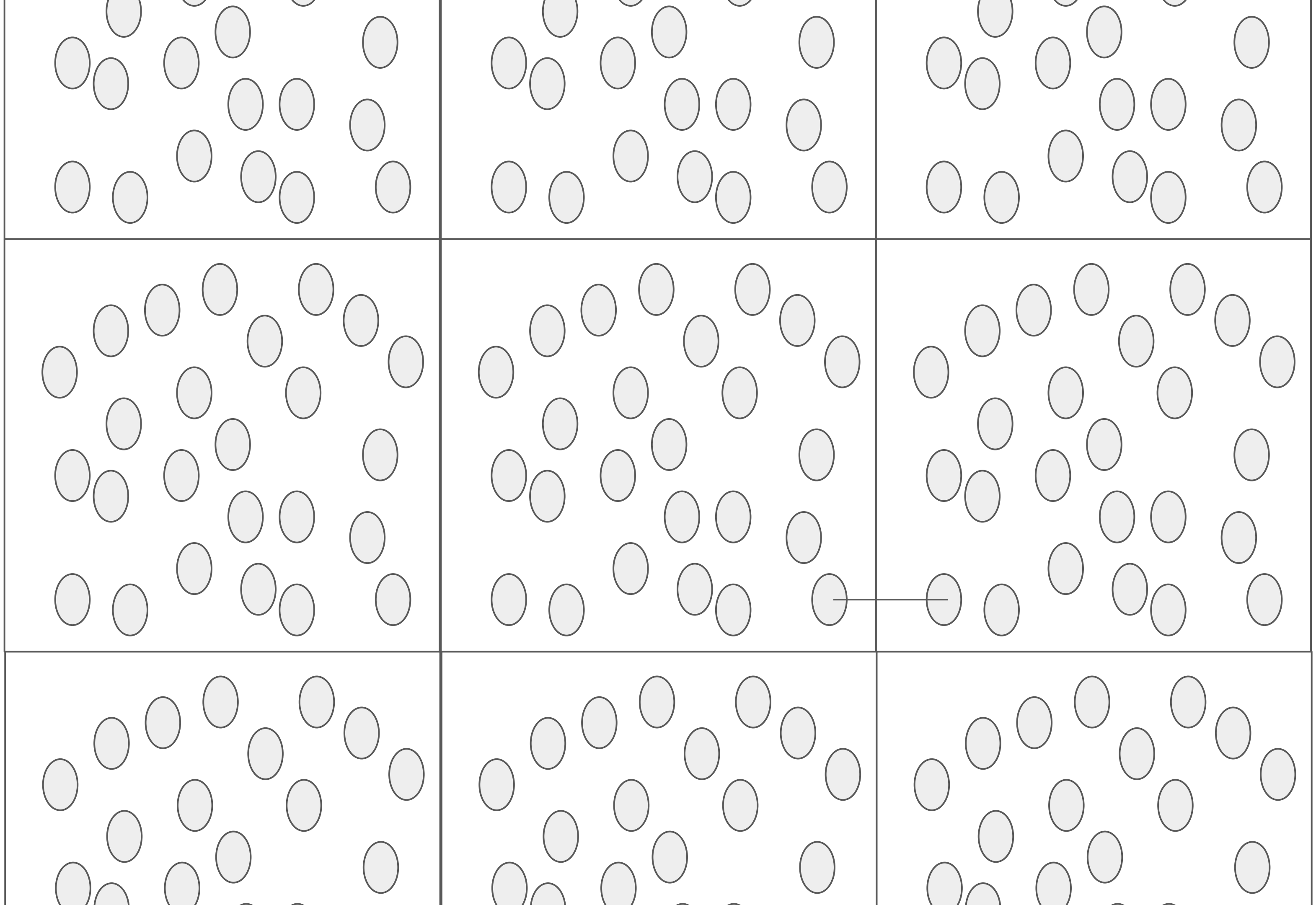
N_bins = 5

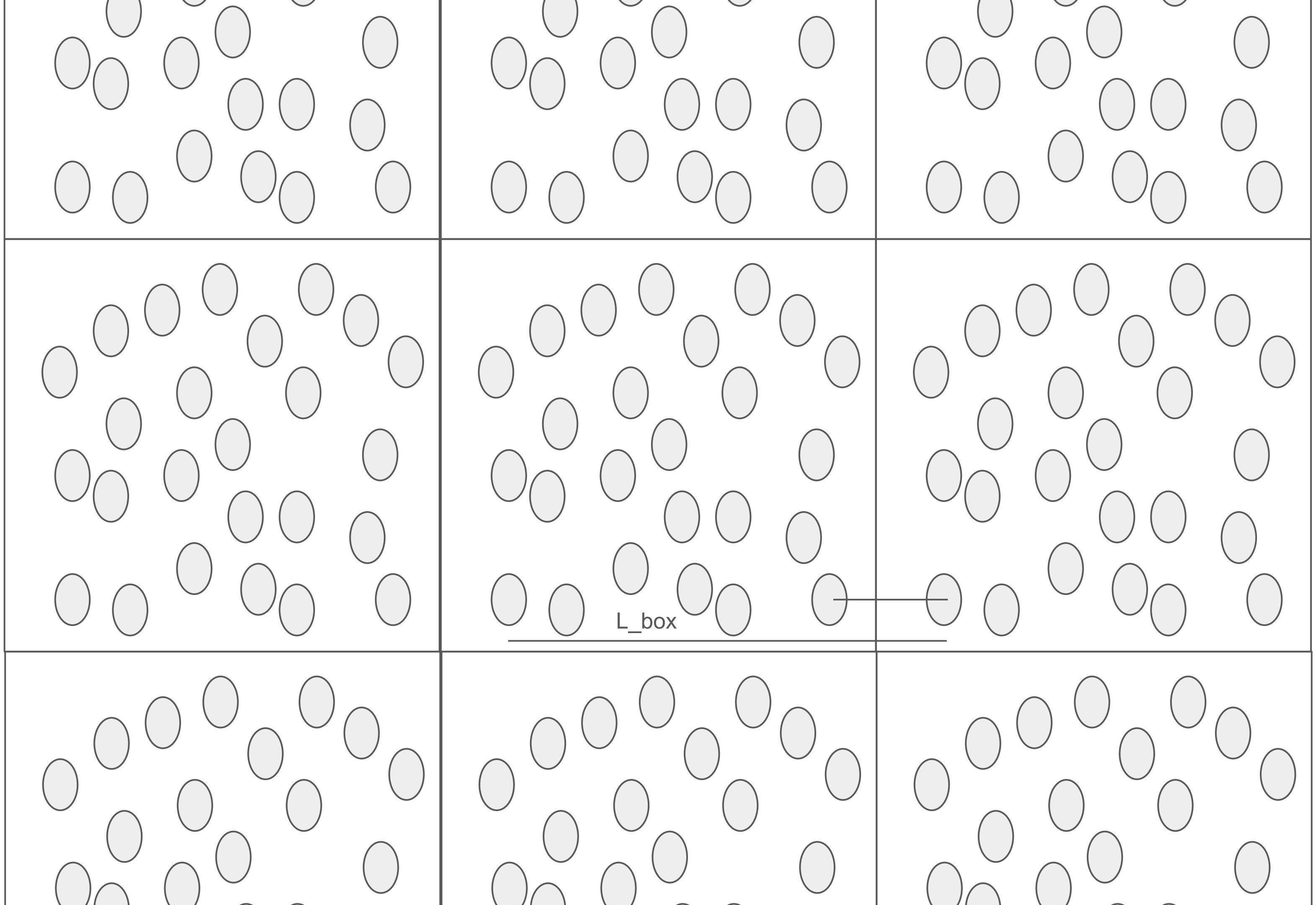


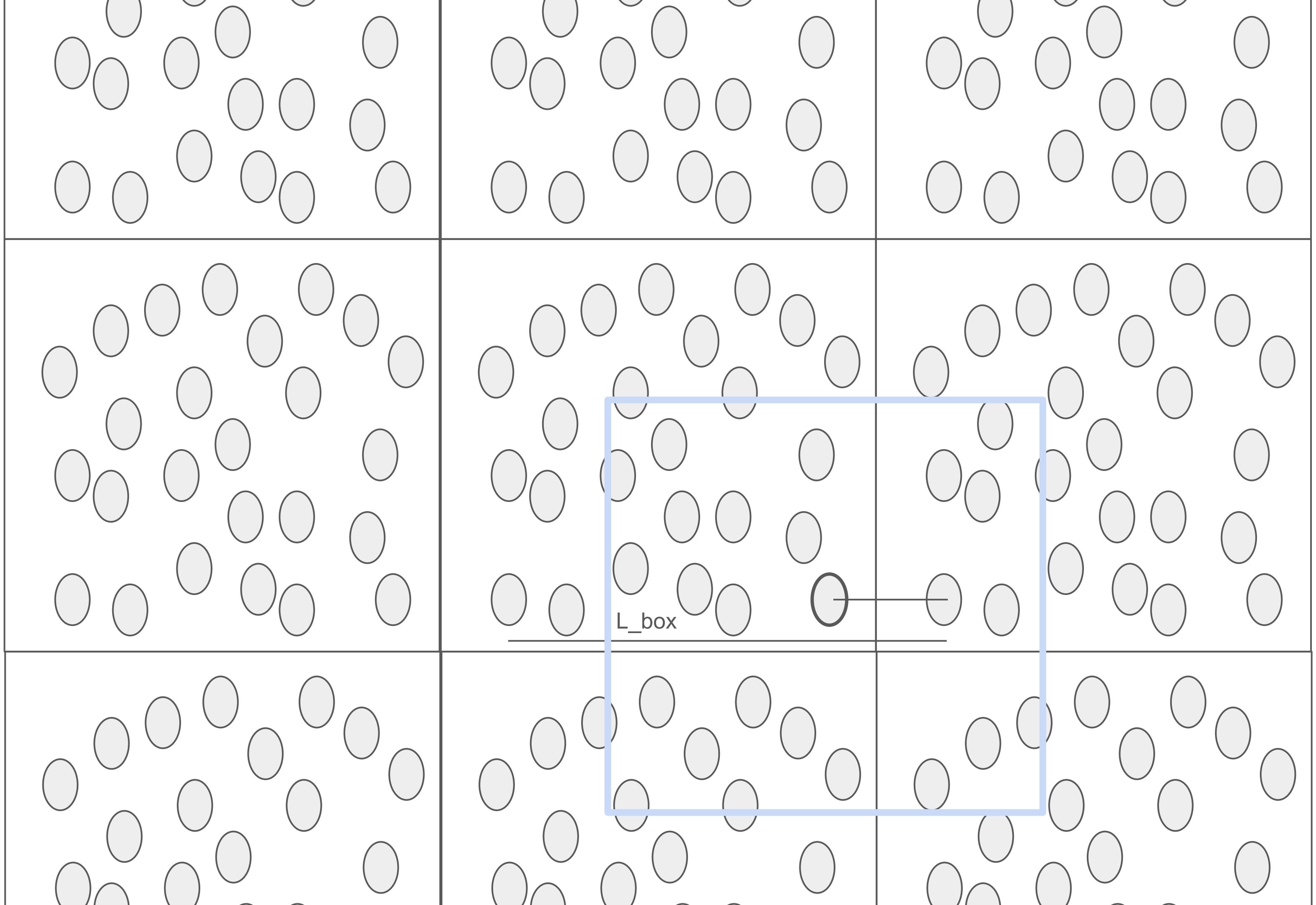
g_count = [0, 1, 2, 1, 0]



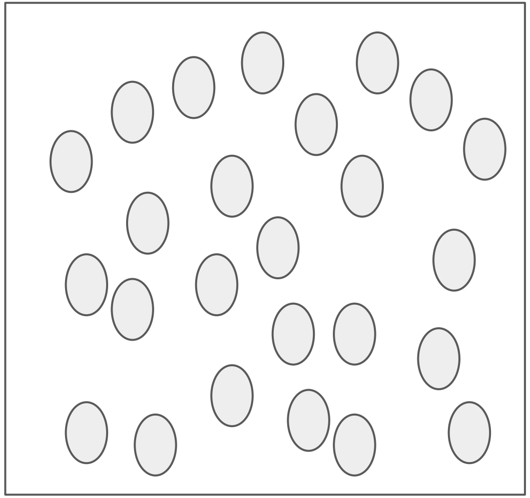




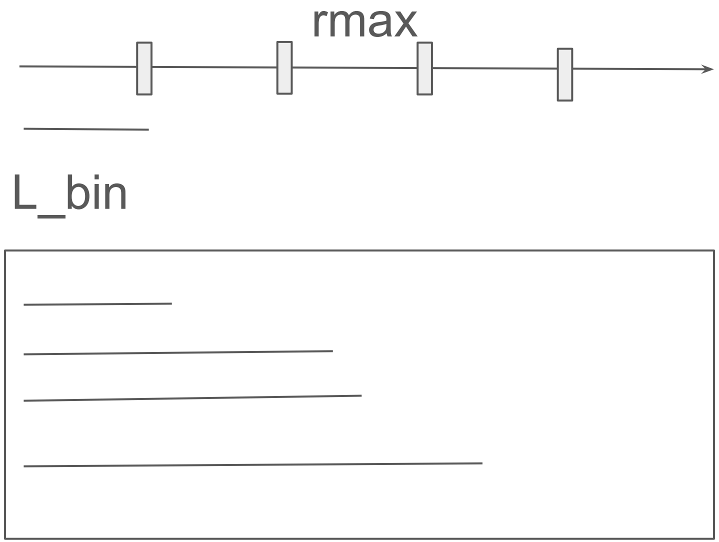

L_box


L_box

Counting

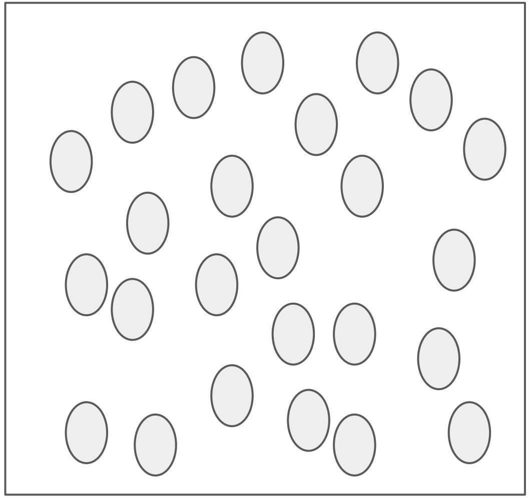


N_bins = 5

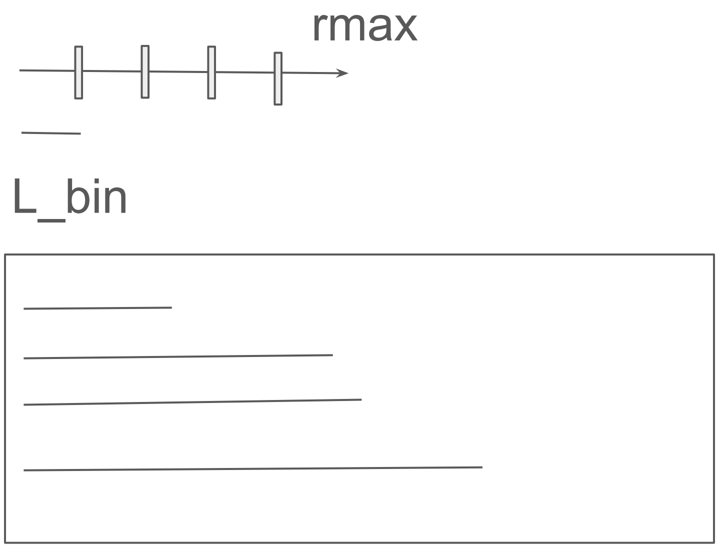


g_count = [0, 1, 2, 1, 0]

Counting

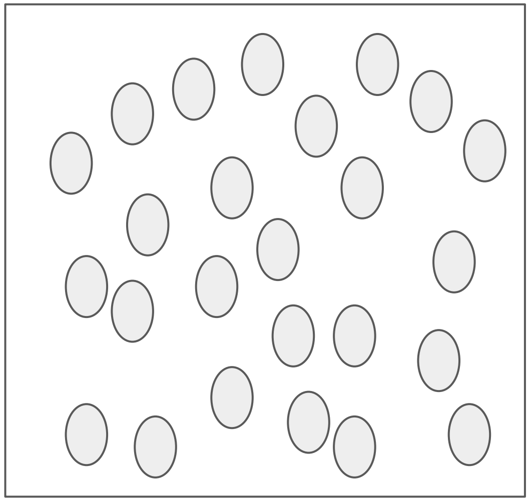


N_bins = 5

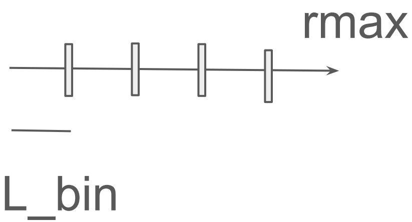


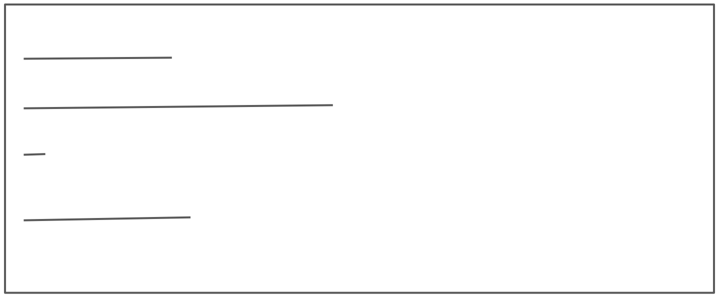
g_count = [0,   1,   2,   1,   0]

Counting



N_bins = 5

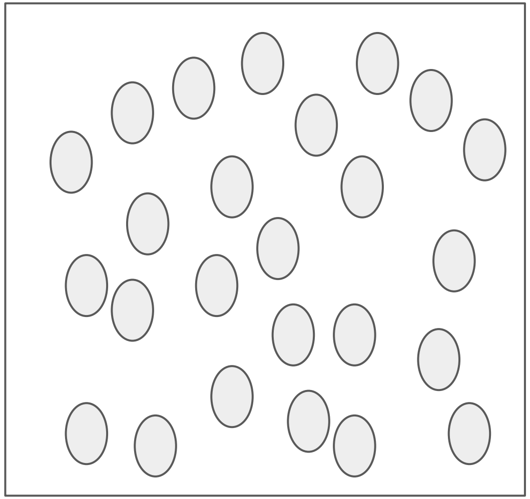


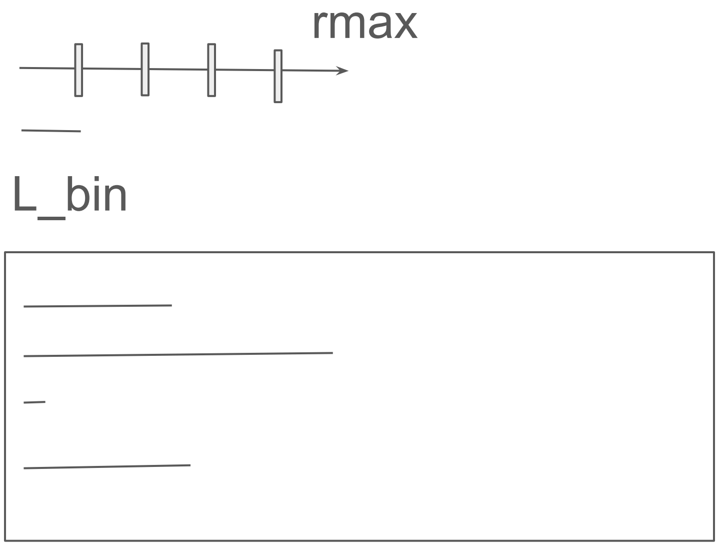


g_count = [0, 1, 2, 1, 0]

Counting

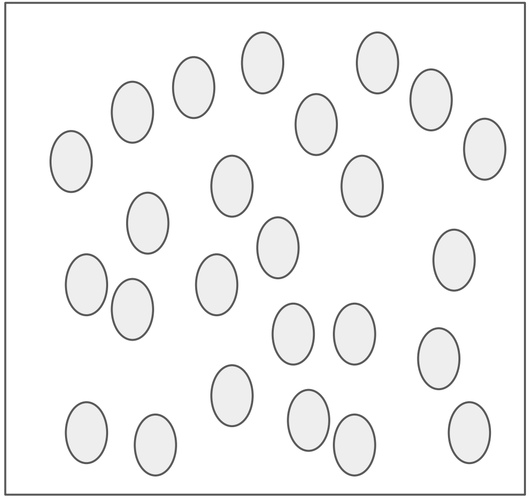
N_bins = 5



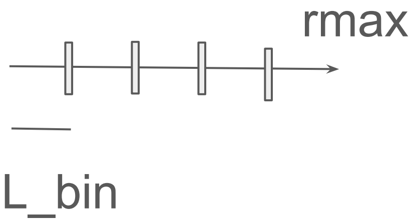


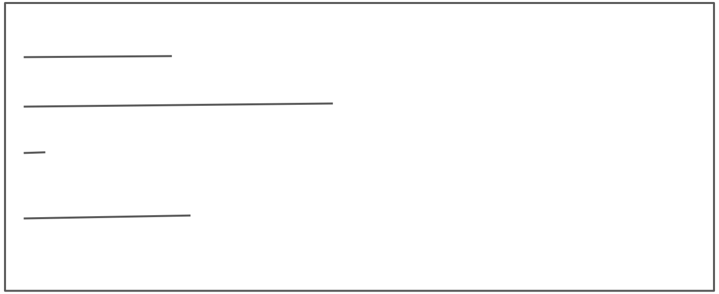
g_count = [1, 0, 2, 0, 1]

Counting



N_bins = 5





g_count = [1, 0, 2, 0, 1]

Counting

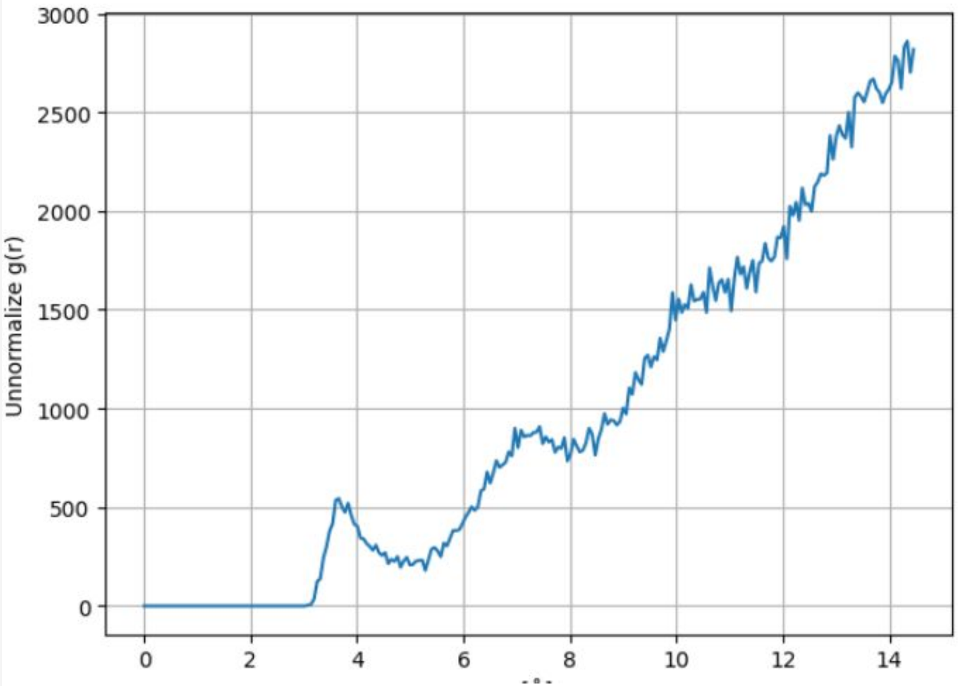
Hint: If dx is normalized,
dx = dx - np.rint(dx)
applies the minimum image convention
in normalized units (why?)

```
N_bins = 512
rmax = np.sqrt(3*L_box**2)/2
L_bin=rmax/N_bins
g_counter=np.zeros(N_bins)
```

```
def counting_distances_frame(i):
    '''
    Add the distances to g_counter corresponding to the i-th frame
    '''
    rx = Trajectory[i][:,0];
    ry = Trajectory[i][:,1];
    rz = Trajectory[i][:,2];

    for k in range(N_atoms-1):
        j=k+1

        # Distances to atoms with superior index (not normalized)
        dx = (rx[k]-rx[j:N_atoms])
        dy = (ry[k]-ry[j:N_atoms])
        dz = (rz[k]-rz[j:N_atoms])

        # Apply minimum image convention to dx, dy and dz



        # dx, dy and dz already with the minimum image convention in real units.
        r2 = dx*dx + dy*dy + dz*dz
        r = np.sqrt(r2)

        for corrected_distance in r:
            g_counter[?] += 2    # Find the expression for ?
```

Counting: Solution

```python
def counting_distances_frame(i):
    '''
    Add the distances to g_counter corresponding to the i-th frame
    '''
    rx = Trajectory[i][:,0];
    ry = Trajectory[i][:,1];
    rz = Trajectory[i][:,2];

    for k in range(N_atoms-1):
        j=k+1

        # Distances to atoms with superior index (not normalized)
        dx = (rx[k]-rx[j:N_atoms])
        dy = (ry[k]-ry[j:N_atoms])
        dz = (rz[k]-rz[j:N_atoms])

        # Apply minimum image convention to dx, dy and dz
        dx = dx/L_box
        dy = dy/L_box
        dz = dz/L_box

        dx = dx - np.rint(dx)
        dy = dy - np.rint(dy)
        dz = dz - np.rint(dz)

        dx = dx * L_box
        dy = dy * L_box
        dz = dz * L_box

        # dx, dy and dz already with the minimum image convention in real units.
        r2 = dx*dx + dy*dy + dz*dz
        r = np.sqrt(r2)

        for corrected_distance in r:
            g_counter[int(corrected_distance/L_bin)] += 2    # Find the expression for ?
```



Normalization

- 2 Types

Normalization

- 2 Types
    1. r & Δr independent
    2. r & Δr dependent

Normalization

- 2 Types
  1. r & Δr independent
     - Averages -> N_atoms*N_frames
     - Definition -> (N_atoms/L_box**3) Density

     norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)
  2. r & Δr dependent

Normalization

- 2 Types
    1. r & Δr independent
        - Averages -> N_atoms*N_frames
        - Definition -> (N_atoms/L_box**3)

Density

norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)

    2. r & Δr dependent
        - Volumen of the shells

Normalization

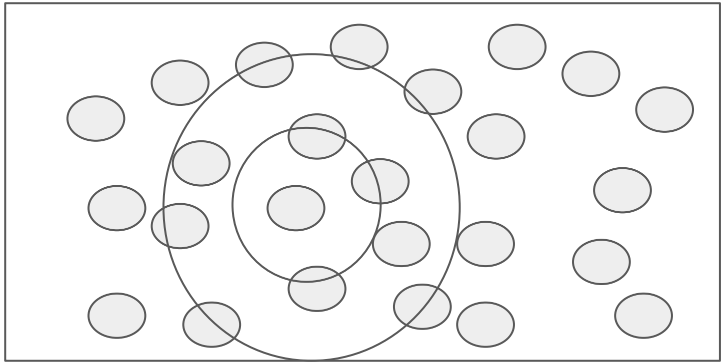
- 2 Types
  1. r & Δr independent
     - Averages -> N_atoms*N_frames
     - Definition -> (N_atoms/L_box**3)

Density

norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)

  2. r & Δr dependent
     - Volumen of the shells

g_count = [10, 65]



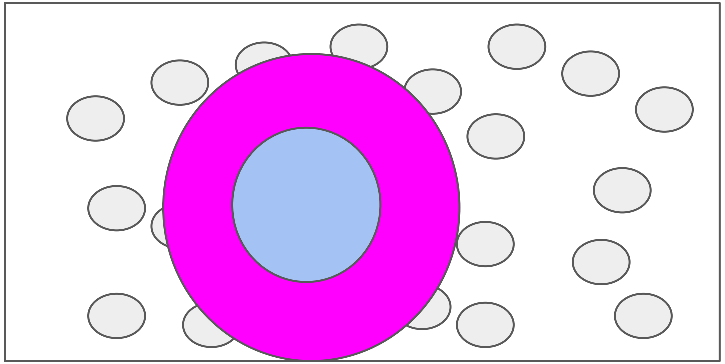
Normalization

- 2 Types
    1. r & Δr independent
        - Averages -> N_atoms*N_frames
        - Definition -> (N_atoms/L_box**3)

Density

norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)

    2. r & Δr dependent
        - Volumen of the shells



g_count = [10, 65]

g_normalization = [vol,vol]*norm_factor

Normalization

g_r =g_count/g_normalization

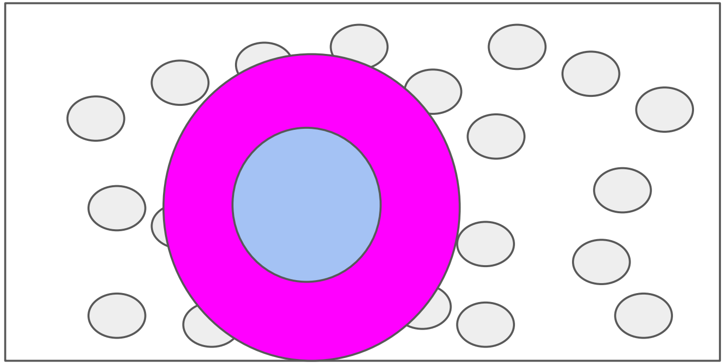
- 2 Types

    1. r & Δr independent
        - Averages -> N_atoms*N_frames
        - Definition -> (N_atoms/L_box**3)

Density

norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)

    2. r & Δr dependent
        - Volumen of the shells

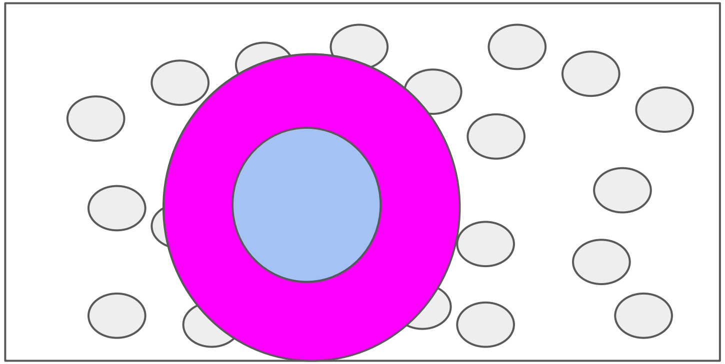


g_count = [10, 65]

g_normalization = [vol,vol]*norm_factor

Normalization

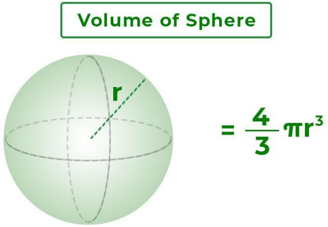
```
g_normalization = np.zeros(N_bins)
for i in range(N_bins):
    g_normalization =                # Comput the normalization such that g(r) = g_counter/g_normalization
```



norm_factor = N_atoms*N_frames*(N_atoms/L_box**3)

g_normalization = [vol,vol]*norm_factor

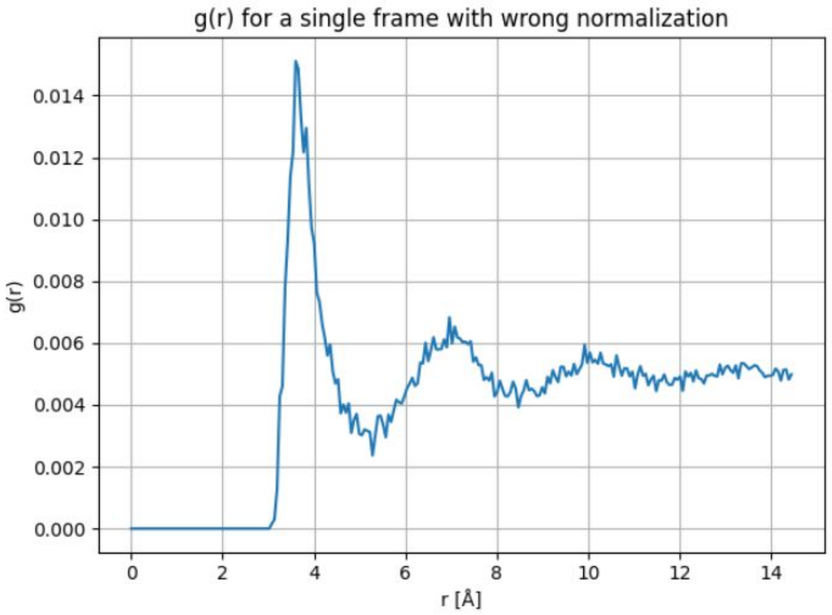
Hint: The volume of a sphere is

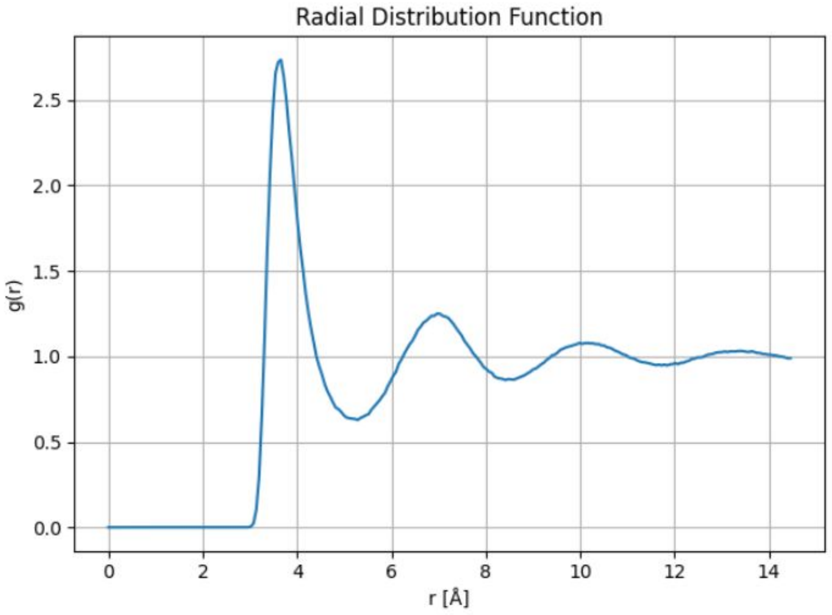


Hint: the radius can be written as
L_bin * integer and contiguous radii are
L_bin * integer & L_bin * (integer+1)

Normalization: Solution

```
g_normalization = np.zeros(N_bins)
for i in range(N_bins):
    g_normalization[i] = (4/3)*np.pi*((L_bin*(i+1))**3-(L_bin*i)**3)*N_atoms*N_frames*(N_atoms/L_box**3
```





- MD Visualization
- Analyse MD data
    - Property 1: MSD & Diffusion
    - Property 2
    - Property 3
- Produce MD data

- MD Visualization ✓
- Analyse MD data
  - Property 1: MSD & Diffusion ✓
  - Property 2: Radial distribution function ✓
  - Property 3
- Produce MD data

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

- Average over time and over particles!

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

- Average over time and over particles!

- For a given τ:

t=0

v_1(τ) · v_1(0)
v_2(τ) · v_2(0)
v_3(τ) · v_3(0)
v_4(τ) · v_4(0)

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

- Average over time and over particles!

- For a given τ:

$$\begin{array}{l} \qquad t=0 \\ \left.\begin{array}{l} v_1(\tau) \cdot v_1(0) \\ v_2(\tau) \cdot v_2(0) \\ v_3(\tau) \cdot v_3(0) \\ v_4(\tau) \cdot v_4(0) \end{array}\right\} \begin{array}{c}\text{Particle} \\ \text{av. } t=0\end{array} \end{array}$$

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

- Average over time and over particles!

- For a given τ:

t=0

$$\left.\begin{array}{l} v\_1(\tau) \cdot v\_1(0) \\ v\_2(\tau) \cdot v\_2(0) \\ v\_3(\tau) \cdot v\_3(0) \\ v\_4(\tau) \cdot v\_4(0) \end{array}\right\} \text{Particle av. t=0}$$

t=1

$$\left.\begin{array}{l} v\_1(\tau+1) \cdot v\_1(1) \\ v\_2(\tau+1) \cdot v\_2(1) \\ v\_3(\tau+1) \cdot v\_3(1) \\ v\_4(\tau+1) \cdot v\_4(1) \end{array}\right\} \text{Particle av. t=1}$$

t=2

$$\left.\begin{array}{l} v\_1(\tau+2) \cdot v\_1(2) \\ v\_2(\tau+2) \cdot v\_2(2) \\ v\_3(\tau+2) \cdot v\_3(2) \\ v\_4(\tau+2) \cdot v\_4(2) \end{array}\right\} \text{Particle av. t=2}$$

Velocity autocorrelation function

- Definition

$$VACF(\tau) = \langle v(\tau) \cdot v(0) \rangle$$

- Average over time and over particles!

- For a given τ:

Particle & time av.

t=0

$$\left.\begin{array}{l} v_1(\tau) \cdot v_1(0) \\ v_2(\tau) \cdot v_2(0) \\ v_3(\tau) \cdot v_3(0) \\ v_4(\tau) \cdot v_4(0) \end{array}\right\} \text{Particle av. } t=0$$

t=1

$$\left.\begin{array}{l} v_1(\tau+1) \cdot v_1(1) \\ v_2(\tau+1) \cdot v_2(1) \\ v_3(\tau+1) \cdot v_3(1) \\ v_4(\tau+1) \cdot v_4(1) \end{array}\right\} \text{Particle av. } t=1$$

t=2

$$\left.\begin{array}{l} v_1(\tau+2) \cdot v_1(2) \\ v_2(\tau+2) \cdot v_2(2) \\ v_3(\tau+2) \cdot v_3(2) \\ v_4(\tau+2) \cdot v_4(2) \end{array}\right\} \text{Particle av. } t=2$$

Velocity autocorrelation function

```python
def compute_vacf(vels, max_lag=N_frames):
    '''
    Compute the velocity autocorrelation function
    '''
    vacf = np.zeros(max_lag)
    for lag in range(max_lag):
        dot_sum = 0.0
        count = 0
        for t in range(N_frames - lag):
            v0 = vels[t]
            vlag = vels[t + lag]
            dot_sum +=                # Complete the function!
            count += N_atoms
        vacf[lag] = dot_sum / count
    return vacf
```

Particle & time av.

t=0

v_1(τ) · v_1(0)
v_2(τ) · v_2(0)
v_3(τ) · v_3(0)
v_4(τ) · v_4(0)

Particle av. t=0

t=1

v_1(τ+1) · v_1(1)
v_2(τ+1) · v_2(1)
v_3(τ+1) · v_3(1)
v_4(τ+1) · v_4(1)

Particle av. t=1

t=2

v_1(τ+2) · v_1(2)
v_2(τ+2) · v_2(2)
v_3(τ+2) · v_3(2)
v_4(τ+2) · v_4(2)

Particle av. t=2

Velocity autocorrelation function

```
def compute_vacf(vels, max_lag=N_frames):
    '''
    Compute the velocity autocorrelation function
    '''
    vacf = np.zeros(max_lag)
    for lag in range(max_lag):
        dot_sum = 0.0
        count = 0
        for t in range(N_frames - lag):
            v0 = vels[t]
            vlag = vels[t + lag]
            dot_sum +=                    # Complete the function!
            count += N_atoms
        vacf[lag] = dot_sum / count
    return vacf
```

Hint: The missing line includes only one multiplication and two nested np.sum()'s

Particle & time av.

t=0

v_1(τ) · v_1(0)
v_2(τ) · v_2(0)
v_3(τ) · v_3(0)
v_4(τ) · v_4(0)

Particle av. t=0

t=1

v_1(τ+1) · v_1(1)
v_2(τ+1) · v_2(1)
v_3(τ+1) · v_3(1)
v_4(τ+1) · v_4(1)

Particle av. t=1

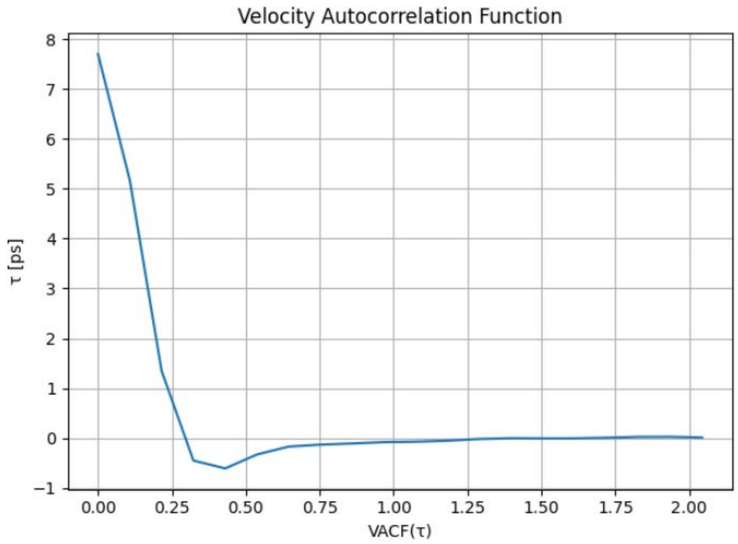
t=2

v_1(τ+2) · v_1(2)
v_2(τ+2) · v_2(2)
v_3(τ+2) · v_3(2)
v_4(τ+2) · v_4(2)

Particle av. t=2

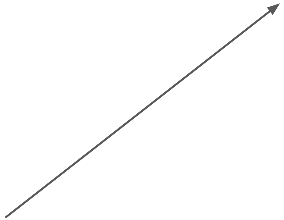
Velocity autocorrelation function: Solution

```python
def compute_vacf(vels, max_lag=N_frames):
    '''
    Compute the velocity autocorrelation function
    '''
    vacf = np.zeros(max_lag)
    for lag in range(max_lag):
        dot_sum = 0.0
        count = 0
        for t in range(N_frames - lag):
            v0 = vels[t]
            vlag = vels[t + lag]
            dot_sum += np.sum(np.sum(v0 * v1, axis=1)) # Complete the function!
            count += N_atoms
        vacf[lag] = dot_sum / count
    return vacf
```



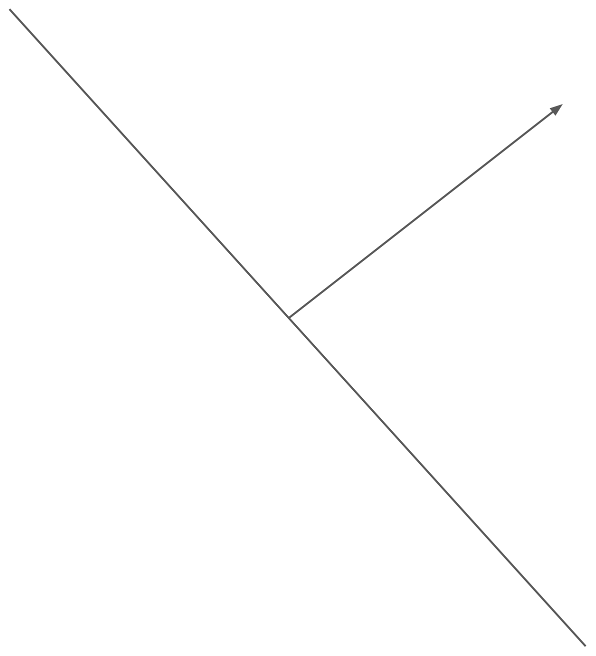
Velocity autocorrelation function

- When the velocity autocorrelation function is negative?

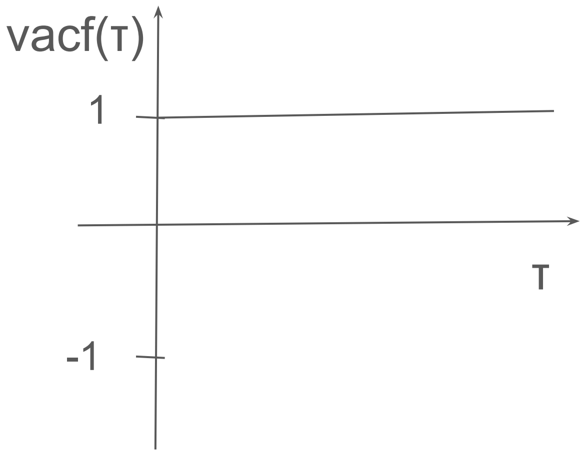


Velocity autocorrelation function

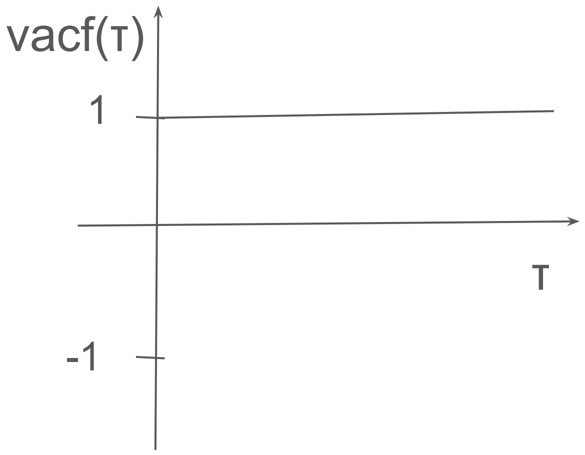
- When the velocity autocorrelation function is negative?

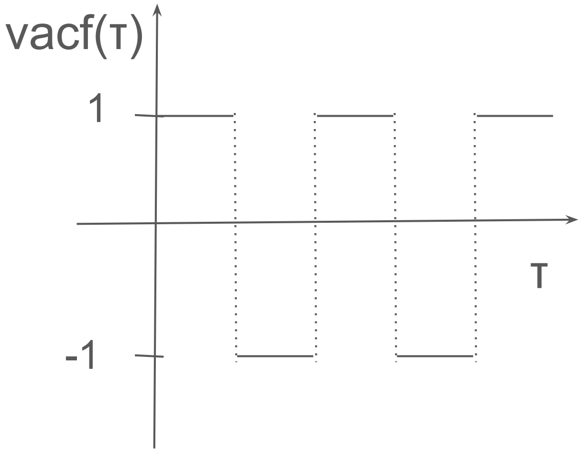


Velocity autocorrelation function



Velocity autocorrelation function





Area below VAFC

Estimate the integral of the VACF

```
np.trapezoid(vacf, time)/3
```

Area below VAFC

Estimate the integral of the VACF

```
np.trapezoid(vacf, time)/3
```

This is an estimation of a value that you obtained before, which one?

- MD Visualization

- Analyse MD data
    - Property 1: MSD & Diffusion
    - Property 2: Radial distribution function
    - Property 3

- Produce MD data

- MD Visualization

- Analyse MD data
    - Property 1: MSD & Diffusion
    - Property 2: Radial distribution function
    - Property 3: Velocity Autocorrelation function & Diffusion

- Produce MD data

Results comparison

Results comparison

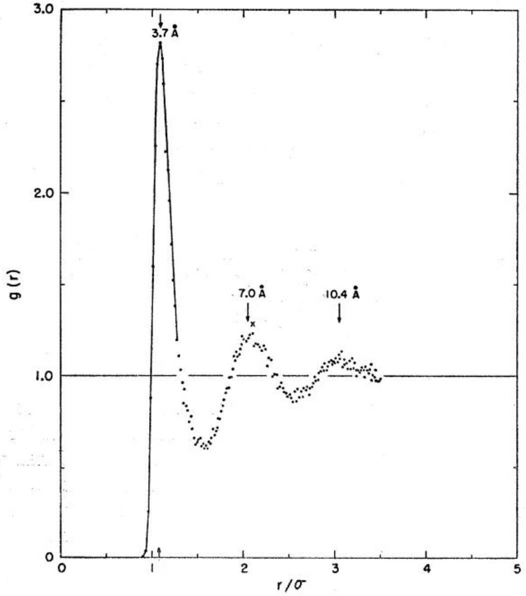


FIG. 2. Pair-correlation function obtained in this calculation at 94.4°K and 1.374 gcm^{-3}. The Fourier transform of this function has peaks at $\kappa\sigma = 6.8$, 12.5, 18.5, 24.8.

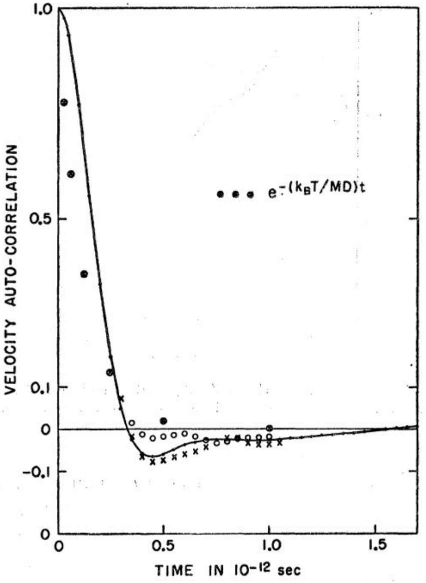


FIG. 4. The velocity autocorrelation function. The Langevin-type exponential function is also shown. The continuous curve, the circles, and the crosses correspond to the curves shown in Fig. 3.

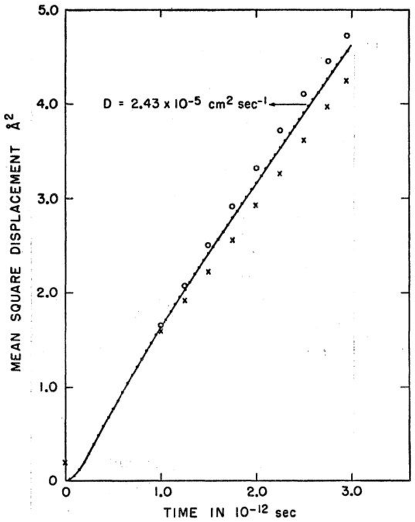


FIG. 3. Mean-square displacement of particles. The continuous curve is the mean of a set of 64 curves; the two members of the set which have *maximum* departures from the mean are shown as circles and as crosses. The asymptotic form of the continuous curve is $6Dt+C$, with D as shown on the figure and $C = 0.2$ Å^2.

Results comparison

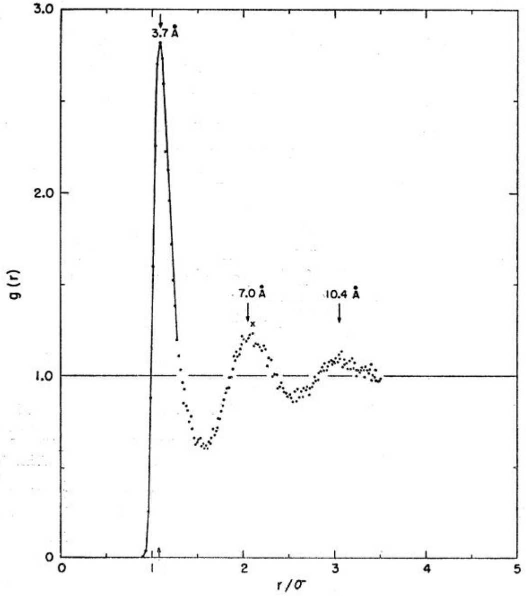


FIG. 2. Pair-correlation function obtained in this calculation at 94.4°K and 1.374 gcm^{-3}. The Fourier transform of this function has peaks at $\kappa\sigma = 6.8$, 12.5, 18.5, 24.8.

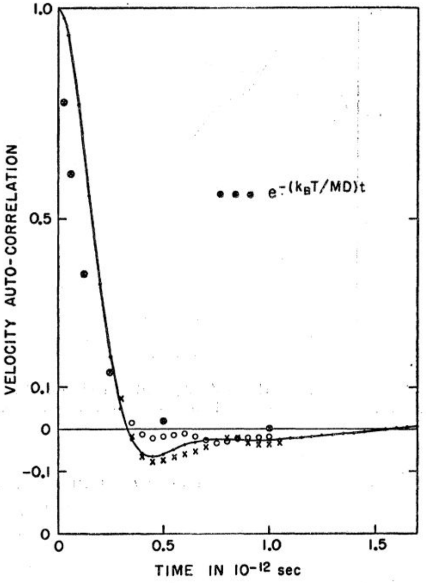


FIG. 4. The velocity autocorrelation function. The Langevin-type exponential function is also shown. The continuous curve, the circles, and the crosses correspond to the curves shown in Fig. 3.

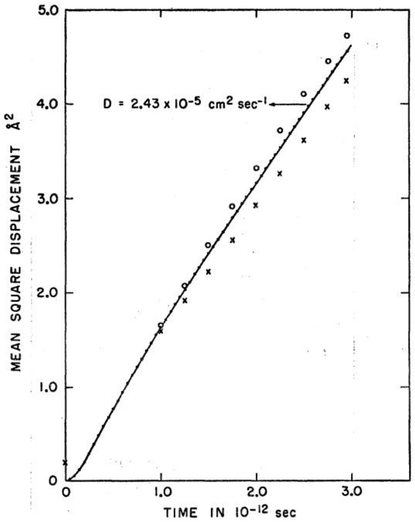


FIG. 3. Mean-square displacement of particles. The continuous curve is the mean of a set of 64 curves; the two members of the set which have *maximum* departures from the mean are shown as circles and as crosses. The asymptotic form of the continuous curve is $6Dt+C$, with D as shown on the figure and $C=0.2$ Å^2.

Question: The temperature of our system is higher or lower than the system in the paper? How do you know it?

About the paper

- MD Visualization

- Analyse MD data
    - Property 1: MSD & Diffusion
    - Property 2: Radial distribution function
    - Property 3: Velocity Autocorrelation function & Diffusion

- Produce MD data

- MD Visualization
- Analyse MD data
  - Property 1: MSD & Diffusion
  - Property 2: Radial distribution function
  - Property 3: Velocity Autocorrelation function & Diffusion
- Produce MD data

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.
- Explore the files NVE_main.py and NVE_lj.py to see the parameters.

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.
- Explore the files NVE_main.py and NVE_lj.py to see the parameters.
- Run the code with python NVE_main.py and explore the terminal messages

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.
- Explore the files NVE_main.py and NVE_lj.py to see the parameters.
- Run the code with python NVE_main.py and explore the terminal messages
- Challenges:
    - Reproduce the results of Rahman’s paper
    - Simulate a system in solid state
    - Simulate a system in the gaseous state.

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.
- Explore the files NVE_main.py and NVE_lj.py to see the parameters.
- Run the code with python NVE_main.py and explore the terminal messages
- Challenges:
    - Reproduce the results of Rahman's paper
    - Simulate a system in solid state
    - Simulate a system in the gaseous state.
- Corroborate (or refute) your predictions made during the Jupyter notebook.

Produce Data

- Go to the folder MDpython. This is the code with the parameters used to produce the data that we have analysed.
- Explore the files NVE_main.py and NVE_lj.py to see the parameters.
- Run the code with python NVE_main.py and explore the terminal messages
- Challenges:
  - Reproduce the results of Rahman’s paper
  - Simulate a system in solid state
  - Simulate a system in the gaseous state.
- Corroborate (or refute) your predictions made during the Jupyter notebook.
- Discuss your plots with your nearest neighbors.