# polypy Documentation

***Release 0.8.1***

**Adam R. Symington**

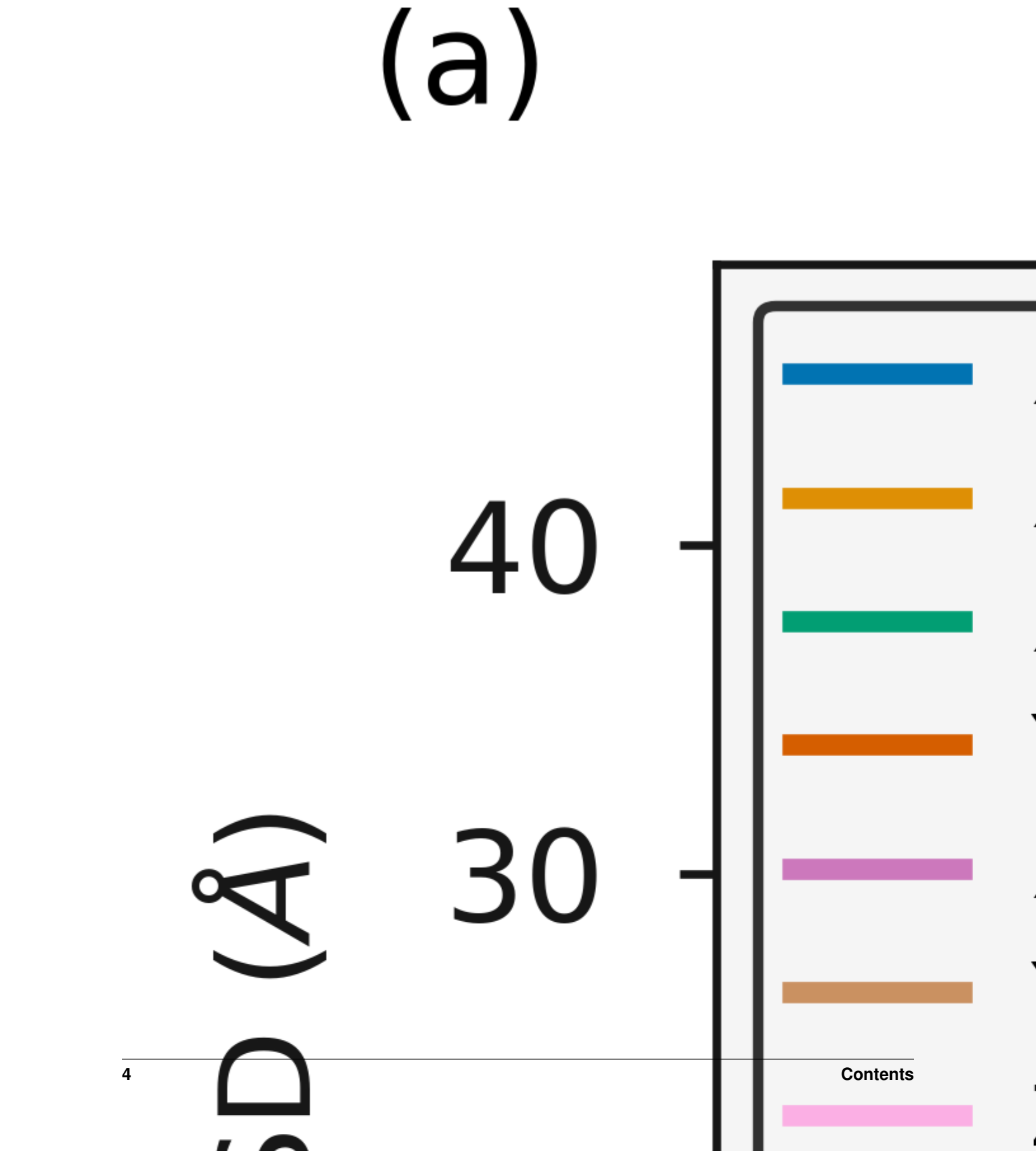
**Mar 04, 2021**

# Contents

This is the documentation for the open-source Python project - *polypy*. A library designed to facilitate the analysis of DL_POLY and DL_MONTE calculations. *polypy* is built on existing Python packages that those in the solid state physics/chemistry community should already be familiar with. It is hoped that this tool will bring some benfits to the solid state community and facilitate data analysis and the generation of publication ready plots (powered by Matplotlib.)

The main features include:

1. **Method to analyse the number denisty of a given species in one and two dimensions.**
    - Generate a plot of the total number of species in bins perpendicular to a specified direction.
    - Generate a plot of the total number of species in cuboids parallel to a specified direction.
2. **Method calculate the charge density from the number density.**
    - Convert number densities of all species in bins perpendicular to a specified direction into the charge density.
3. **Calculate the electric field and electrostatic potential from the charge density.**
    - Solves the Poisson Boltzmann equation to convert the charge density into the electric field and the electrostatic potential.
4. **Calculate the diffusion coefficient for a given species from a mean squared displacement.**
    - Carries out a mean squared displacement on an MD trajectory.
    - Calculates the diffusion coefficient.
    - Uses the density analysis and the diffusion coefficient to calculate the ionic conductivity.


(a)
40
30
(Å)

The code has been developed to analyse DL_POLY and DL_MONTE calculations however other codes can be incorporated if there is user demand. *polypy* was developed during a PhD project and as such the functionality focuses on the research questions encountered during that project, which we should clarify are wide ranging. Code contributions aimed at expanding the code to new of problems are encouraged. The code has been developed to analyse DL_POLY and DL_MONTE calculations however other codes can be incorporated if there is user demand. Other formats, such as pdb or xyz can be converted to *DL_POLY* format with codes such as <atomsk. and then analysed with *polypy*. Users are welcome to increase the file coverage by adding a reading function for a different format. This can be accomplished by adding to the *read* module which has a class for each unique file type that converts it to a *polypy.read.trajectory* object.

*polypy* is free to use.

A full list of examples can be found in the examples folder of the git repository, these include both the Python scripts and jupyter notebook tutorials which combine the full theory with code examples. It should be noted however that DL_POLY HISTORY files and DL_MONTE ARCHIVE files are sizable (1-5GB) and as such only short example trajectories are included in this repository. Notebooks are provided here to illustrate the theory but are not practicle.

CHAPTER 1

# Installation

*polypy* is a Python 3 package and requires a typical scientific Python stack. Use of the tutorials requires Anaconda/Jupyter to be installed.

To build from source:

```
pip install -r requirements.txt

python setup.py build

python setup.py install
```

Or alternatively install with pip

```
pip install polypy
```

Using conda,

```
conda skeleton pypi polypy

conda build polypy

conda install --use-local polypy
```

CHAPTER 2

# Tests

Tests can be run by typing:

```
python setup.py test
```

in the root directory.

CHAPTER 3

# Documentation

To build the documentation from scratch

```
cd docs

make html
```

CHAPTER 4

# License

*polypy* is made available under the MIT License.

CHAPTER 5

# Detailed requirements

*polypy* is compatible with Python 3.5+ and relies on a number of open source Python packages, specifically:

- Numpy
- Scipy
- Matplotlib

CHAPTER 6

# Contributing

## 6.1 Contact

If you have questions regarding any aspect of the software then please get in touch with the developer Adam Symington via email - ars44@bath.ac.uk. Alternatively you can create an issue on the Issue Tracker.

## 6.2 Bugs

There may be bugs. If you think you've caught one, please report it on the <Issue Tracker. This is also the place to propose new ideas for features or ask questions about the design of *polypy*. Poor documentation is considered a bug so feel free to request improvements.

## 6.3 Code contributions

We welcome help in improving and extending the package. This is managed through Github pull requests; for external contributions we prefer the "fork and pull" workflow while core developers use branches in the main repository:

1. First open an Issue to discuss the proposed contribution. This discussion might include how the changes fit surfinpy's scope and a general technical approach.
2. Make your own project fork and implement the changes there. Please keep your code style compliant with PEP8.
3. Open a pull request to merge the changes into the main project. A more detailed discussion can take place there before the changes are accepted.

For further information please contact Adam Symington, ars44@bath.ac.uk

## 6.4 Future

Listed below are a series of useful additions that we would like to make to the codebase. Users are encouraged to fork the repository and work on any of these problems. Indeed, if functionality is not listed below you are more than welcome to add it.

- RDF
- Diagonal slices
- Regional MSDs in a cube

## 6.5 Acknowledgements

This package was written during a PhD project that was funded by AWE and EPSRC (EP/R010366/1). The *polypy* software package was developed to analyse data generated using the Balena HPC facility at the University of Bath and the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk) via our membership of the UK's HEC Ma-terials Chemistry Consortium funded by EPSRC (EP/L000202).The author would like to thank Andrew R. McCluskey, Benjamin Morgan, Marco Molinari, James Grant and Stephen C. Parker for their help and guidance during this PhD project.

## 6.6 API

### 6.6.1 Installation

`polypy` can be installed from the PyPI package manager with `pip`

```
pip install polypy
```

Alternatively, if you would like to download and install the latest development build, it can be found Github along with specific installation instructions.

### 6.6.2 Theory

*polypy* is a Python module to analyse DLPOLY and DLMONTE trajectory files. Before using this code you will need to generate the relevant data. *polypy* is aimed at the solid state community and there are a wide range of applications.

#### Charge Density

Using the density module you can calculate the number density of atoms

$$\rho_q(z) = \sum_i q_i \rho_i(z)$$

where $\rho_i$ is the density of atom i and $q_i$ is its charge.

### Electric Field and Electrostatic Potential

The charge density can be converted into the electric field and the electrostatic potential.

The electric field is calculated according to

$$E(z) = \frac{1}{-\epsilon_0} \int_{z_0}^{z} \rho_q(z')dz'$$

where $\epsilon_0$ is the permittivity of the vacuum and $\rho_q$ is the charge density. The electrostatic potential is calculated according to

$$\Delta_\psi(z) = \int_{z_0}^{z} E(z')dz'$$

### Mean Squared Displacement

Molecules in liquds, gases and solids do not stay in the same place and move constantly. Think about a drop of dye in a glass of water, as time passes the dye distributes throughout the water. This process is called diffusion and is common throughout nature and an incredibly relevant property for materials scientists who work on things like batteries.

Using the dye as an example, the motion of a dye molecule is not simple. As it moves it is jostled by collisions with other molecules, preventing it from moving in a straight path. If the path is examined in close detail, it will be seen to be a good approximation to a random walk. In mathmatics a random walk is a series of steps, each taken in a random direction. This was analysed by Albert Einstein in a study of Brownian motion and he showed that the mean square of the distance travelled by a particle following a random walk is proportional to the time elapsed.

$$\left\langle r_i^2 \right\rangle = 6D_t + C$$

where

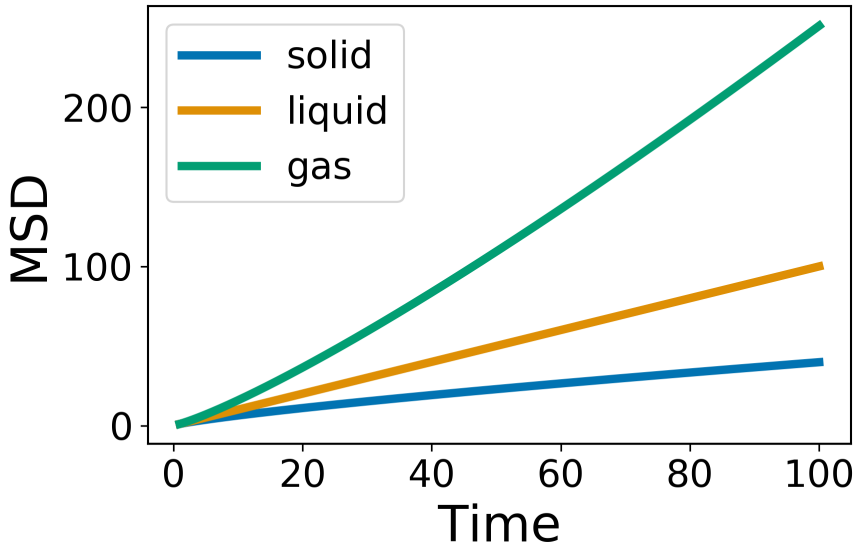
$$\left\langle r_i^2 \right\rangle = \frac{1}{3}\left\langle |r_i(t) - r_i(0)|^2 \right\rangle,$$

where $\left\langle r^2 \right\rangle$ is the mean squared distance, t is time, $D_t$ is the diffusion rate and C is a constant. If $\left\langle r_i^2 \right\rangle$ is plotted as a function of time, the gradient of the curve obtained is equal to 6 times the self-diffusion coefficient of particle i.

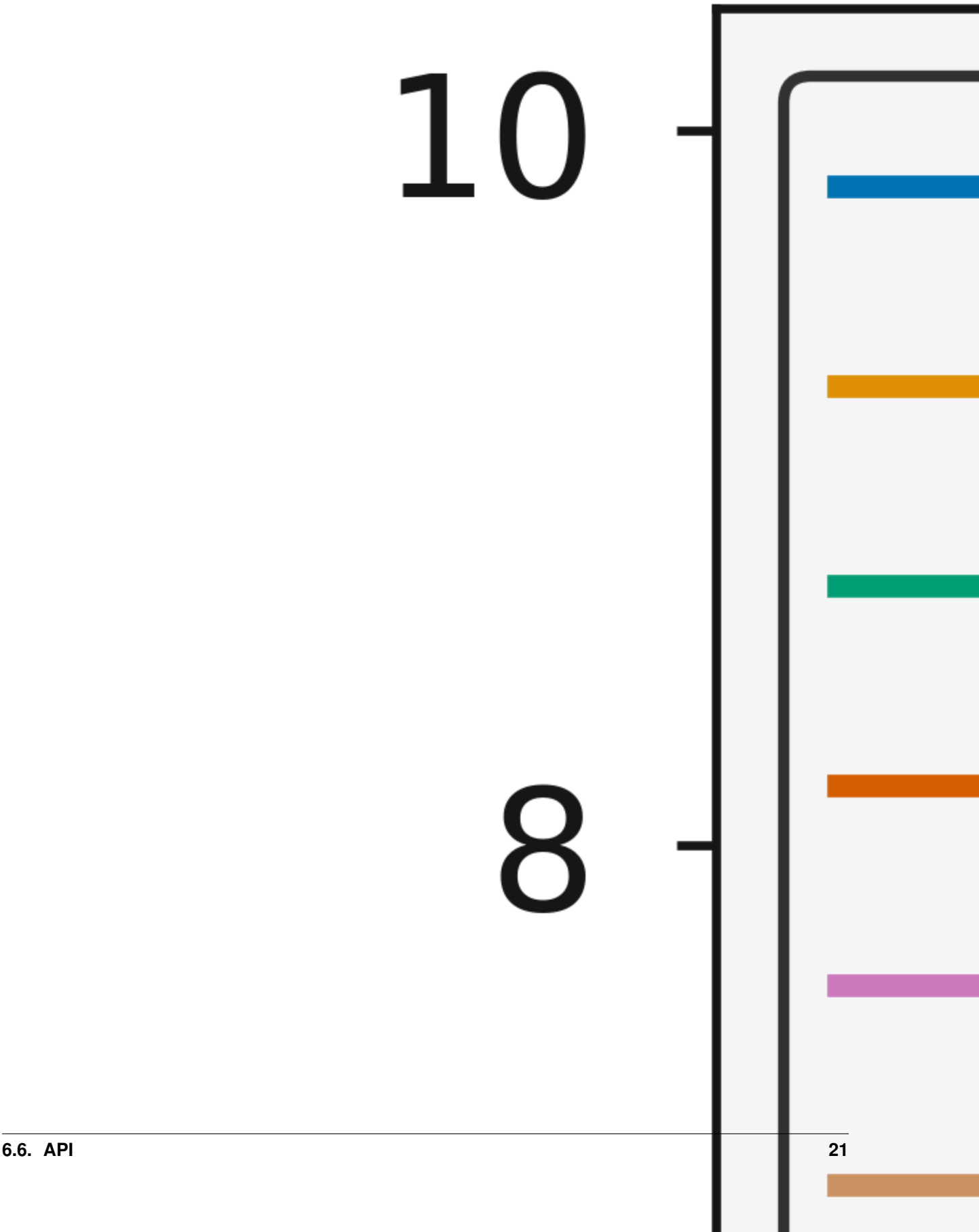
### What is the mean squared displacement

Going back to the example of the dye in water, lets assume for the sake of simplicity that we are in one dimension. Each step can either be forwards or backwards and we cannot predict which. From a given starting position, what distance is our dye molecule likely to travel after 1000 steps? This can be determined simply by adding together the steps, taking into account the fact that steps backwards subtract from the total, while steps forward add to the total. Since both forward and backward steps are equally probable, we come to the surprising conclusion that the probable distance travelled sums up to zero.

By adding the square of the distance we will always be adding positive numbers to our total which now increases linearly with time. Based upon equation 1 it should now be clear that a plot of $\left\langle r_i^2 \right\rangle$ vs time with produce a line, the gradient of which is equal to 6D. Giving us direct access to the diffusion coefficient of the system. The state of the matter effects the shape of the MSD plot, solids, where little to no diffusion is occuring, has a flat MSD profile. In a liquid however, the particles diffusion randomly and the gradient of the curve is proportional to the diffusion coefficient.




The following example is for fluorine diffusion in $CaF_2$.


10
8

#### Ionic Conductivity

Usefully, as we have the diffusion coefficient, the number of particles (charge carriers) and the ability to calculate the volume, we can convert this data into the ionic conductivity and then the resistance.

$$\sigma = \frac{DC_F e^2}{k_B T} Hr$$

where $\sigma$ is the ionic conductivity, D is the diffusion coefficient,:math:*C_F* is the concentration of charge carriers, which in this case if F ions, $e^2$ is the charge of the diffusing species, $k_B$ is the Boltzmann constant, T is the temperature and Hr is the Haven ratio.

The resistance can then be calculated according to

$$\Omega = \frac{1}{\sigma}$$

#### Arrhenius

It is possible to calculate the diffusion coefficients over a large temperature range and then use the Arrhenius equation to calculate the activation energy for diffusion. Common sense and chemical intuition suggest that the higher the temperature, the faster a given chemical reaction will proceed. Quantitatively this relationship between the rate a reaction proceeds and its temperature is determined by the Arrhenius Equation. At higher temperatures, the probability that two molecules will collide is higher. This higher collision rate results in a higher kinetic energy, which has an effect on the activation energy of the reaction. The activation energy is the amount of energy required to ensure that a reaction happens.

$$k = Ae^{(-Ea/RT)}$$

where k is the rate coefficient, A is a constant, Ea is the activation energy, R is the universal gas constant, and T is the temperature (in kelvin).

### 6.6.3 Using polypy

The are number of ways to get and use polypy

- Fork the code: please feel free to fork the code on Github and add functionality that interests you.
- Run it locally: polypy is available through the pip package manager.
- Get in touch: Adam R.Symington (ars44@bath.ac.uk) is always keen to chat to potential users.

### 6.6.4 Tutorials

These tutorials are replicated in jupyter notebook form and contained within examples. All of these examples can be found in examples/notebooks.

All tutorials use fluorite $CeO_2$ grain boundary as an example. Due to the large size of DL_POLY and DL_MONTE trajectory files, the tutorial notebooks contained within the git repository use a very short $CaF_2$ trajectory.

### Tutorial 1 - Reading data

The HISTORY, ARCHIVE and CONFIG classes expects two things, the filename of the history file and a list of atoms to read. They will return a `polypy.read.Trajectory` object, which stores the the atom labels (`Trajectory.atom_list`), datatype (`Trajectory.datatype`), cartesian coordinates (`Trajectory.cartesian_coordinates`), fractiona coordinates (`Trajectory.fractional_coordinates`), reciprocal lattice vectors (`Trajectory.reciprocal_lv`), lattice vectors (`Trajectory.lv`) cell lengths (`Trajectory.cell_lengths`), total atoms in the file (`Trajectory.atoms_in_history`), timesteps (`Trajectory.timesteps`), total atoms per timestep (`Trajectory.total_atoms`).

### HISTORY Files

```
from polypy import read as rd
```

```
history = rd.History("../example_data/HISTORY_CaF2", ["CA", "F"])
```

```
print(history.trajectory.fractional_trajectory)
```

```
[[0.5170937  0.51658126 0.51643485]
 [0.51658126 0.61669107 0.61654466]
 [0.61669107 0.51658126 0.61691069]
 ...
 [0.46866197 0.25395423 0.58485915]
 [0.37035211 0.58795775 0.45221831]
 [0.36552817 0.48637324 0.17484859]]
```

```
print(history.trajectory.timesteps)
```

```
500
```

```
print(history.trajectory.atoms_in_history)
```

```
750000
```

```
print(history.trajectory.total_atoms)
```

```
1500
```

It is often necessary to remove the equilibriation timesteps from the simulation. This can be accomlished with the remove_initial_timesteps method to remove timesteps at the start of the simulation and the remove_final_timesteps, to remove timesteps at the end of the simulation.

```
new_history = history.trajectory.remove_initial_timesteps(10)
print(new_history.timesteps)
```

```
490
```

```
new_history = new_history.remove_final_timesteps(10)
print(new_history.timesteps)
```

```
480
```

It is possible to return the trajectory for a single timestep within the history file or to return the trajectory for a single atom.

```
config_ca = history.trajectory.get_atom("CA")

print(config_ca.fractional_trajectory)
```

```
[[0.5170937  0.51658126 0.51643485]
 [0.51658126 0.61669107 0.61654466]
 [0.61669107 0.51658126 0.61691069]
 ...
 [0.31458099 0.41869718 0.41764085]
 [0.42742958 0.32461268 0.42507042]
 [0.42485915 0.42183099 0.31564789]]
```

```
config_1 = history.trajectory.get_config(1)

print(config_1.fractional_trajectory)
```

```
[[0.53227339 0.51016082 0.50950292]
 [0.52116228 0.62894737 0.61761696]
 [0.62240497 0.50526316 0.6056652 ]
 ...
 [0.39444444 0.44974415 0.45102339]
 [0.45599415 0.37865497 0.39890351]
 [0.36343202 0.49309211 0.3690424 ]]
```

### CONFIG Files

```
config = rd.Config("../example_data/CONFIG", ["CA", "F"])
```

```
print(config.trajectory.fractional_trajectory)
```

```
[[0.51666667 0.51666667 0.51666667]
 [0.51666667 0.61666667 0.61666667]
 [0.61666667 0.51666667 0.61666667]
 ...
 [0.36666667 0.46666667 0.46666667]
 [0.46666667 0.36666667 0.36666667]
 [0.36666667 0.46666667 0.36666667]]
```

### DLMONTE

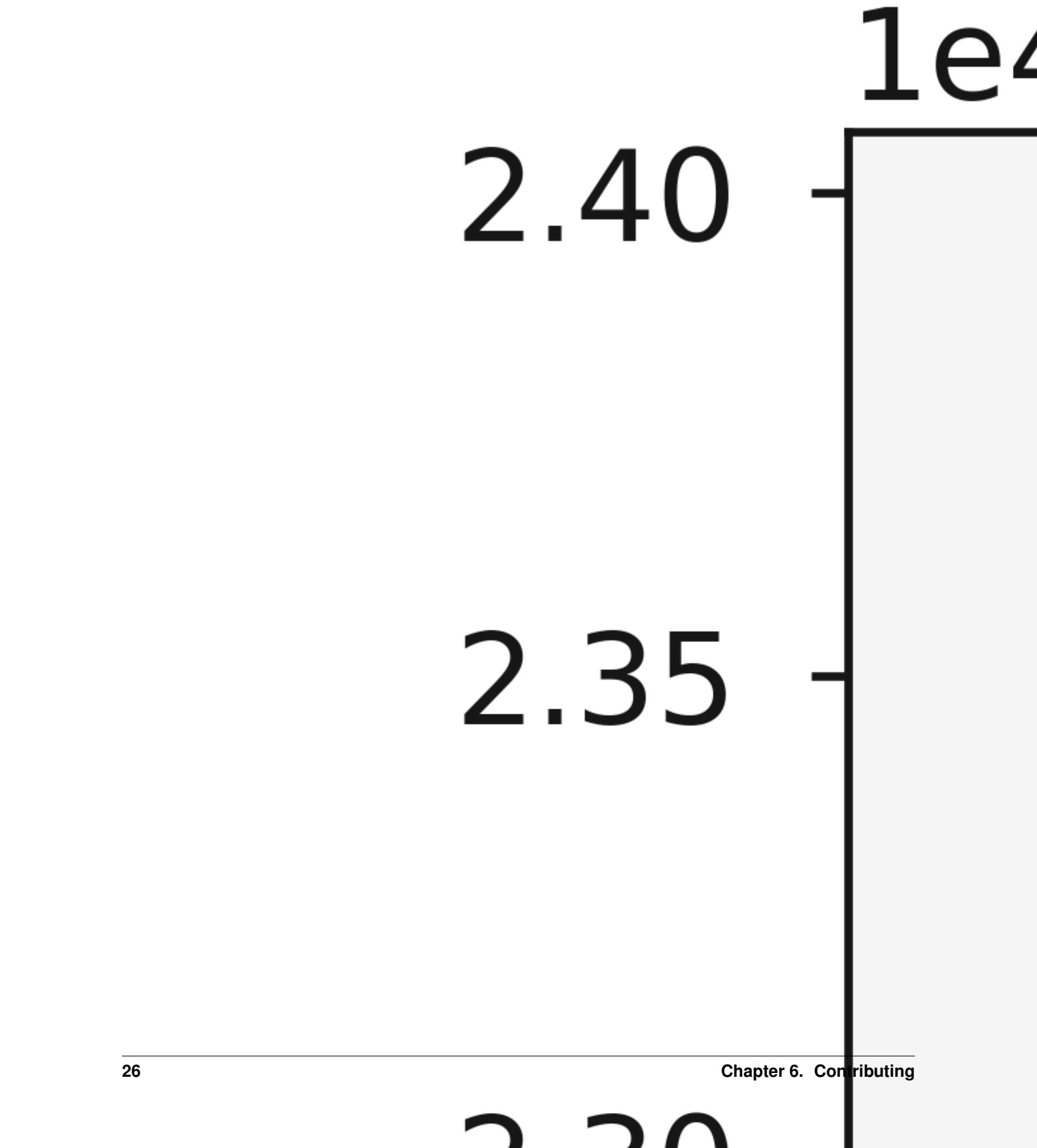
```
archive = rd.Archive("../example_data/ARCHIVE_Short", ["AL"])
```

```
print(archive.trajectory.timesteps)
```

```
1000
```

### Volume

```
from polypy import analysis
from polypy import plotting
from polypy import read as rd
import matplotlib.pyplot as plt


history = rd.History("../example_data/HISTORY_CaF2", ["CA"])

volume, step = analysis.system_volume(history.trajectory)

ax = plotting.volume_plot(step, volume)
plt.show()
```


1e4
2.40
2.35

### Atomic Density

### Density Analysis

Understanding the positions of atoms in a material is incredibly useful when studying things like atomic structure and defect segregation. Consider a system with an interface, it may be interesting to know how the distributions of the materials atoms change at that interface, e.g is there an increase or decrease in the amount of a certain species at the interface and does this inform you about any segregation behaviour?

This module of polypy allows the positions of atoms in a simulation to be evaluated in one and two dimensions, this can then be converted into a charge density and (in one dimension) the electric field and electrostatic potential.

```
from polypy.read import History
from polypy.read import Archive
from polypy.density import Density
from polypy import analysis
from polypy import utils as ut
from polypy import plotting

import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

In this tutorial, we will use polypy to analyse a molecular dynamics simulation of a grain boundary in fluorite cerium oxide and a Monte Carlo simulation of Al, Li and Li vacancy swaps in lithium lanthanum titanate.

### Example 1 - Cerium Oxide Grain Boundary

In this example we will use `polypy` to analyse a molecular dynamics simulation of a grain boundary in cerium oxide.

The first step is to read the data. We want the data for both species so need to provide a list of the species.

```
["CE", "O"]
```

Note. In all examples, an `xlim` has been specified to highlight the grain boundary. Feel free to remove the `ax.set_xlim(42, 82)` to see the whole plot.

```
history = History("../example_data/HISTORY_GB", ["CE", "O"])

print(np.amin(history.trajectory.cartesian_trajectory))
print(np.amax(history.trajectory.fractional_trajectory))
```

```
-63.929
0.9999993486383602
```

The next step is to create the density object for both species. In this example we create a seperate object for the cerium and oxygen atoms and we will be analysing the positions to a resolution of 0.1 angstroms.

```
ce_density = Density(history.trajectory, atom="CE", histogram_size=0.1)
o_density = Density(history.trajectory, atom="O", histogram_size=0.1)
```

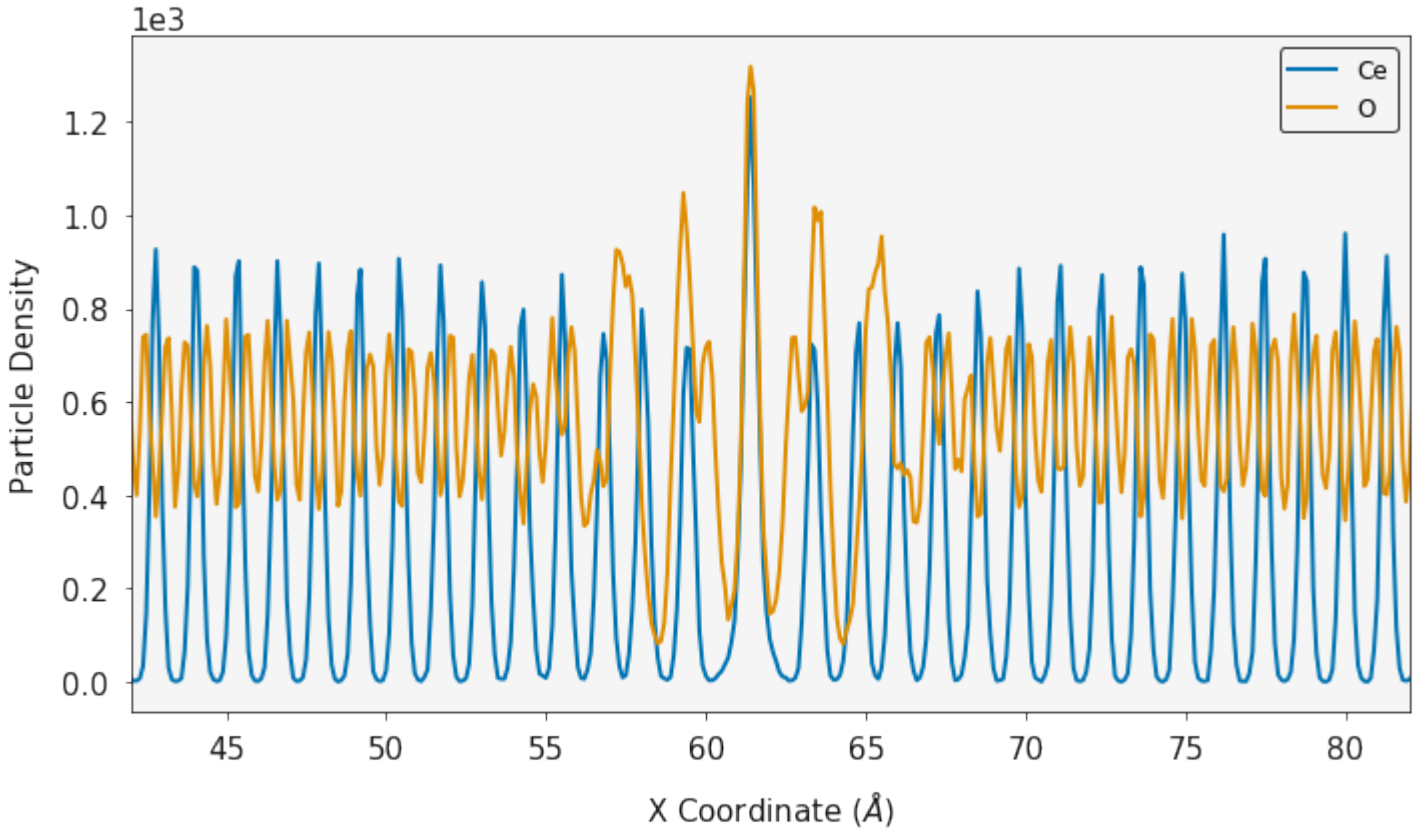
All subsequent analysis is performed on these two objects.

### One Dimension

The `one_dimensional_density` function will take a direction which corresponds to a dimension of the simulation cell. For example, 'x' corresponds to the first lattice vector. The code will calculate the total number of a species in 0.1 angstrom histograms along the first cell dimension.

The function will return the positions of the histograms and the total number of species. These can then be plotted with the `one_dimensional_density_plot` function which takes a list of histogram values, a list of particle densities and a list of labels.

```
cx, cy, c_volume = ce_density.one_dimensional_density(direction="z")
ox, oy, o_volume = o_density.one_dimensional_density(direction="z")

ax = plotting.one_dimensional_density_plot([cx, ox], [cy, oy], ["Ce", "O"])
ax.set_xlim(42, 82)
plt.show()
```



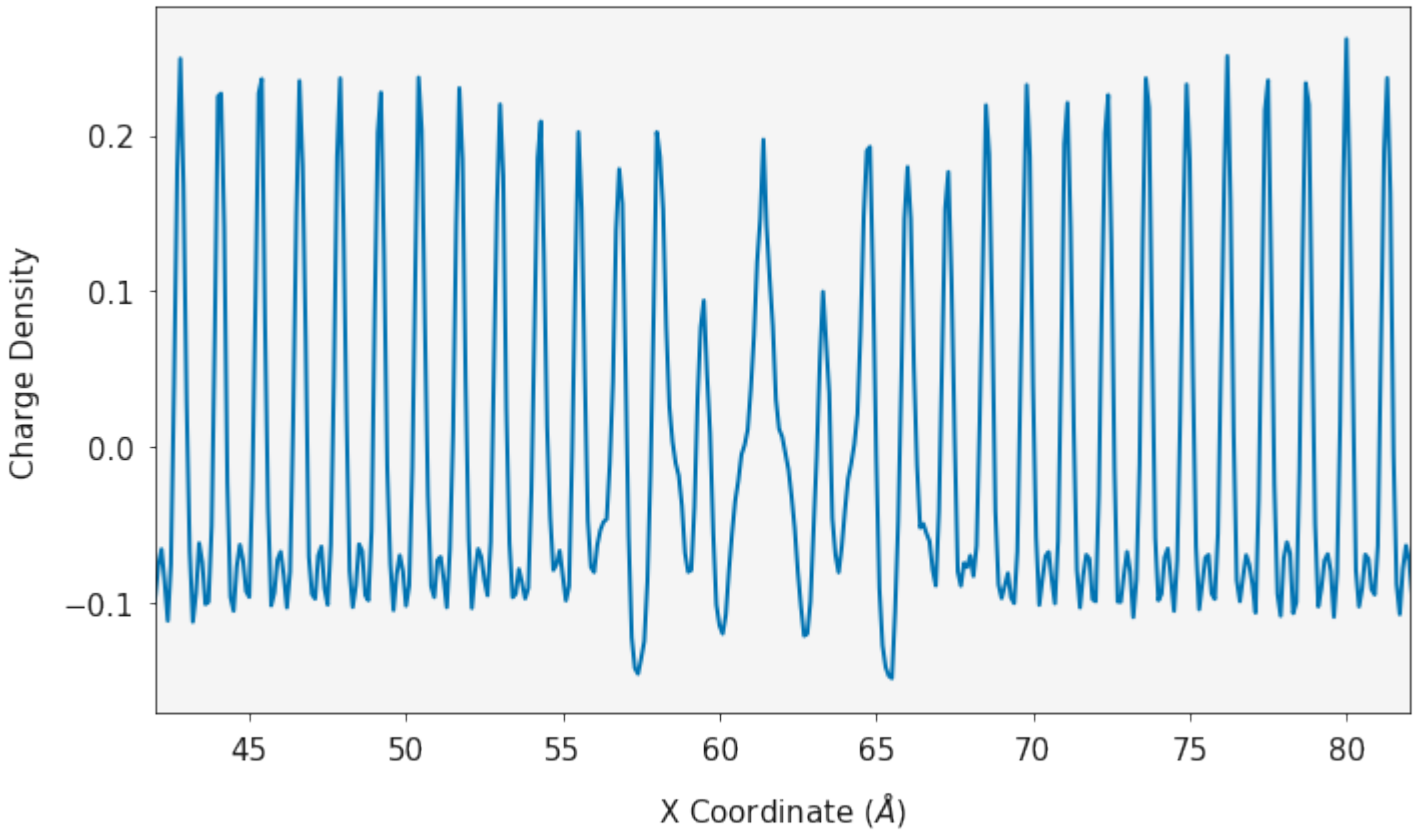

The particle densities can be combined with the atom charges to generate the one dimensional charge density according to

$$\rho_q(z) = \sum_i q_i \rho_i(z)$$

where $\rho_i$ is the particle density of atom i and $q_i$ is its charge.

The `OneDimensionalChargeDensity` class is used for the charge density, electric field and electrostatic potential. It requires a list of particle densities, list of charges, the histogram volume and the total number of timesteps.

```
charge = analysis.OneDimensionalChargeDensity(ox, [oy, cy], [-2.0, 4.0], c_volume,␣
↪history.trajectory.timesteps)

dx, charge_density = charge.calculate_charge_density()

ax = plotting.one_dimensional_charge_density_plot(dx, charge_density)
ax.set_xlim(42, 82)

plt.show()
```



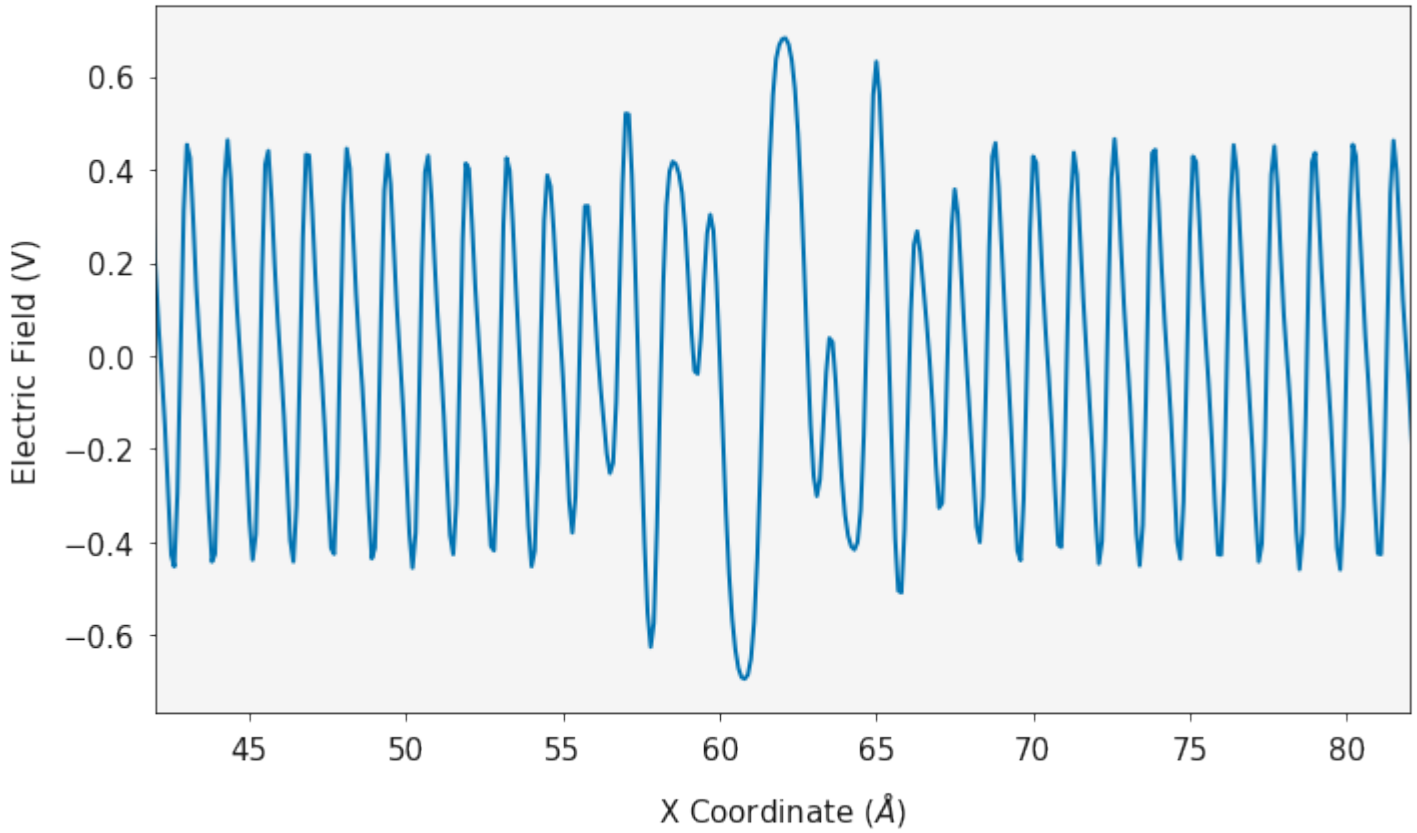

The charge density can be converted into the electric field and the electrostatic potential.

$$E(z) = \frac{1}{-\epsilon_0} \int_{z_0}^{z} \rho_q(z')dz'$$

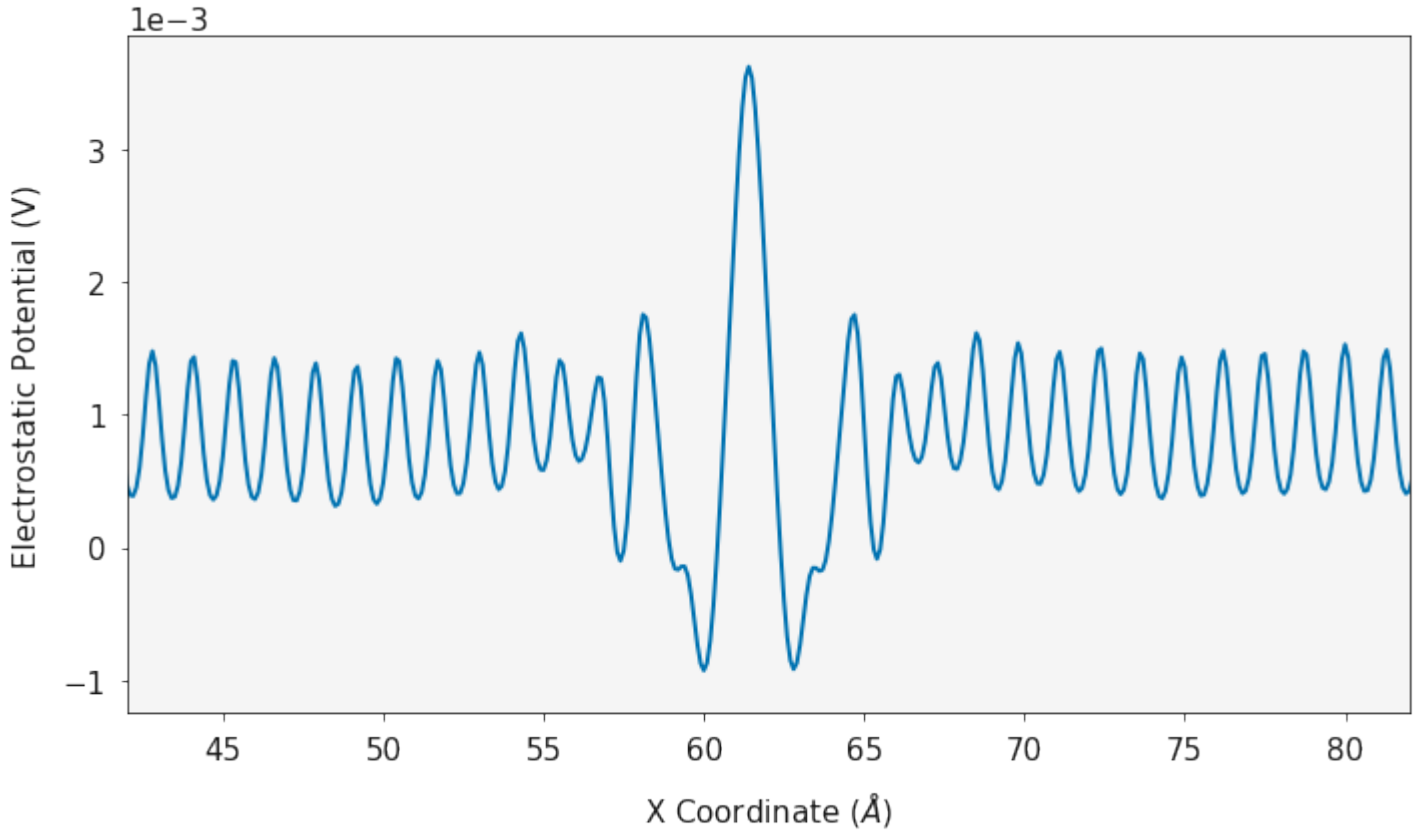
$$\Delta_\psi(z) = \int_{z_0}^{z} E(z')dz'$$

where $\rho_i$ is the charge density and $\epsilon_0$ is the permittivity of free space The `calculate_electric_field` and `calculate_electrostatic_potential` functions will return the electric field and the electrostatic potential.

```
dx, electric_field = charge.calculate_electric_field()

ax = plotting.electric_field_plot(dx, electric_field)
ax.set_xlim(42, 82)
plt.show()
```




```
dx, electrostatic_potential = charge.calculate_electrostatic_potential()

ax = plotting.electrostatic_potential_plot(dx, electrostatic_potential)
ax.set_xlim(42, 82)

plt.show()
```

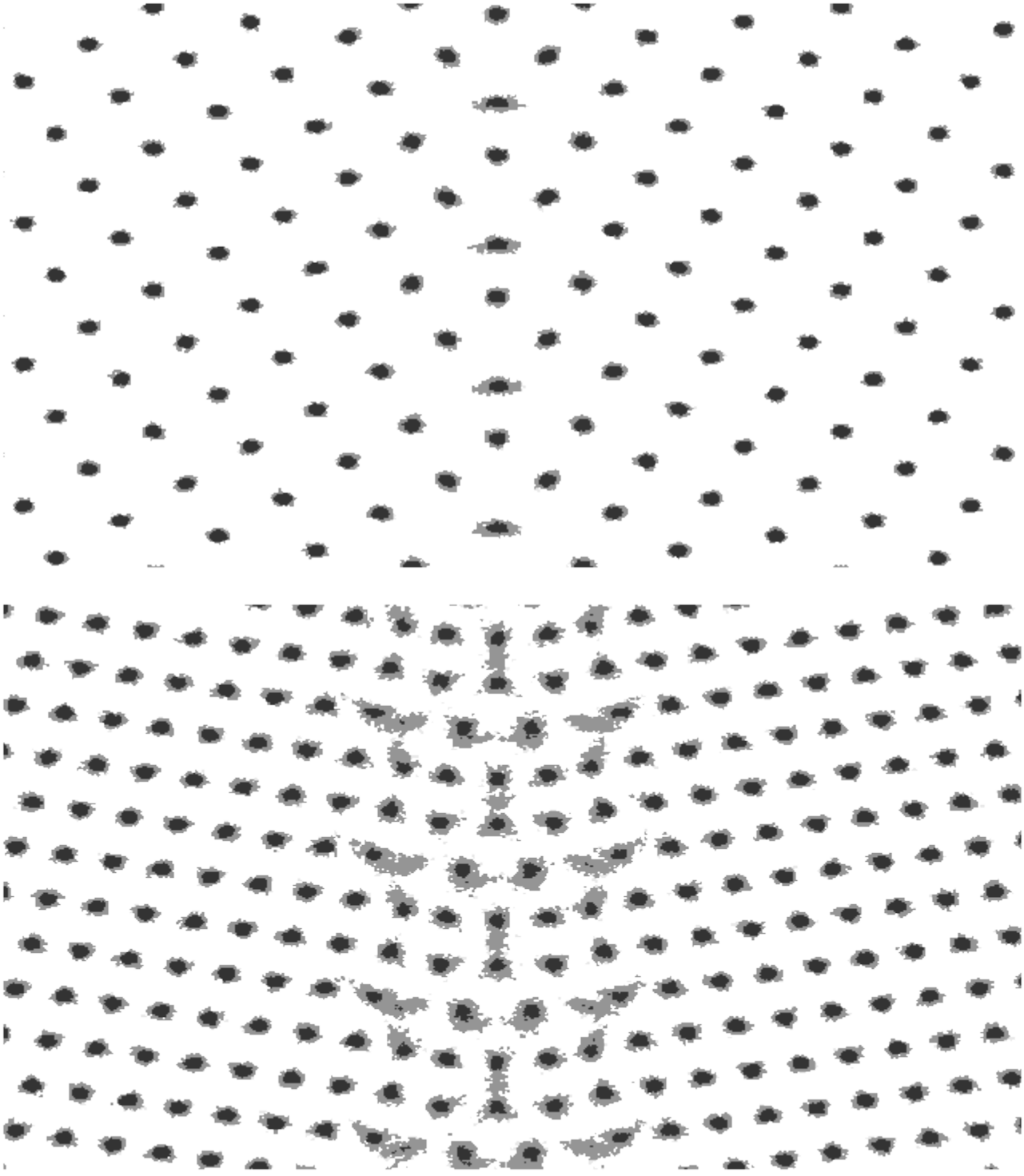



### Two Dimensions

The particle density can be evaluated in two dimensions. The `two_dimensional_density` function will calculate the total number of species in histograms. The coordinates in x and y of the box are returned and a grid of species counts are returned.

In this example, the colorbar has been turned off, we are using a grey palette and the data is being plotted on a log scale.

```
cx_2d, cy_2d, cz_2d, c_volume = ce_density.two_dimensional_density(direction="x")
ox_2d, oy_2d, oz_2d, o_volume = o_density.two_dimensional_density(direction="x")

fig, ax = plotting.two_dimensional_density_plot(cx_2d, cy_2d, cz_2d, colorbar=False,␣
↪palette="Greys", log=True)
ax.set_xlim(42, 82)
ax.axis('off')
plt.show()

fig, ax = plotting.two_dimensional_density_plot(ox_2d, oy_2d, oz_2d, colorbar=False,␣
↪palette="Greys", log=True)
ax.set_xlim(42, 82)
ax.axis('off')
plt.show()
```



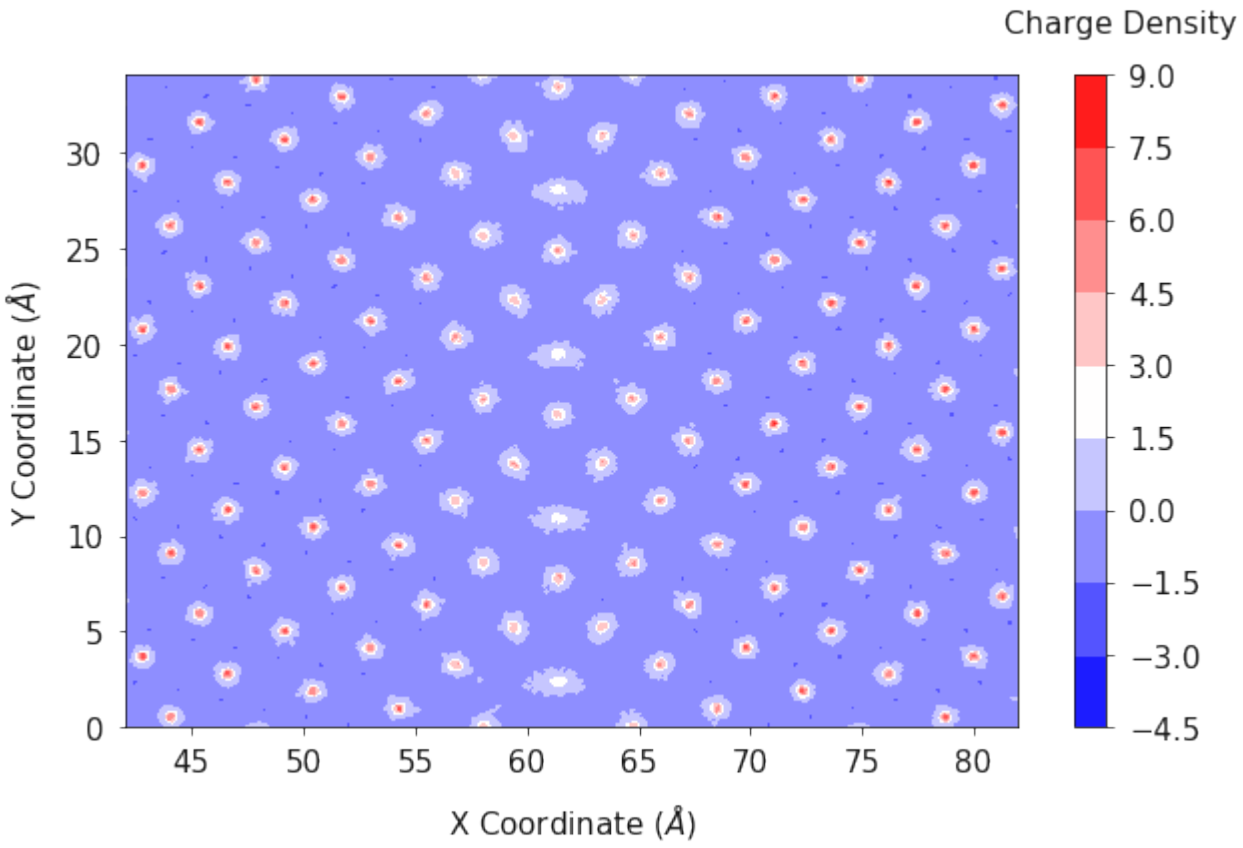
In the same fashion as the one dimensional case, the charge density can be evaluated in two dimensions using the `two_dimensional_charge_density` function. This function requires the two dimensional array of atom positions, the atom charges, the volume at each grid point and the total number of timesteps in the simulation.

```
charge_density = analysis.two_dimensional_charge_density([oz_2d, cz_2d], [-2.0, 4.0],␣
↪o_volume, history.trajectory.timesteps)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plotting.two_dimensional_charge_density_plot(ox_2d, oy_2d, charge_density,␣
↪palette='bwr')
ax.set_xlim(42, 82)
plt.show()
```




### One and Two Dimensions

The contour plots can give a good understanding of the average positions of the atoms (or the location of the lattice sites) however it does not give a good representation of how many species are actually there. The `combined_density_plot` function will evaluate the particle density in one and two dimensions and then overlay the two on to a single plot, allowing both the lattice sites, and total density to be viewed.

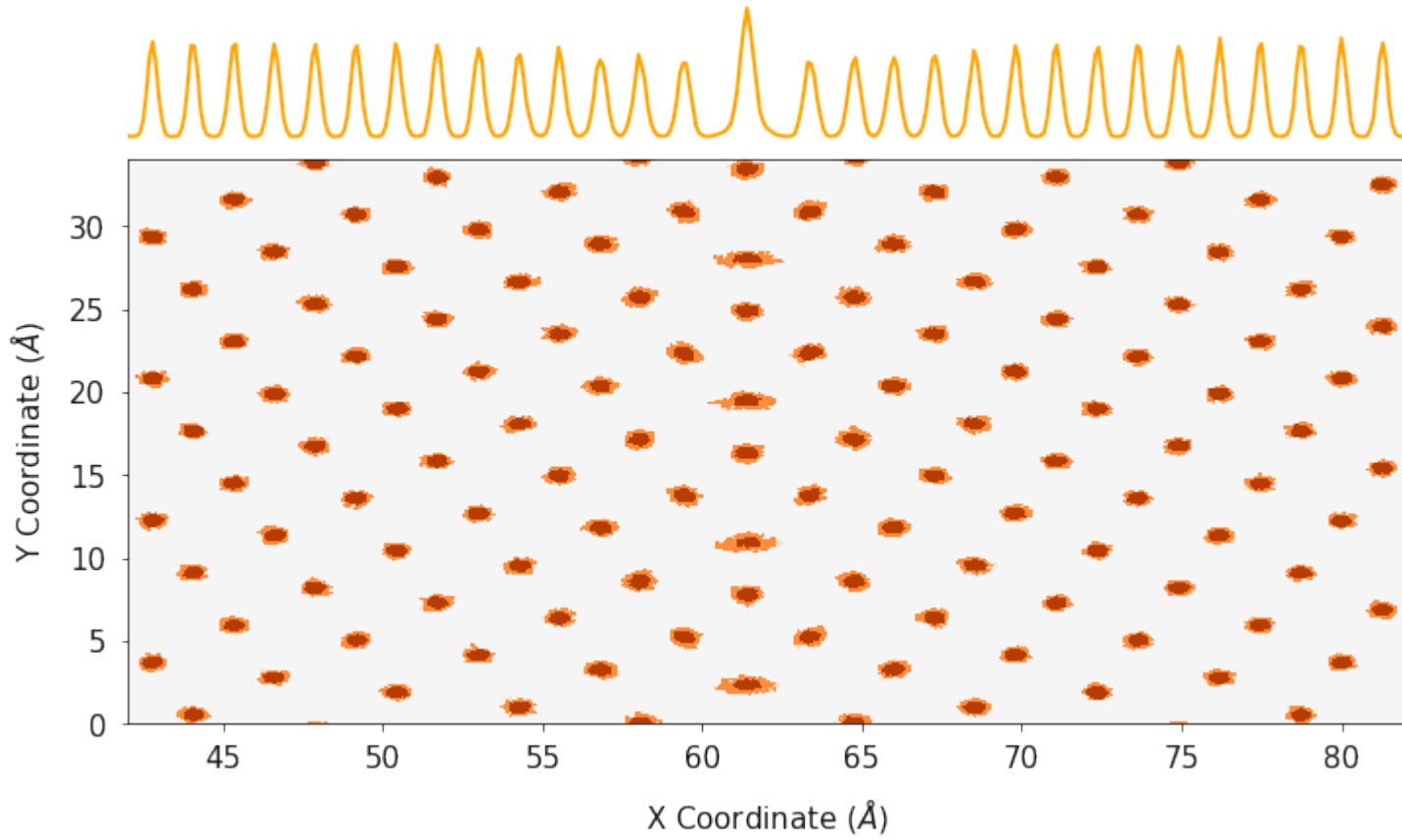
In this example we are using an orange palette and orange line color for the cerium atoms, a blue palette and blue line for the oxygen positions and the data is plotted on a log scale.

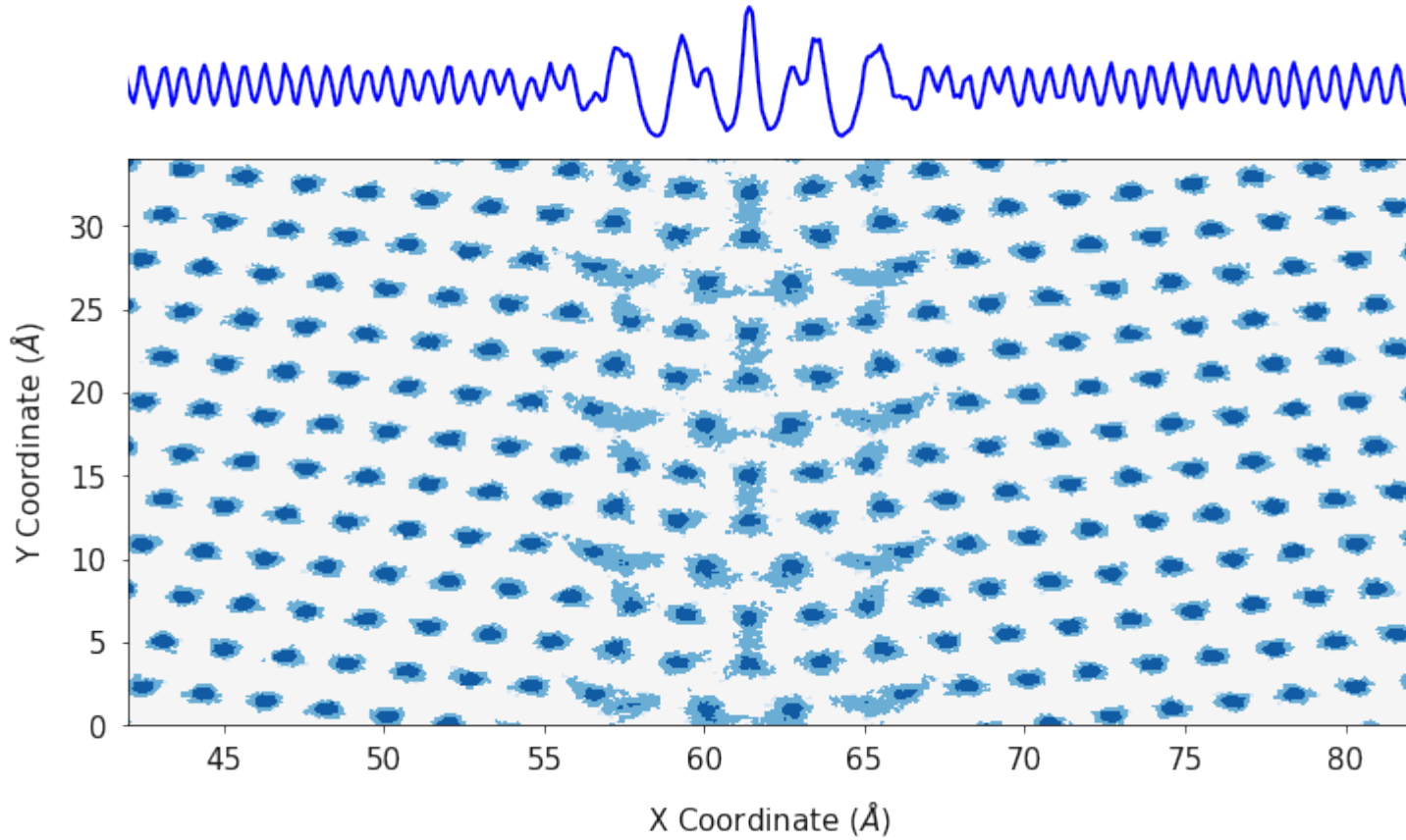
```
fig, ax = plotting.combined_density_plot(cx_2d, cy_2d, cz_2d, palette="Oranges",␣
↪linecolor="orange", log=True)
for axes in ax:
    axes.set_xlim(42, 82)
plt.show()

fig, ax = plotting.combined_density_plot(ox_2d, oy_2d, oz_2d, palette="Blues",␣
↪linecolor="blue", log=True)
for axes in ax:
```

(continues on next page)

(continued from previous page)

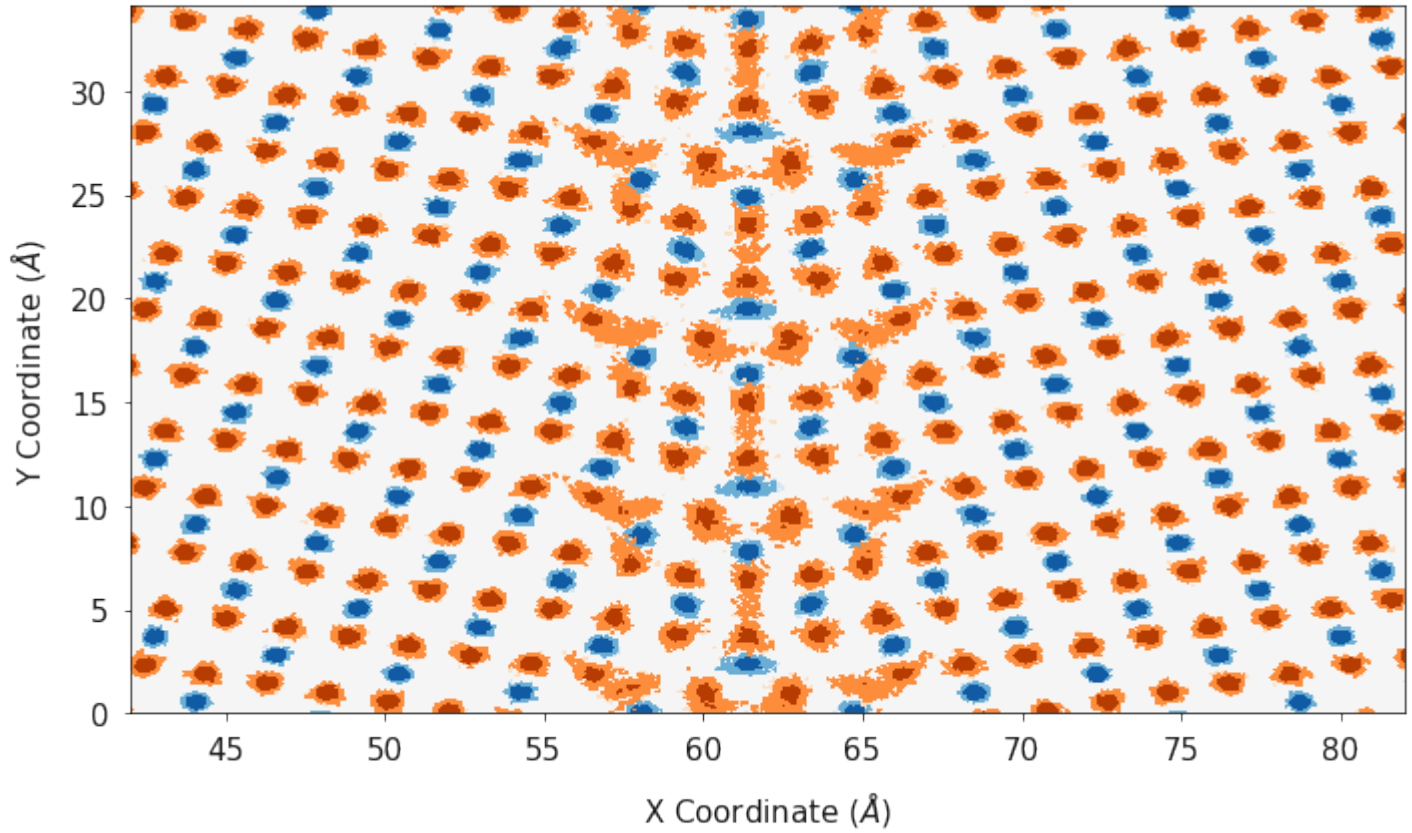
```
    axes.set_xlim(42, 82)
plt.show()
```





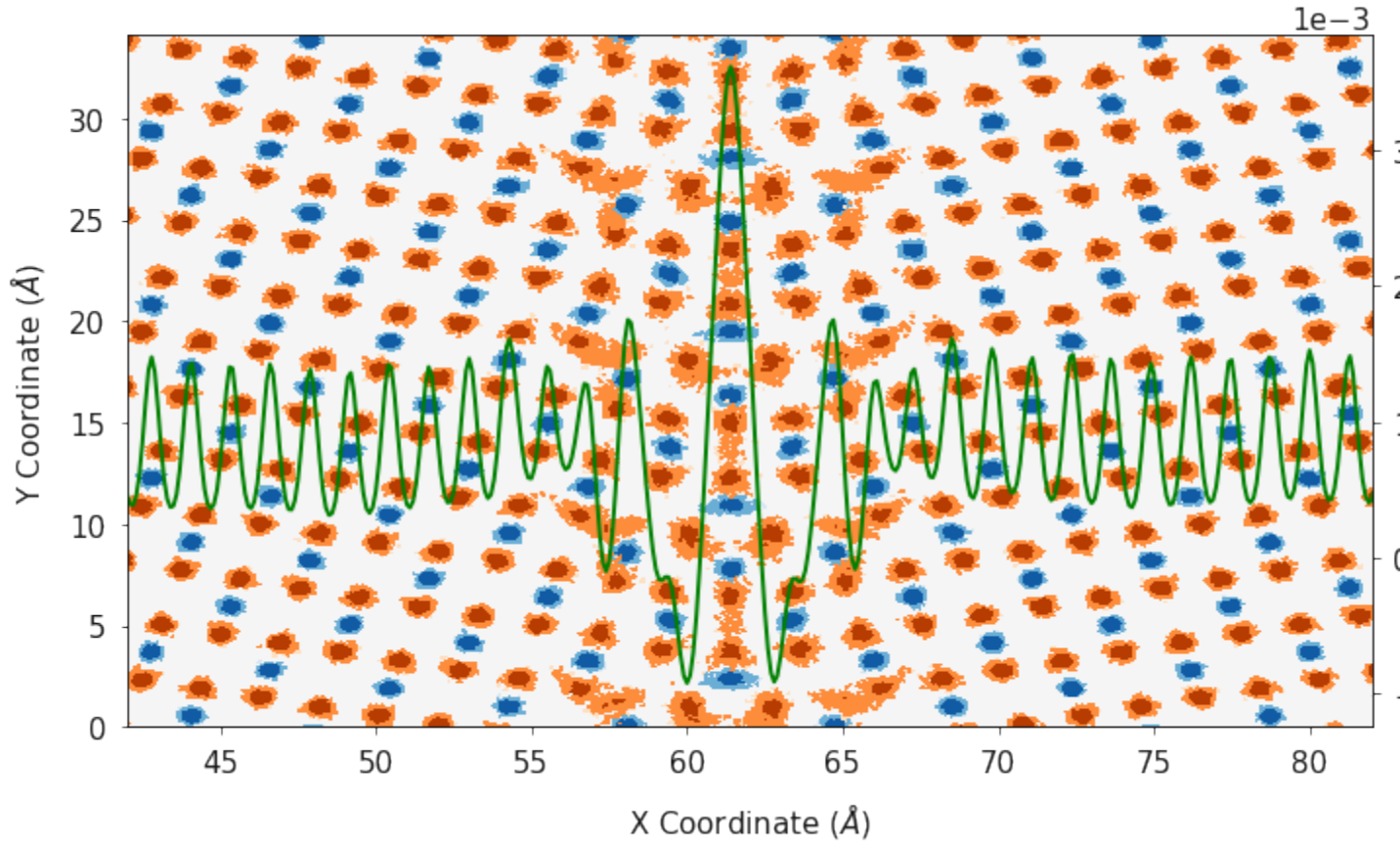

Finally, `polypy.plotting` has some functions that will generate a single contour plot for all species. This function requires the a list of x axes, a list of y axes, a list of two dimensional arrays corresponding to the x and y axes and a list of color palettes.

```
fig, ax = plotting.two_dimensional_density_plot_multiple_species([cx_2d, ox_2d], [cy_
↪2d, oy_2d],
                                                                 [cz_2d, oz_2d], [
↪"Blues", "Oranges"],
                                                                 log=True)
ax.set_xlim(42, 82)
plt.show()
```



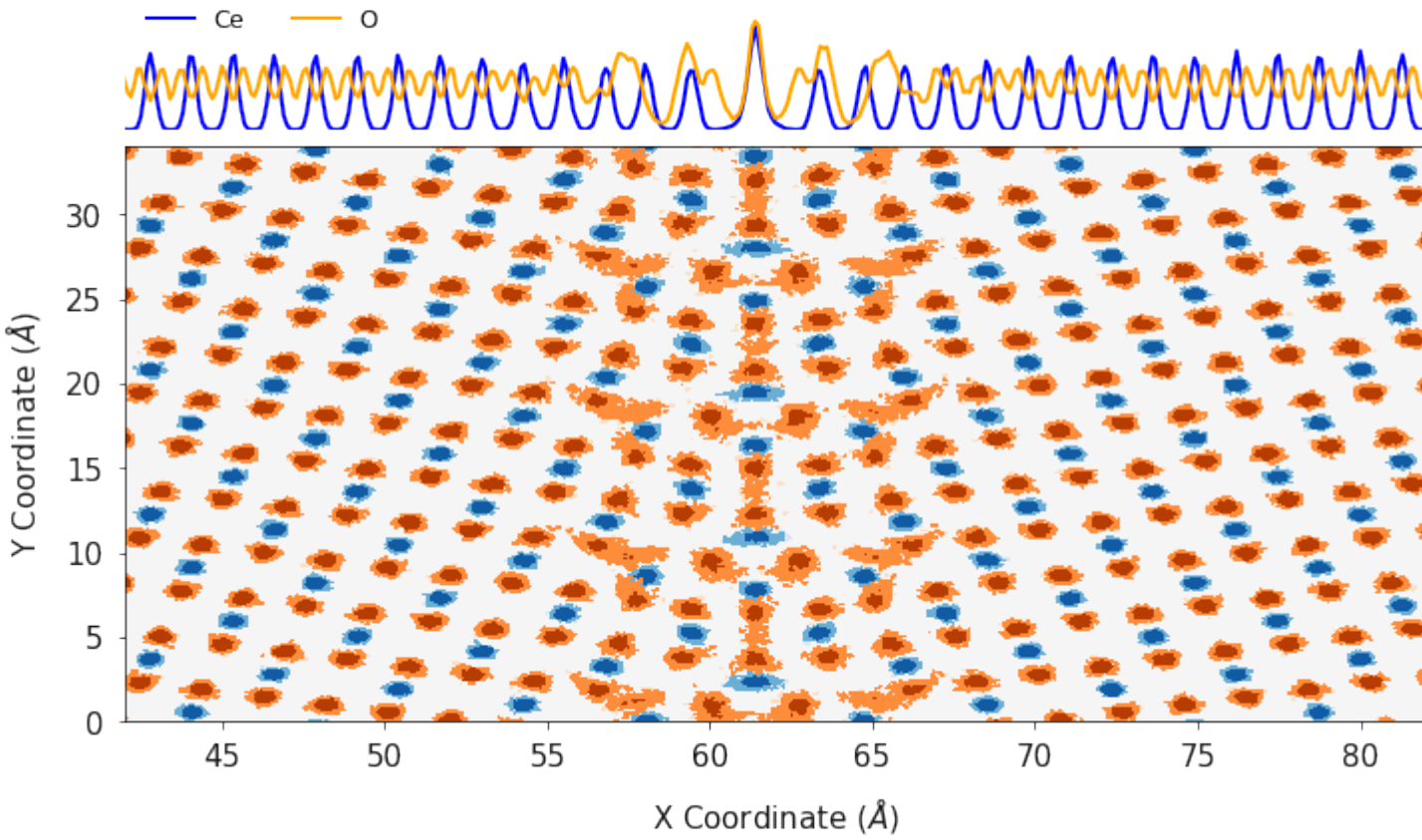

When analysing things like the electrostatic potential, it is useful to be able to view how the electrostatic potential changes with structure, it is very easy to use the `polypy.plotting` functions in conjunction with matplotlib to visualise the relationships.

```
fig, ax = plotting.two_dimensional_density_plot_multiple_species([cx_2d, ox_2d], [cy_
↪2d, oy_2d],
                                                                 [cz_2d, oz_2d], [
↪"Blues", "Oranges"],
                                                                 log=True)
ax.set_xlim(42, 82)
ax2 = ax.twinx()
ax2.plot(dx, electrostatic_potential, color="green")
ax2.set_ylabel("Electrostatic Potential (V)")
plt.show()
```




Finally, `polypy.plotting` can generate a contour plot showing the number density in one and two dimensions in a single plot. This function requires the a list of x axes, a list of y axes, a list of two dimensional arrays corresponding to the x and y axes, a list of color palettes, a list of labels and a list of line colors.

```
fig, ax = plotting.combined_density_plot_multiple_species(x_list=[cx_2d, ox_2d],
                                                          y_list=[cy_2d, oy_2d],
                                                          z_list=[cz_2d, oz_2d],
                                                          palette_list=["Blues",
↪"Oranges"],
                                                          label_list=['Ce', 'O'],
                                                          color_list=["blue", "orange
↪"],
                                                          log=True)
for axes in ax:
    axes.set_xlim(42, 82)
plt.show()
```




### Example 2 - Li, Al and Li vacancy swaps

In this example we will analyse a Monte Carlo simulation of Al doped lithium lanthanum titanate. It is possible to use molecular dynamics simulations to study defect segregation if the defects have a relatively high diffusion coefficient. One could randomly dope a configuration, run a long molecular dynamics simulation and then analyse the evolution of the defect locations. When the diffusion coefficient of your defect is very low, it is not possible to use molecular dynamics simulations to study defect segregation because you would need a huge MD simulation, in order to record enough statistics. Monte Carlo simulations allow you to perform unphysical moves and with a comparitively small Monte Carlo simulation, you can generate enough statistics to reliably study things like defect segregation.

In this example, we are analysing a MC simulation of Al in LLZO. $Al^{3+}$ has been doped on the $Li^{+}$ sites and charge compensating Li vacancies have been added. Ultimately, we want to calculate how the Al doping effects the Li conductivity, however without a representative distribution of Al/Li/Li vacancies we can't calculate a representative conductivity. After 10 ns of MD, the distribution of Al was unchanged, so Monte Carlo simulations with swap moves are needed to shake up the distribution. The following swap moves were used;

- Al <-> Li
- Al <-> $V_{Li}$
- Li <-> $V_{Li}$

ARCHIVE_LLZO is a short MC trajectory that we will analyse.

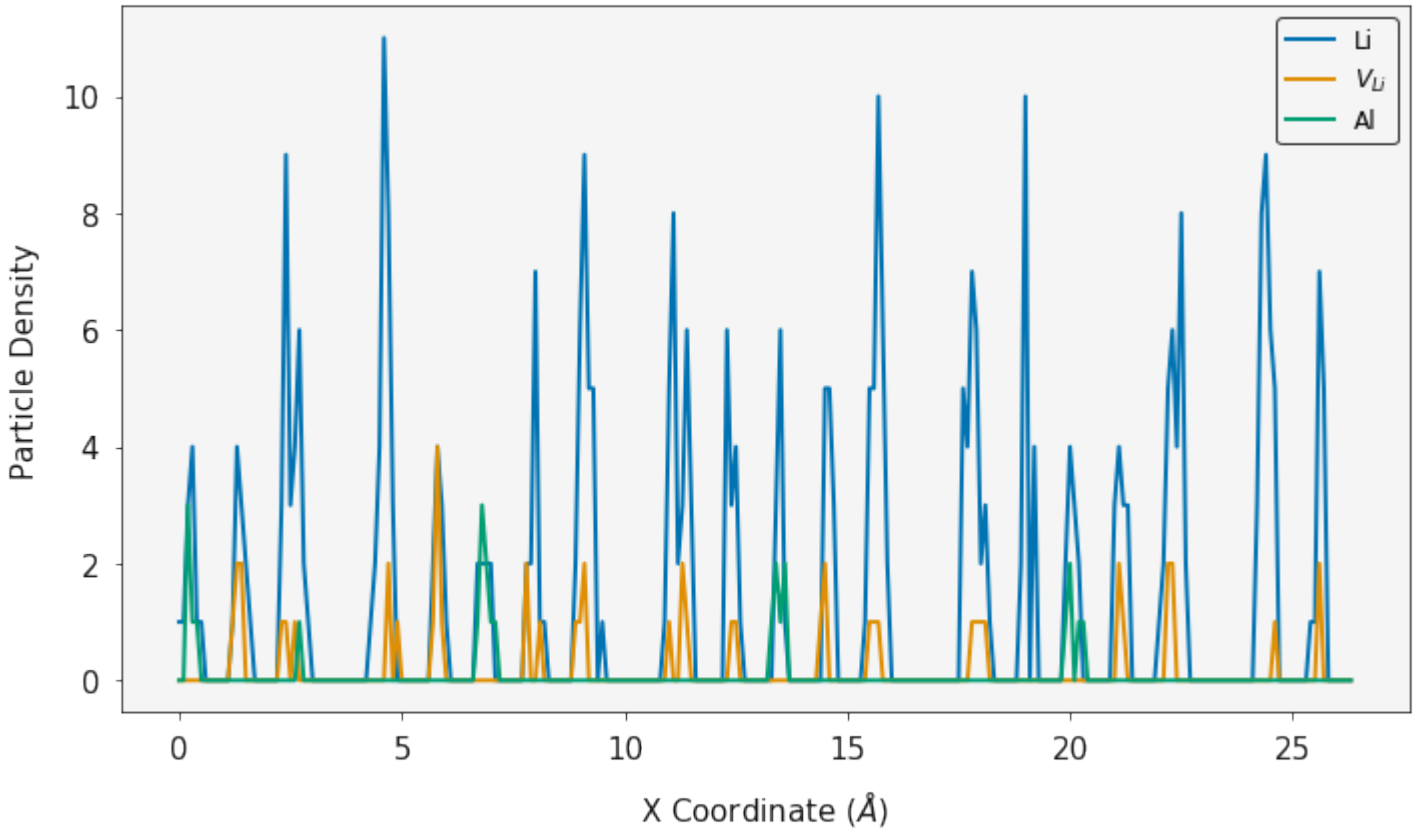
First we will extract and plot the configuration at the first timestep and then we will plot the positions across the whole simulation to see how the distributions have changed.

```
archive = Archive("../example_data/ARCHIVE_LLZO", ["LI", "AL", "LV"])
config_1 = archive.trajectory.get_config(1)
```

### Timestep 1

```
li_density = Density(config_1, atom="LI", histogram_size=0.1)
al_density = Density(config_1, atom="AL", histogram_size=0.1)
lv_density = Density(config_1, atom="LV", histogram_size=0.1)
```

```
lix, liy, li_volume = li_density.one_dimensional_density(direction="y")
alx, aly, al_volume = al_density.one_dimensional_density(direction="y")
lvx, lvy, lv_volume = lv_density.one_dimensional_density(direction="y")

ax = plotting.one_dimensional_density_plot([lix, lvx, alx], [liy, lvy, aly], ["Li", "
↪$V_{Li}$", "Al"])
plt.show()
```



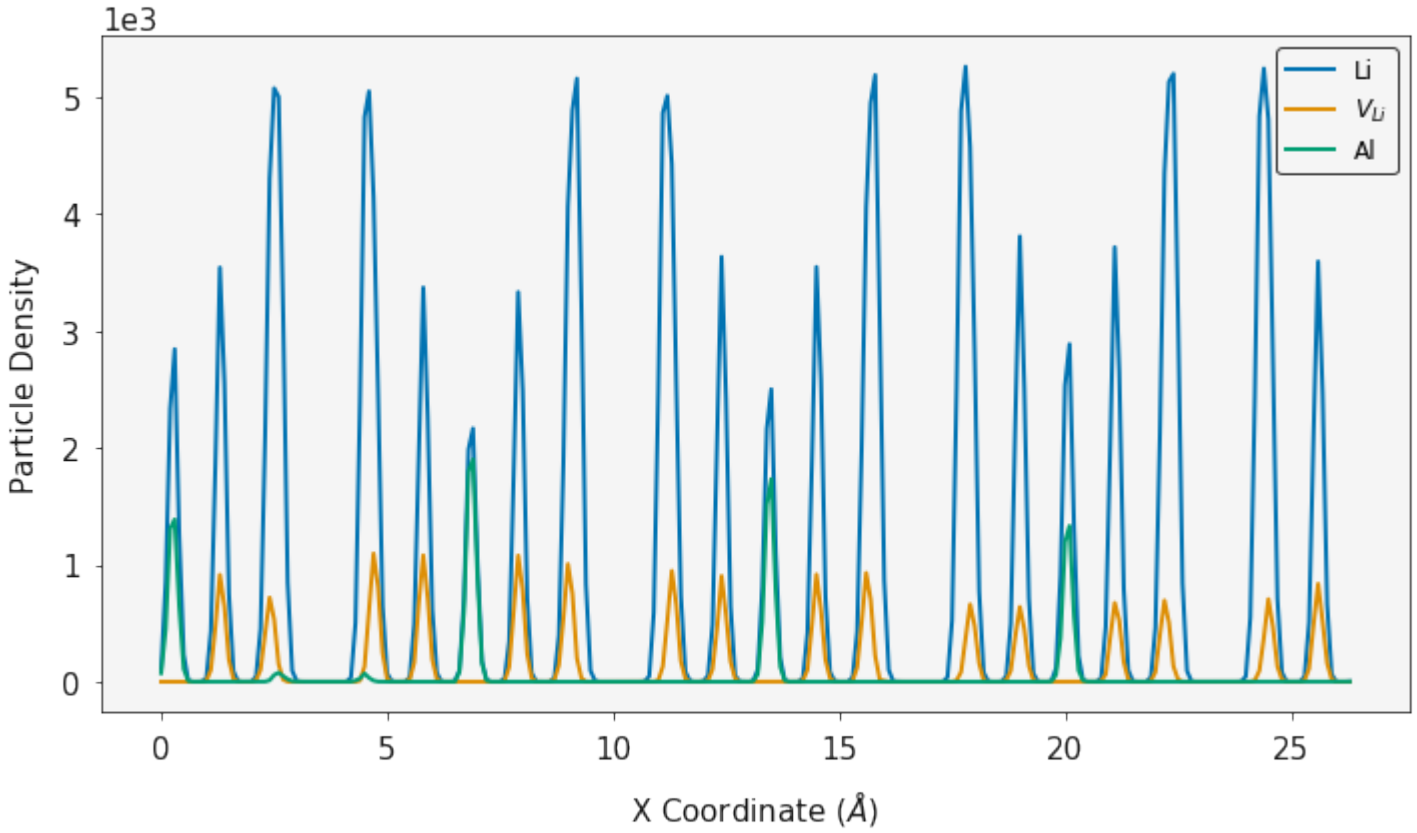

### Full Simulation

Disclaimer. This is a short snapshot of a simulation and is not fully equilibriated, however it provides an example of the `polypy` functionailty.

Interestingly, what we find is that the Al, Li and $V_{Li}$ tend to distribute in an even pattern within the structure. This is in sharp contrast to the distribution at the start of the simulation.

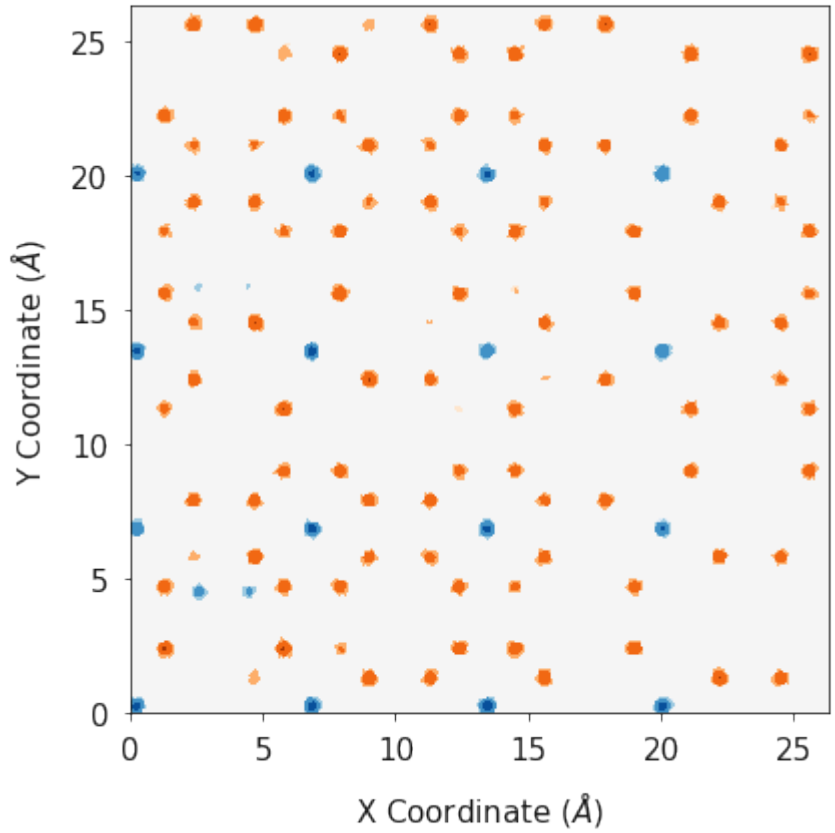
```
li_density = Density(archive.trajectory, atom="LI", histogram_size=0.1)
al_density = Density(archive.trajectory, atom="AL", histogram_size=0.1)
lv_density = Density(archive.trajectory, atom="LV", histogram_size=0.1)
```

```
lix, liy, li_volume = li_density.one_dimensional_density(direction="y")
alx, aly, al_volume = al_density.one_dimensional_density(direction="y")
lvx, lvy, lv_volume = lv_density.one_dimensional_density(direction="y")

ax = plotting.one_dimensional_density_plot([lix, lvx, alx], [liy, lvy, aly], ["Li", "
↪$V_{Li}$", "Al"])
plt.show()
```

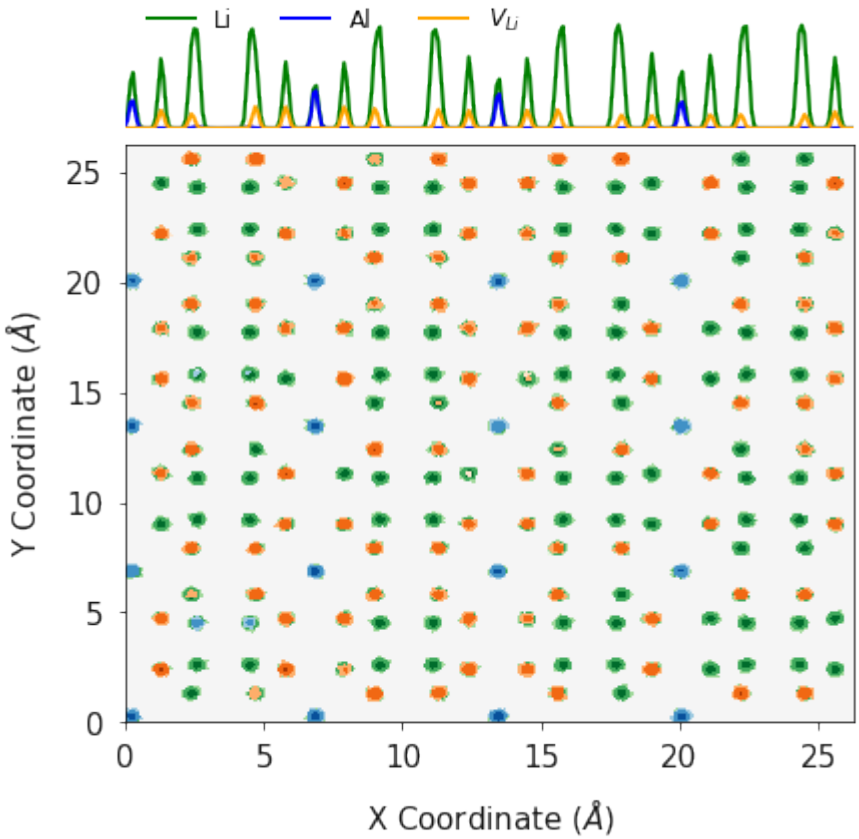



```
lix_2d, liy_2d, liz_2d, li_volume = li_density.two_dimensional_density(direction="z")
alx_2d, aly_2d, alz_2d, al_volume = al_density.two_dimensional_density(direction="z")
lvx_2d, lvy_2d, lvz_2d, lv_volume = lv_density.two_dimensional_density(direction="z")
```

```
fig, ax = plotting.two_dimensional_density_plot_multiple_species([alx_2d, lvx_2d],␣
↪[aly_2d, lvy_2d],
                                                                  [alz_2d, lvz_2d], [
↪"Blues", "Oranges"],
                                                                  log=True, figsize=(6,
↪ 6))
plt.show()
```




```
fig, ax = plotting.combined_density_plot_multiple_species(x_list=[lix_2d, alx_2d, lvx_
↪2d],
                                                          y_list=[liy_2d, aly_2d, lvy_
↪2d],
                                                          z_list=[liz_2d, alz_2d, lvz_
↪2d],
                                                          palette_list=["Greens",
↪"Blues", "Oranges"],
                                                          label_list=["Li", 'Al', '$V_
↪{Li}$'],
                                                          color_list=["green", "blue",
↪ "orange"],
                                                          log=True, figsize=(6, 6))
plt.show()
```




### Mean Squared Displacement MSD

### Mean Squared Displacement (MSD)

Molecules in liquds, gases and solids do not stay in the same place and move constantly. Think about a drop of dye in a glass of water, as time passes the dye distributes throughout the water. This process is called diffusion and is common throughout nature and an incredibly relevant property for materials scientists who work on things like batteries.

Using the dye as an example, the motion of a dye molecule is not simple. As it moves it is jostled by collisions with other molecules, preventing it from moving in a straight path. If the path is examined in close detail, it will be seen to be a good approximation to a random walk. In mathmatics a random walk is a series of steps, each taken in a random direction. This was analysed by Albert Einstein in a study of Brownian motion and he showed that the mean square of the distance travelled by a particle following a random walk is proportional to the time elapsed.

$$\left\langle r_i^2 \right\rangle = 6D_t + C$$

where

$$\left\langle r_i^2 \right\rangle = \frac{1}{3}\left\langle |r_i(t) - r_i(0)|^2 \right\rangle$$

where $\left\langle r^2 \right\rangle$ is the mean squared distance, t is time, $D_t$ is the diffusion rate and C is a constant. If $\left\langle r_i^2 \right\rangle$ is plotted as a function of time, the gradient of the curve obtained is equal to 6 times the self-diffusion coefficient of particle i. The state of the matter effects the shape of the MSD plot, solids, where little to no diffusion is occuring, has a flat MSD profile. In a liquid however, the particles diffusion randomly and the gradient of the curve is proportional to the diffusion coefficient.

### What is the mean squared displacement

Going back to the example of the dye in water, lets assume for the sake of simplicity that we are in one dimension. Each step can either be forwards or backwards and we cannot predict which. From a given starting position, what distance is our dye molecule likely to travel after 1000 steps? This can be determined simply by adding together the steps, taking into account the fact that steps backwards subtract from the total, while steps forward add to the total. Since both forward and backward steps are equally probable, we come to the surprising conclusion that the probable distance travelled sums up to zero.

By adding the square of the distance we will always be adding positive numbers to our total which now increases linearly with time. Based upon equation 1 it should now be clear that a plot of $\left\langle r_i^2 \right\rangle$ vs time with produce a line, the gradient of which is equal to 6D. Giving us direct access to the diffusion coefficient of the system.

```
from polypy import read as rd
from polypy.msd import MSD
from polypy.msd import RegionalMSD
from polypy import analysis
from polypy import utils as ut
from polypy import plotting
import numpy as np
import matplotlib.pyplot as plt
```

This example will use a short (50,000 steps), pre-prepared trajectory of bulk $CaF_2$. In reality we probably want a considerably longer simulation (~10,000,000 steps). Such simulations generate huge files (5GB) and the analysis would take too long for this tutorial.

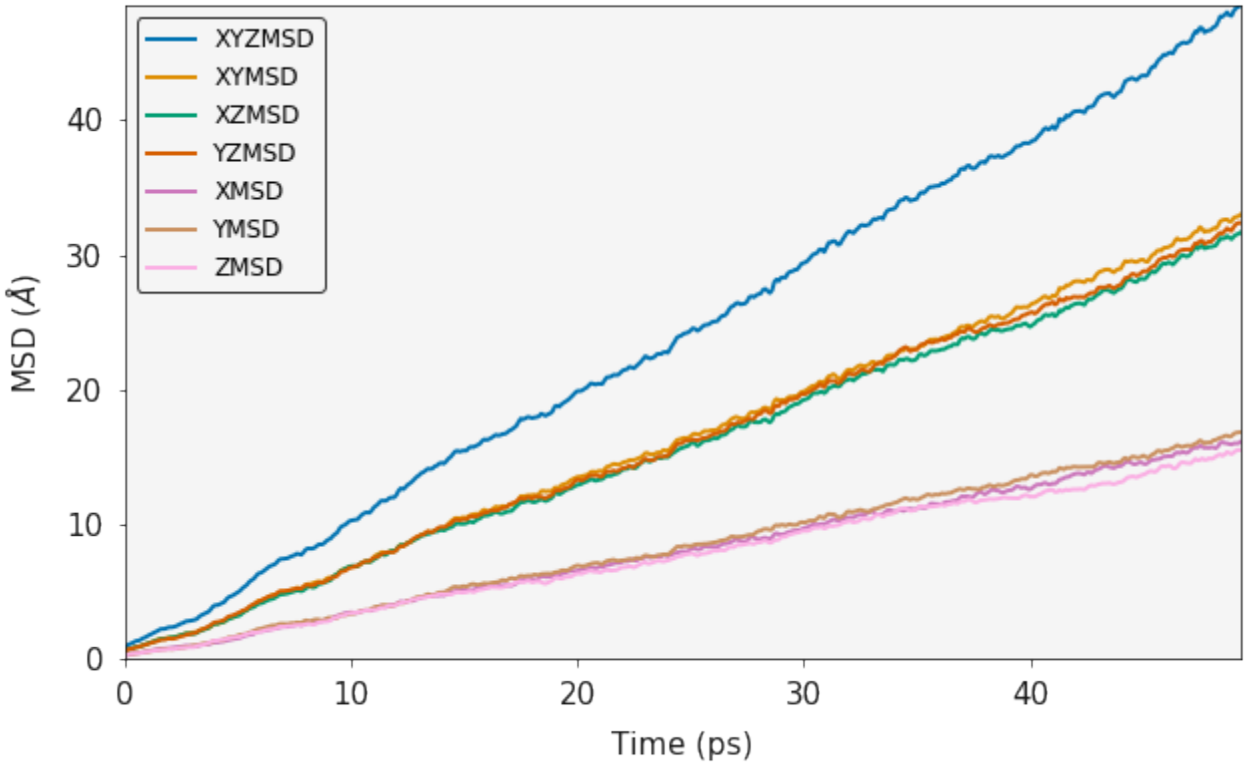
The first step is to read the history file to generate the data. The `HISTORY` class expects two things, the filename of the history file and a list of atoms to read. It will return a `polypy.read.Trajectory` object, which stores the the atom labels (`Trajectory.atom_labels`), datatype (`Trajectory.data_type`), cartesian coordinates (`Trajectory.cartesian_coordinates`), fractiona coordinates (`Trajectory.fractional_coordinates`), reciprocal lattice vectors (`Trajectory.reciprocal_lv`), lattice vectors (`Trajectory.lv`) cell lengths (`Trajectory.cell_lengths`), total atoms in the file (`Trajectory.atoms_in_history`), timesteps (`Trajectory.timesteps`), total atoms per timestep (`Trajectory.total_atoms`).

```
history_caf2 = rd.History("../example_data/HISTORY_CaF2", ["F"])
```

Once the data has been read into the code the MSD calculation can be performed using the `MSD` class. The code will return a `polypy.MSD.MSDContainer` object, which contains the MSD information.

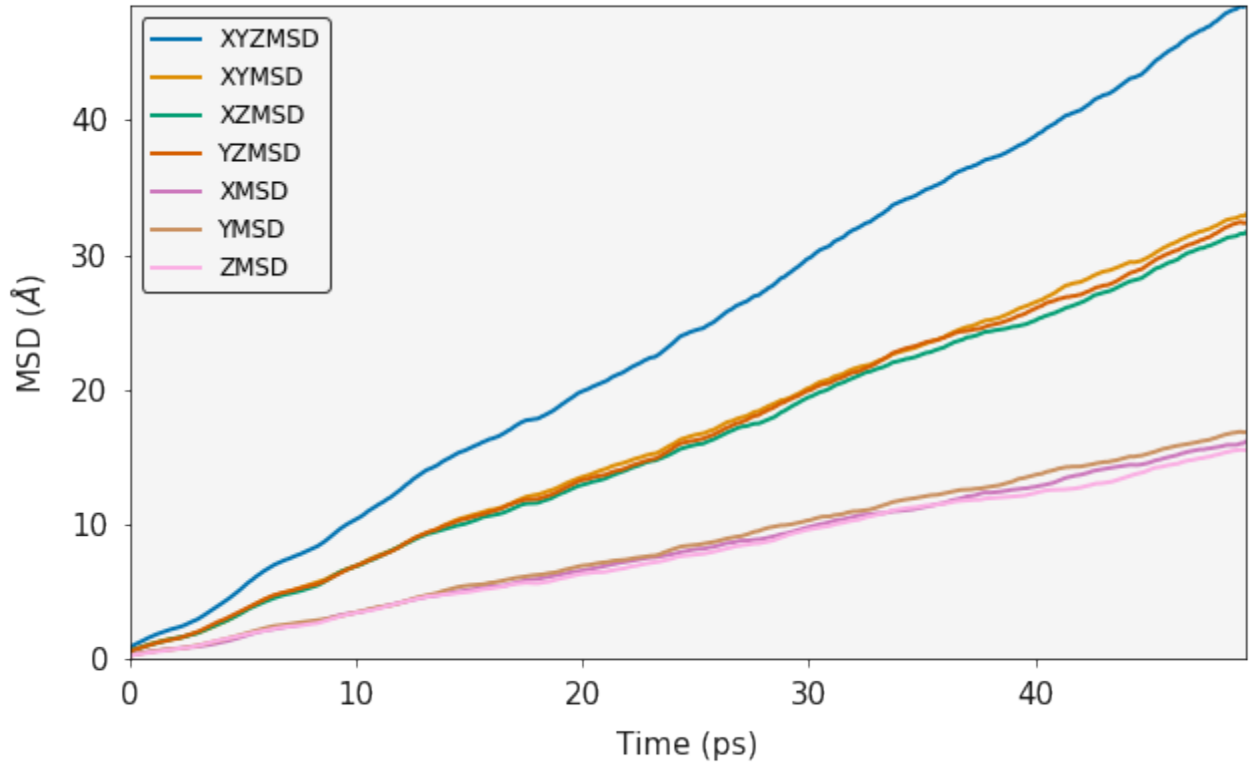
```
f_msd = MSD(history_caf2.trajectory, sweeps=2)

output = f_msd.msd()
```

```
ax = plotting.msd_plot(output)

plt.show()
```




MSD calculations require a large number of statistics to be considered representative. A full msd will use every single frame of the trajectory as a starting point and effectively do a seperate msd from each starting point, these are then averaged to give the final result. An MSD is technically an ensemble average over all sweeps and number of particles. The sweeps paramter is used to control the number of frames that are used as starting points in the calculation. For simulations with lots of diffusion events, a smaller number will be sufficient whereas simulations with a small number of diffusion events will require a larger number.

```
f_msd = MSD(history_caf2.trajectory, sweeps=10)

output = f_msd.msd()

ax = plotting.msd_plot(output)
plt.show()
```




```
print("Three Dimensional Diffusion Coefficient", output.xyz_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in X", output.x_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Y", output.y_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Z", output.z_diffusion_coefficient())
```

```
Three Dimensional Diffusion Coefficient 1.6078332646337548
One Dimensional Diffusion Coefficient in X 1.6045620180115865
One Dimensional Diffusion Coefficient in Y 1.6856414148385679
One Dimensional Diffusion Coefficient in Z 1.5332963610511103
```

Note: An MSD is supposed to be linear only after a ballistic regime and it usually lacks statistics for longer times. Thus the linear fit to extract the slope and thus the diffusion coefficient should be done on a portion of the MSD only. This can be accomplished using the *exclude_initial* and *exclude_final* parameters

```
print("Three Dimensional Diffusion Coefficient", output.xyz_diffusion_
↪coefficient(exclude_initial=50,

↪exclude_final=50))
print("One Dimensional Diffusion Coefficient in X", output.x_diffusion_
↪coefficient(exclude_initial=50,

↪exclude_final=50))
print("One Dimensional Diffusion Coefficient in Y", output.y_diffusion_
↪coefficient(exclude_initial=50,

↪exclude_final=50))
print("One Dimensional Diffusion Coefficient in Z", output.z_diffusion_
↪coefficient(exclude_initial=50,

↪exclude_final=50))
```

```
Three Dimensional Diffusion Coefficient 1.5912662736409342
One Dimensional Diffusion Coefficient in X 1.5862517497696607
One Dimensional Diffusion Coefficient in Y 1.6753802400942055
One Dimensional Diffusion Coefficient in Z 1.5121668310589353
```

### Arrhenius

It is then possible to take diffusion coefficients, calculated over a large temperature range and, using the Arrhenius equation calculate the activation energy for diffusion. Common sense and chemical intuition suggest that the higher the temperature, the faster a given chemical reaction will proceed. Quantitatively this relationship between the rate a reaction proceeds and its temperature is determined by the Arrhenius Equation. At higher temperatures, the probability that two molecules will collide is higher. This higher collision rate results in a higher kinetic energy, which has an effect on the activation energy of the reaction. The activation energy is the amount of energy required to ensure that a reaction happens.

$$k = A * e^{(-Ea/RT)}$$

where k is the rate coefficient, A is a constant, Ea is the activation energy, R is the universal gas constant, and T is the temperature (in kelvin).

### Ionic Conductivity

Usefully, as we have the diffusion coefficient, the number of particles (charge carriers) and the ability to calculate the volume, we can convert this data into the ionic conductivity and then the resistance.

$$\sigma = \frac{DC_F e^2}{k_B T}$$

where $\sigma$ is the ionic conductivity, D is the diffusion coefficient, $C_F$ is the concentration of charge carriers, which in this case if F ions, $e^2$ is the charge of the diffusing species, $k_B$ is the Boltzmann constant and T is the temperature.

The resitance can then be calculated according to

$$\Omega = \frac{1}{\sigma}$$

So the first step is to calculate the volume, the system voume module will do this from the given data.

```
volume, step = analysis.system_volume(history_caf2.trajectory)
average_volume = np.mean(volume[:50])
```

The number of charge carriers is just the total number of atoms.

```
sigma = analysis.conductivity(history_caf2.trajectory.total_atoms,
                        average_volume,
                        output.xyz_diffusion_coefficient(),
                        1500, 1)
```

```
print("Ionic Conductivity :", sigma)
```

```
Ionic Conductivity : 0.0008752727736501591
```

```
print("Resistivity :", (1 / sigma))
```

```
Resistivity : 1142.5009781004494
```

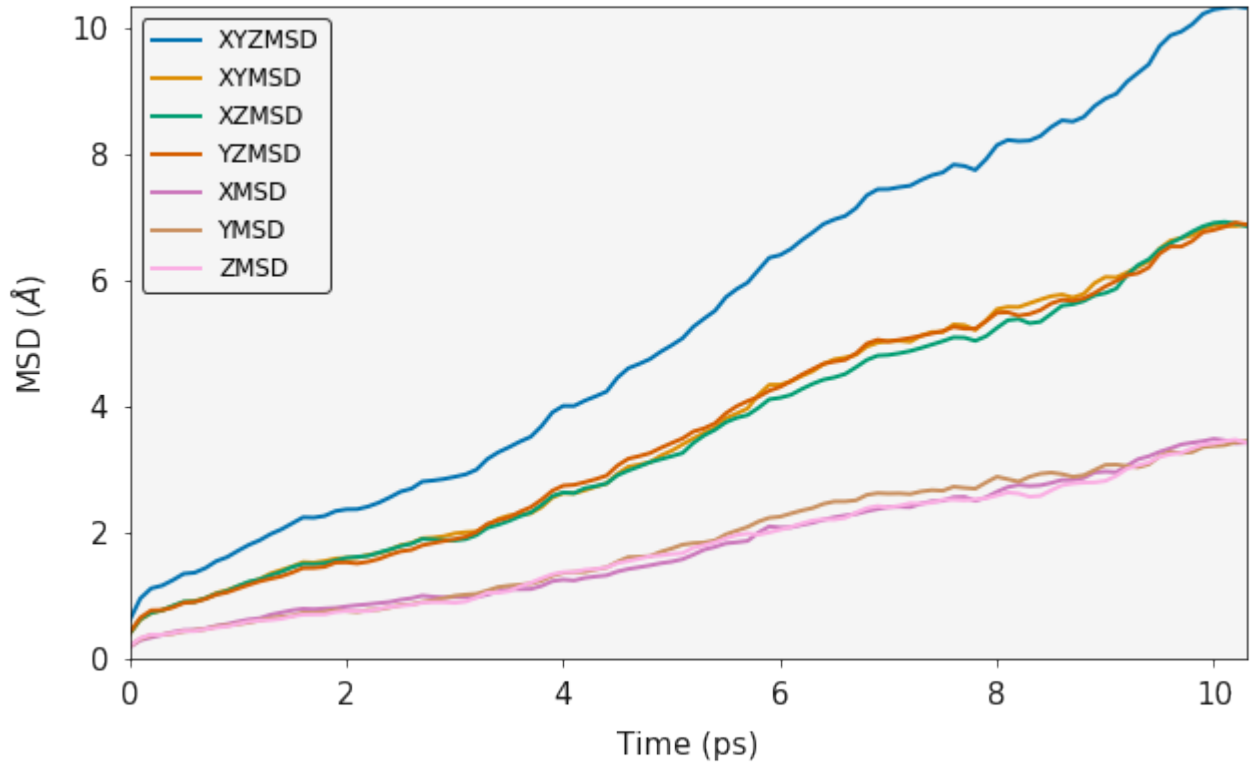
### Simulation Length

It is important to consider the lenght of your simulation (Number of steps). The above examples use a short trajectory but it is at a sufficient temperature that there are enough diffusion events to get a good MSD plot. The following example is of a very short simulation, you will hopefully note that the MSD plot is clearly not converged.

```
history_short = rd.History("../example_data/HISTORY_short", atom_list=["F"])
```

```
f_msd_short = MSD(history_short.trajectory, sweeps=2)

output = f_msd_short.msd()
```

```
ax = plotting.msd_plot(output)
plt.show()
```




```
print("Three Dimensional Diffusion Coefficient", output.xyz_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in X", output.x_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Y", output.y_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Z", output.z_diffusion_coefficient())
```

```
Three Dimensional Diffusion Coefficient 1.58656319093229
One Dimensional Diffusion Coefficient in X 1.5739020833099904
```

(continues on next page)

(continued from previous page)

```
One Dimensional Diffusion Coefficient in Y 1.630216356788139
One Dimensional Diffusion Coefficient in Z 1.5555711326987387
```

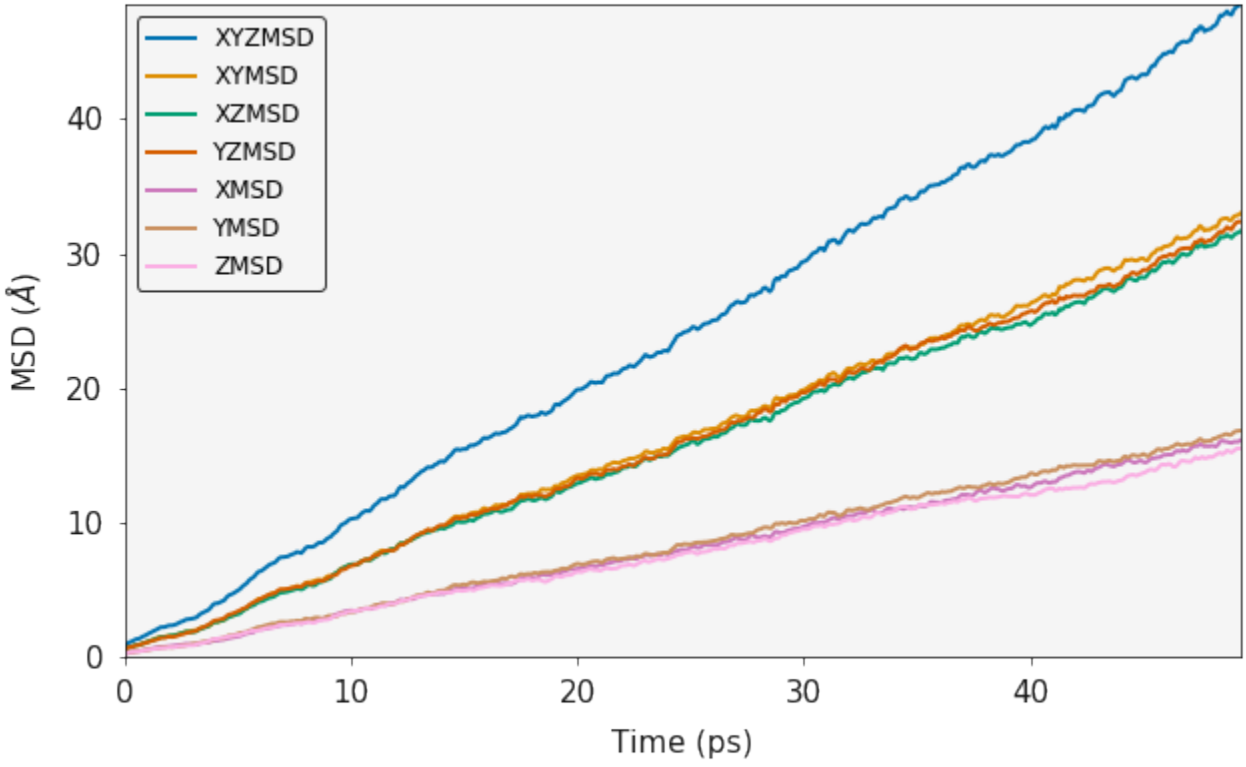
Amusingly, this actually does not seem to have a huge effect on the diffusion coefficient compared to the longer simulation. However these trajectories are from a $CaF_2$ simulation at 1500 K and there are thus a large number of diffusion events in the short time frame.

### State of Matter

It is possible to identify the phase of matter from the MSD plot.

The fluorine diffusion discussed already clearly shows that the fluorine sub lattice has melted and the diffusion is liquid like. Whereas, carrying out the same analysis on the calcium sub lattice shows that while the fluorine sub lattice has melted, the Calcium sub lattice is still behaving like a solid.

```
f_msd = MSD(history_caf2.trajectory, sweeps=2)

output = f_msd.msd()

ax = plotting.msd_plot(output)
plt.show()
```




### Regional MSD Calculations

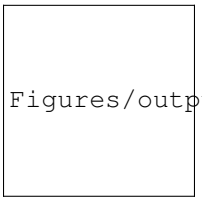
Often in solid state chemistry simulations involve defects, both structural e.g. grain boundaries, dislocations and surface, and chemical e.g. point defects. It is important to try and isolate the contributions of these defects to the

overall properties. Regarding diffusion, it could be imagined that a certain region within a structure will have different properties compared with the stoichiometric bulk, e.g. a grain boundary vs the grains, or the surface vs the bulk. polypy has the capability to isolate trajectories that pass within certain regions of a structure and thus calculate a diffusion coefficient for those regions.

In this example we will calculate the diffusion coefficient in a box between -5.0 and 5.0 in the dimension of the first lattice vector.

```
f_msd = RegionalMSD(history_caf2.trajectory, -5, 5, dimension="x")
output = f_msd.analyse_trajectory()
```

```
ax = plotting.msd_plot(output)

plt.show()
```




```
print("Three Dimensional Diffusion Coefficient", output.xyz_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in X", output.x_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Y", output.y_diffusion_coefficient())
print("One Dimensional Diffusion Coefficient in Z", output.z_diffusion_coefficient())
```

```
Three Dimensional Diffusion Coefficient 1.597047044241002
One Dimensional Diffusion Coefficient in X 1.6120172452124801
One Dimensional Diffusion Coefficient in Y 1.671268658071343
One Dimensional Diffusion Coefficient in Z 1.5078552294391845
```

### DLMONTE

```
archive = rd.Archive("../example_data/ARCHIVE_LLZO", atom_list=["O"])
```

```
f_msd = MSD(archive.trajectory, sweeps=2)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-20-2e636209fda5> in <module>
----> 1 f_msd = MSD(archive.trajectory, sweeps=2)


/opt/anaconda3/lib/python3.7/site-packages/polypy-0.7-py3.7.egg/polypy/msd.py in __
↪init__(self, data, sweeps)
    153             raise ValueError("ERROR: MSD can only handle one atom type.␣
↪Exiting")
    154         if data.data_type == "DL_MONTE ARCHIVE":
--> 155             raise ValueError("DLMONTE simulations are not time resolved")
    156         self.distances = []
    157         self.msd_information = MSDContainer()
```

(continues on next page)

(continued from previous page)

```
ValueError: DLMONTE simulations are not time resolved
```

### 6.6.5 API

#### polypy.analysis

Analysis functions

**class** polypy.analysis.**OneDimensionalChargeDensity**(*histogram_positions*, *atom_densities*, *atom_charges*, *histogram_volume*, *timesteps*)

Bases: object

The *polypy.analysis.OneDimensionalChargeDensity* class converts one dimensional number densitie into the charge density, electric field and electrostatic potential.

**Parameters**

- **histogram_positions** (array_like) – Histogram locations.
- **atom_densities** (list) – List of histograms.
- **atom_charges** (list) – List of atom charges.
- **histogram_volume** (float) – Volume of the histograms.
- **timesteps** (float) – Simulation timestep.

**calculate_charge_density**()

Calculates the charge density in one dimension.

**Returns** Charge density.

**Return type** charge_density (array_like)

**calculate_electric_field**()

Calculates the electric field.

**Returns** Electric field.

**Return type** e_field (array_like)

**calculate_electrostatic_potential**()

Calculates the electrostatic potential.

**Returns** Electrostatic potential.

**Return type** potential (array_like)

polypy.analysis.**conductivity**(*charge_carriers*, *volume*, *diff*, *temperature*, *hr*)

Calculate the ionic conductivity.

**Parameters**

- **charge_carriers** (float) – Number of charge carriers.
- **volume** (float) – Average cell volume.
- **diff** (float) – Diffusion coefficient.
- **temperature** (float) – Temperature.

- **hr** (float) – Haven ratio.

**Returns** Ionic conductivity.

**Return type** conductivity (float)

polypy.analysis.**system_volume**(*data*)
Calculate the volume at each timestep and return a volume as function of time.

**Parameters** **data** (*polypy.read.Trajectory*) – polypy Trajectory object.

**Returns** Volume as a function of timestep. step (array_like): Timestep.

**Return type** volume (array_like)

polypy.analysis.**two_dimensional_charge_density**(*atoms_coords*, *atom_charges*, *bin_volume*, *timesteps*)
Calculates the charge density in two dimensions.

**Parameters**

- **atoms_coords** (list) – List of atomic coordinates
- **atom_charges** (list) – List of atomic charges
- **bin_volume** (float) – Volume of histograms

**Returns** Charge density.

**Return type** charge_density (array_like)

### polypy.density

Density functions included with *polypy*. The Density class will determine generate a three dimensional grid that stores the total number of times that an atom spends within a xyz grid point during the simulation.

**class** polypy.density.**Density**(*data*, *histogram_size=0.1*, *atom=None*)
Bases: object

The *polypy.density.Density* class evaluates the positions of all atoms in the simulation.

**Parameters**

- **data** (*polypy.read.Trajectory*) – Object containing the information from the HISTORY or ARCHIVE files.
- **histogram_size** (float, optional) – Specifies the spacing between histograms.
- **atom** (str, optional) – Specifies the atom to calculate the density for.

**build_map**()
Constructs three dimensional grid of histogram_size * histogram_size * histogram_size. containing a count for how many times an atom passes through each histogram_size ** 3 cube.

**find_limits**()
Determine the upper and lower limits of the simulation cell in all three dimensions.

**one_dimensional_density**(*direction='x'*)
Calculate the particle density within one dimensional histograms of a structure.

**Parameters** **direction** (str) – The dimension perpendicular to the histograms.

**Returns** Locations of histograms. y (array_like): Size of histograms. bin_volume (float): Volume of histograms.

**Return type** x (array_like)

**two_dimensional_density**(*direction='x'*)
Calculate the particle density within two dimensional pixels of a structure.

**Parameters** **direction** (str) – The dimension normal to the pixels.

**Returns** Locations of one dimension of the pixels. y (array_like): Locations of one dimension of the pixels. z (array_like): Size of pixels. bin_volume (float): Volume of pixels.

**Return type** x (array_like)

**update_map**(*position*)
Determines the specific location of a given atom and adds it to the corresponding location in the three dimensional map of atomic positions.

### polypy.msd

MSD functions included with *polypy*. There are two MSD classes and one class to store the data generated from the MSD calculation. The first class performs a standard MSD calculation for the entire dataset while the second class will perform an MSD calculation within a specified region of the simulation cell.

**class** polypy.msd.**MSD**(*data*, *sweeps=1*)
Bases: object

The *polypy.msd.MSD* class calculates the mean squared displacements for a given atom.

**Parameters**

- **data** (*polypy.read.Trajectory*) – Object containing the information from the HISTORY or ARCHIVE files.
- **sweeps** (int, optional) – How many times should the starting timestep be changed. Default is 1.

**calculate_distances**(*trajectories*, *start*)
Calculates the distances.

**Parameters**

- **trajectories** (array_like) – Fractional coordinates.
- **start** (float) – Timestep to start the calculation.

**Returns** Distances. timestamp (array_like): Timesteps.

**Return type** distances (array_like)

**msd**()
Calculates the mean squared displacement for the trajectory.

**Returns** Object containing the information for the MSD.

**Return type** (*polypy.msd.MSDContainer*)

**one_dimension_square_distance**(*distances*, *run*)
Calculate the MSD in one dimension.

**Parameters**

- **distances** (array_like) – Distances.
- **run** (float) – Timestep to start the calculation.

**squared_displacements**(*distances*, *run*)
Calculates the squared distances.

**Parameters**

- **distances** (array_like) – Distances.
- **run** (float) – Timestep to start the calculation.

**three_dimension_square_distance**(*distances*, *run*)
Calculate the MSD in three dimensions.

**Parameters**

- **distances** (array_like) – Distances.
- **run** (float) – Timestep to start the calculation.

**two_dimension_square_distance**(*distances*, *run*)
Calculate the MSD in two dimensions.

**Parameters**

- **distances** (array_like) – Distances.
- **run** (float) – Timestep to start the calculation.

**class** polypy.msd.**MSDContainer**
Bases: object

The *polypy.msd.MSDContainer* class stores the output from the msd calculation.

**clean_data**()
Post msd the data is a list of time vs msd for each run. This needs to be normalised to give one continuous series of points.

**smooth_msd_data**(*x*, *y*)
Smooths the data from the smoothed msd function. The data consists of multiple msd runs but the data is unordered. This function will order the data and average all y values with equivalent x values.

**Parameters**

- **x** (array_like) – Time data.
- **y** (array_like) – MSD data.

**Returns** Time / MSD data.

**Return type** z (array_like)

**x_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the one dimensional x diffusion coefficient.

**Returns** x Diffusion coefficient.

**Return type** (float)

**xy_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the two dimensional xy diffusion coefficient.

**Returns** xy Diffusion coefficient.

**Return type** (float)

**xyz_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the three dimensional xyz diffusion coefficient.

**Returns** xyx Diffusion coefficient.

**Return type** (float)

**xz_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the two dimensional xz diffusion coefficient.

**Returns** xz Diffusion coefficient.

**Return type** (float)

**y_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the one dimensional y diffusion coefficient.

**Returns** y Diffusion coefficient.

**Return type** (float)

**yz_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the two dimensional yz diffusion coefficient.

**Returns** yz Diffusion coefficient.

**Return type** (float)

**z_diffusion_coefficient**(*exclude_initial=0*, *exclude_final=1*)
Calculates the one dimensional z diffusion coefficient.

**Returns** z Diffusion coefficient.

**Return type** (float)

**class** polypy.msd.**RegionalMSD**(*data*, *lower_boundary*, *upper_boundary*, *dimension='x'*, *sweeps=1*, *trajectory_length=100*)
Bases: object

The *polypy.msd.RegionalMSD* class calculates the mean squared displacements for a given atom in a specific region of a simulation cell.

**Parameters**

- **data** (*polypy.read.Trajectory*) – Object containing the information from the HISTORY or ARCHIVE files.
- **lower_boundary** (float) – Coordinate of the lower limit of the region of interest.
- **upper_boundary** (int, optional) – Coordinate of the upper limit of the region of interest.
- **dimension** (int, optional) – Direction perpedicular to the region of interest. Default is 'x'.
- **sweeps** (int, optional) – How many times should the starting timestep be changed. Default is 1.

**analyse_trajectory**()
Analyse the trajectory object.

**Returns** MSDContainer object - MSD information.

**Return type** msd_information (*polypy.msd.MSDContainer*)

**check_trajectory**(*trajectory*, *xc*)
Analyse the trajectory of an individual atom.

**Parameters**

- **trajectory** (*polypy.read.Trajectory*) – Trajectory object.
- **xc** (array_like) – Coordinates perpendicular to the region of interest.

**initialise_new_trajectory**()
Create a new MSDContainer object, specific to slice of a trajectory.

**Returns** MSDContainer object.

**Return type** new_trajectory (*polypy.msd.MSDContainer*)

**update_msd_info**(*container*)
Adds the information calculated for a single atom to the information of the whole trajectory.

**Parameters** **container** (*polypy.msd.MSDContainer*) – MSDContainer object - single atom.

## polypy.plotting

Plotting functions included with *polypy*.

polypy.plotting.**combined_density_plot**(*x*, *y*, *z*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Y Coordinate ($\\AA$)'*, *y2_lab='Number Density'*, *palette='viridis'*, *linecolor='black'*, *figsize=(10, 6)*, *log=False*)
Plots the distribution of an atom species in two dimensions and overlays the one dimensional density on top. Think of it as a combination of the two_dimensional_density_plot and one_dimensional_density_plot functions

**Parameters**

- **x** (`array like`) – x axis points - x axis coordinates.
- **y** (`array like`) – y axis points - y axis coordinates.
- **z** (`array like`) – z axis points - 2D array of points.
- **xlab** (`str`) – x axis label. Default is `"X Coordinate ($AA$)"`
- **ylab** (`str`) – y axis label. Default is `"Y Coordinate ($AA$)"`
- **y2_lab** (`str`) – second y axis label. Default is `"Particle Density"`
- **fig_size** (`tuple`) – Horizontal and veritcal size for figure (in inches). Default is `(10, 6)`.
- **log** (`bool`) – Log the z data or not? This can sometimes be useful but obviously one needs to be careful
- **drawing conclusions from the data.** (*when*) –

**Returns** Figure object (`list`): List of axes objects.

**Return type** (`matplotlib.Fig`)

polypy.plotting.**combined_density_plot_multiple_species**(*x_list*, *y_list*, *z_list*, *palette_list*, *label_list*, *color_list*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Y Coordinate ($\\AA$)'*, *figsize=(10, 6)*, *log=False*)
Plots the distribution of a list of atom species in two dimensions. Returns heatmaps for each species stacking on top of one another. It also plots the same density in one dimension on top of the heatmaps.

**Parameters**

- **x** (`list`) – x axis points - x axis coordinates.
- **y** (`list`) – y axis points - y axis coordinates.

- **z** (list) – z axis points - 2D array of points.
- **palette_list** (list) – Color palletes for each atom species.
- **label_list** (list) – List of species labels.
- **color_list** (list) – List of colors for one dimensional plot.
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Y Coordinate ($AA$)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** Figure object (list): List of axes objects.

**Return type** (matplotlib.Fig)

polypy.plotting.**electric_field_plot**(*x*, *y*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Electric Field (V)'*, *figsize=(10, 6)*)

Gathers the data and creates a line plot for the electric field in one dimension.

**Parameters**

- **x** (array like) – x axis points - position in simulation cell
- **y** (array like) – y axis points - electric field
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Electric Field (V)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**electrostatic_potential_plot**(*x*, *y*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Electrostatic Potential (V)'*, *figsize=(10, 6)*)

Gathers the data and creates a line plot for the electrostatic potential in one dimension.

**Parameters**

- **x** (array like) – x axis points - position in simulation cell
- **y** (array like) – y axis points - electrostatic potential
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Electrostatic Potential (V)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**line_plot**(*x*, *y*, *xlab*, *ylab*, *figsize=(10, 6)*)

Simple line plotting function. Designed to be generic and used in several different applications.

**Parameters**

- **x** (array like) – x axis points.

- **y** (array like) – y axis points.
- **xlab** (str) – x axis label.
- **ylab** (str) – y axis label.
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**msd_plot**(*msd_data*, *show_all_dimensions=True*, *figsize=(10, 6)*)

Plotting function for the mean squared displacements (MSD).

**Parameters**

- **msd_data** ():py:class:*polypy.msd.MSDContainer*) – MSD data.
- **show_all_dimensions** (bool) – Display all MSD data or the total MSD. Default is bool
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**one_dimensional_charge_density_plot**(*x*, *y*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Charge Density'*, *figsize=(10, 6)*)

Gathers the data and creates a line plot for the charge density in one dimension.

**Parameters**

- **x** (array like) – x axis points - position in simulation cell
- **y** (array like) – y axis points - charge density
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Charge Density"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**one_dimensional_density_plot**(*x*, *y*, *data_labels*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Particle Density'*, *figsize=(10, 6)*)

Plots the number density of all given species in one dimension.

**Parameters**

- **x** (list) – x axis points - list of numpy arrays containing x axis coordinates.
- **y** (list) – y axis points - list of numpy arrays containing y axis coordinates.
- **data_labels** (list) – List of labels for legend.
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Particle Density"

- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

polypy.plotting.**two_dimensional_charge_density_plot**(*x*, *y*, *z*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Y Coordinate ($\\AA$)'*, *palette='viridis'*, *figsize=(10, 6)*, *colorbar=True*, *log=False*)

Plots the charge density in two dimensions.

**Parameters**

- **x** (array like) – x axis points - x axis coordinates.
- **y** (array like) – y axis points - y axis coordinates.
- **z** (array like) – z axis points - 2D array of points.
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Y Coordinate ($AA$)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).
- **colorbar** (bool) – Include the colorbar or not.

**Returns** Figure object (matplotlib.axes.Axes): The axes with new plots.

**Return type** (matplotlib.Fig)

polypy.plotting.**two_dimensional_density_plot**(*x*, *y*, *z*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Y Coordinate ($\\AA$)'*, *palette='viridis'*, *figsize=(10, 6)*, *colorbar=True*, *log=False*)

Plots the distribution of an atom species in two dimensions.

**Parameters**

- **x** (array like) – x axis points - x axis coordinates.
- **y** (array like) – y axis points - y axis coordinates.
- **z** (array like) – z axis points - 2D array of points.
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Y Coordinate ($AA$)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).
- **colorbar** (bool) – Include the colorbar or not.
- **log** (bool) – Log the z data or not? This can sometimes be useful but obviously one needs to be careful
- **drawing conclusions from the data.** (*when*) –

**Returns** Figure object (matplotlib.axes.Axes): The axes with new plots.

**Return type** (matplotlib.Fig)

polypy.plotting.**two_dimensional_density_plot_multiple_species**(*x_list*, *y_list*, *z_list*, *palette_list*, *xlab='X Coordinate ($\\AA$)'*, *ylab='Y Coordinate ($\\AA$)'*, *y2_lab='Number Density'*, *figsize=(10, 6)*, *log=False*)

Plots the distribution of a list of atom species in two dimensions. Returns heatmaps for each species stacking on top of one another. This is limited to four species.

**Parameters**

- **x_list** (array like) – x axis points - x axis coordinates.
- **y_list** (array like) – y axis points - y axis coordinates.
- **z_list** (array like) – z axis points - 2D array of points.
- **palette_list** (array like) – Color palletes for each atom species.
- **xlab** (str) – x axis label. Default is "X Coordinate ($AA$)"
- **ylab** (str) – y axis label. Default is "Y Coordinate ($AA$)"
- **y2_lab** (str) – second y axis label. Default is "Particle Density"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).
- **log** (bool) – Log the z data or not? This can sometimes be useful but obviously one needs to be careful
- **drawing conclusions from the data.** (*when*) –

**Returns** Figure object (matplotlib.axes.Axes): The axes with new plots.

**Return type** (matplotlib.Fig)

polypy.plotting.**volume_plot**(*x*, *y*, *xlab='Timestep (ps)'*, *ylab='System Volume ($\\AA$)'*, *figsize=(10, 6)*)

Gathers the data and creates a line plot for the system volume as a function of simulation timesteps

**Parameters**

- **x** (array like) – x axis points - simulation timesteps
- **y** (array like) – y axis points - Volume
- **xlab** (str) – x axis label. Default is "Timestep (ps)"
- **ylab** (str) – y axis label. Default is "System Volume ($AA$)"
- **fig_size** (tuple) – Horizontal and veritcal size for figure (in inches). Default is (10, 6).

**Returns** The axes with new plots.

**Return type** (matplotlib.axes.Axes)

### polypy.read

Read functions of *polypy*. Herein contains classes to read DL_POLY HISTORY / CONFIG files and DL_MONTE ARCHIVE files. All of the data that is extracted from these files is stored in a trajectory class that is compatible with all three file types.

**class** polypy.read.**Archive**(*file*, *atom_list*)
Bases: object

The *polypy.read.Trajectory* class evaluates the positions of all atoms in a ARCHIVE.

**Parameters**

- **atom_list** (list) – List of unique atom names in trajectory.
- **datatype** (str) – Datatype of the original dataset e.g. DL_MONTE ARCHIVE.

**read_archive**()
Read a DL_MONTE ARCHIVE file line by line and updates a *polypy.read.Trajectory* object.

**class** polypy.read.**Config**(*file*, *atom_list*)
Bases: object

The *polypy.read.Trajectory* class evaluates the positions of all atoms in a CONFIG.

**Parameters**

- **atom_list** (list) – List of unique atom names in trajectory.
- **datatype** (str) – Datatype of the original dataset e.g. DL_POLY CONFIG.

**read_config**()
Read a DL_POLY HISTORY file line by line and updates a *polypy.read.Trajectory* object.

**class** polypy.read.**History**(*file*, *atom_list*)
Bases: object

The *polypy.read.Trajectory* class evaluates the positions of all atoms in the simulation.

**Parameters**

- **atom_list** (list) – List of unique atom names in trajectory.
- **datatype** (str) – Datatype of the original dataset e.g. DL_POLY HISTORY.

**read_history**()
Reads a DL_POLY HISTORY file line by line and updates a *polypy.read.Trajectory* object.

**class** polypy.read.**Trajectory**(*atom_list*, *datatype*)
Bases: object

The *polypy.read.Trajectory* class evaluates the positions of all atoms in the simulation.

**Parameters**

- **atom_list** (list) – List of unique atom names in trajectory.
- **datatype** (str) – Datatype of the original dataset
- **DL_POLY HISTORY or CONFIG.** (*e.g.*) –

**get_atom**(*atom*)
Isolates the trajectory for a specific atom type.

**Parameters** **atom** (str) – Atom label.

**Returns** Trajectory object for desired atom.

**Return type** atom_trajectory (*polypy.read.Trajectory*)

**get_config**(*timestep*)
Isolates a specific DL_POLY CONFIG from a HISTORY file.

**Parameters** **timestep** (int) – Timestep of desired CONFIG.

**Returns** Trajectory object for desired CONFIG.

**Return type** config_trajectory (*polypy.read.Trajectory*)

**remove_final_timesteps**(*timesteps_to_exclude*)
Removes timesteps from the end of a simulation

**Parameters** **timesteps_to_exclude** (int) – Number of timesteps to exclude

**Returns** Trajectory object.

**Return type** new_trajectory (*polypy.read.Trajectory*)

**remove_initial_timesteps**(*timesteps_to_exclude*)
Removes timesteps from the beggining of a simulation

**Parameters** **timesteps_to_exclude** (int) – Number of timesteps to exclude

**Returns** Trajectory object.

**Return type** new_trajectory (*polypy.read.Trajectory*)

### polypy.utils

Util functions

polypy.utils.**calculate_rcplvs**(*lv*)
Convert cartesian lattice vectors to the fractional lattice vectors

**Parameters** **lv** (array_like, optional) – Lattice vectors

**Returns** Reciprcocal lattice vectors lengths (array_like, optional): Cell lengths

**Return type** rcplvs (array_like, optional)

polypy.utils.**cart_2_frac**(*coord*, *lengths*, *rcplvs*)
Convert cartesian coordinates to the fractional coordinates

**Parameters**

- **coord** (array_like, optional) – Cartesian coordinates
- **lengths** (array_like, optional) – Cell lengths
- **rcplvs** (array_like, optional) – Reciprcocal lattice vectors

**Returns** Reciprcocal coordinates

**Return type** coords (array_like, optional)

polypy.utils.**pbc**(*rnew*, *rold*)
Periodic boundary conditions for an msd calculation

**Parameters**

- **rnew** (float, optional) – New atomic position
- **rold** (float, optional) – Previous atomic position

**Returns** Has the atom cross a PBC? rnew (float, optional): New position

**Return type** cross (`bool`, optional)

CHAPTER 7

# indices and tables

- genindex
- modindex
- search

# Python Module Index

## p

# Index

## P



## R



## S



## T



## U



## V



## X



## Y



## Z