Supplementary Material - Code

Candidate 8277T

May 15, 2021

Contents

1 Introduction

Here, we include the primary simulation and analysis routines used in this research. All molecular dynamics simulation work was conducted within LAMMPS, and the input scripts for this are detailed in Section 2; these may be ran in series or in parallel. These scripts require an input file, detailing the initial positions of all molecules. The generation of this file has been automated by a python script, listed in Section 3.

Analysis files are then detailed in Sections 4-5. All scripts in these Sections are in python 3.7, and were written from scratch by the candidate. All primary analysis methods are included in this section, as well as a selection of the primary plotting methods used.

2 Molecular Dynamics

The simulation routines are all contained within '*in_run.loop*', which contains the primary molecular dynamics routines and specifies the data output. The first action of this script, however, is to call '*in_run.setup*', which configures the simulation region and defines particle properties.

The restart file generated from this script is then read by '*in_run.loop*'. The main body of this script has two sections; a 'mixing' section where the simulation region is extended/contracted to adjust the volume fraction, followed by an 'equilibration' section where the system evolves at a constant volume fraction and equilibrium properties are measured. The main body of the system is then iterated over, to sample as many different volume fractions as desired.

2.1 *in.run_setup*

```
# VARIABLES
variable num        equal  1 # labelling results only
variable N          equal  1000
variable len        equal  15
variable box        equal  100 # cubic box of volume 'box'

print "variable N         equal  $N" #for phase plot code to extract

# PARAMETERS
units lj
atom_style molecular
neighbor 0.4 bin
neigh_modify every 1 delay 1 check yes
comm_modify vel yes
comm_modify cutoff 1.9 # This must be longer than the bond length below


# READ
read_data input_data_nunchucks_${N}_${len}_${box}.file
# This determines the initial position and velocity of all molecules

#  BONDS
bond_style harmonic
special_bonds lj 0 1 1
bond_coeff * 500.0 0.98
#0.98 is the distance between ball centres

# ANGLES
angle_style harmonic
angle_coeff * 500.0  180
angle_coeff $((v_len-1)/2) 0.1  180  # opening angle rigidity and value
```

```
# PAIR lj/cut --with interactions, to steady state
pair_style lj/cut 1.0
pair_modify shift yes
pair_coeff * * 1  1.0  1.12
#1.12 is cut-off for potential

write_restart restart.time*  # starting config for in.run_loop
```

2.2 *in.run_loop*

```
#SETUP
# Runs on the first loop only
shell del *.dump     # delete all dump files before first run
shell del restart.time*     # delete all current restart files before first run
shell lmp_serial -in in.run_setup     # Configure sys, generate first restart file

#VARIABLES
variable N          equal  1000  # number of molecules
variable len        equal  15  # length of each molecule
variable T          equal  0.5 # temperature of system in natural units
variable t_step     equal  0.005 #originally 0.005
variable mix_steps index  50000 50000 50000 50000 50000 50000
variable mix_tau    equal  ${t_step}*${mix_steps}
variable run_steps equal  2000000 # steps to run in order to reach steady state
variable run_tau    equal  ${t_step}*${run_steps} # this is the time in LJ units
#1000 particles averages 35,000 steps per minute

#LOOP START
label loop  #start loop from here
read_restart restart.time*
# this reads in latest restart file, * is wildcard for timestep


# THERMAL
timestep ${t_step}
thermo   100   #sampling period for thermodynamic data
thermo_style custom step time temp press pe ke etotal vol
thermo_modify norm no


# DUMP OUTPUT
dump  1  all custom 25000 output_T_${T}_time_*.dump id mol type x y z vx vy vz
# dumps all position/velocity data every 25000 steps
dump_modify 1 sort id

# FIX
fix 2 all nph iso 0.5 0.5 100
fix 3 all langevin ${T} ${T} 1 123456 # specify seed here
run ${mix_steps}

unfix 2
unfix 3
fix 4 all nve
fix 5 all langevin ${T} ${T} 1 123456 # specify seed here
fix recen1 all recenter INIT INIT INIT
```

```
# RUN
run ${run_steps}
write_restart restart.time*  # restart file gives initial config for next stage

clear
next mix_steps          # iterate to next value in mix_steps
jump in.run_loop loop   # rerun file from loop label

shell py phase_plot.py # runs thermodynamic analysis
shell py director_plot.py # generate order parameter plots
```

3 Initialisation

Here, we generate an input file detailing the initial positions of all nunchucks. I am grateful to Iria Pantazi for an initial version of this script, and the version presented below is based on that work. I have modified and extended the capabilities of that code, particularly to generate input files for an arbitrary dilute system (with user-defined particle numbers and particle lengths).

I further modified this script to produce a crystalline configuration in a simple cubic lattice with a predefined spacing between molecules. This was used access high volume fraction regions in simulations, and then sample decreasing volume fractions. While the functions from the original script remained unchanged (although many were not needed in this file), I made substantial changes to the main body of the script, which is replicated in Section 3.2.

3.1 *nunchucks.py*

```
#! /usr/bin/python

""" Written by: Iria Pantazi
    Updated by candidate to account for variable particle mol_length

    Accepts argument for mode (-g) and values for molecule number, length, box size
    Ie it can be run from shell via the command 'py nunchuck.py -g 2 10 20
"""

import time
import argparse

parser = argparse.ArgumentParser(description="generate nunchunks of 'mol_length' beads")
parser.add_argument(
    "-g",
    "--generate",
    nargs="+",
    default=None,
    help="generate new configuration from scratch.",
)
parser.add_argument(
    "-r",
    "--replot",
    nargs="+",
    help="Replots what has been written in the rawdata files.",
)
args = parser.parse_args()
import math
import numpy as np
from numpy import linalg as LA
```

```
from random import random

# import quaternion
from pyquaternion import Quaternion

# unchanged parameters for the beads:
mass = 1.0
dist = 0.98
rad = 0.56

elongation = 1
# aspect ratio of oblong base; y axis is E times longer than x and z axes


def scale_vector(j):
    """Returns scale factor for axis of index j"""
    if j == 1:
        return elongation  # extended y axis
    elif j == 0 or j == 2:
        return 1
    else:
        raise ValueError("Unexpected Index: recieved value " + str(j))


def within_box(ghost, box_lim):
    """Returns boolean to determine whether candidate particle lies within box"""
    flag = True
    toBreak = False
    for i in range(0, mol_length, mol_length - 1):  # check only the edge beads
        for j in range(3):
            if abs(ghost[i][j]) > scale_vector(j) * box_lim:
                flag = False
                toBreak = True
                break
        if toBreak == True:
            break
    return flag


def perform_rand_rot(ghost):
    new_ghost = np.zeros((mol_length, 3))
    ran_rot = Quaternion.random()
    for i in range(mol_length):
        new_ghost[i] = ran_rot.rotate(ghost[i])
    return new_ghost


def they_overlap(ghost, accepted, mol, rad):
    overlap = False
    lista = list(range(mol_length))
    break_flag = False

    for k in range(mol):
        for i in lista:
            for j in lista:
                dist = np.linalg.norm(accepted[mol_length * k + i] - ghost[j])
                if dist < 2 * rad:
                    break_flag = True
```

```
                    overlap = True
                    break
            if break_flag == True:
                break
        if break_flag == True:
            break
    return overlap


def gen_ghost(box_limit, dist):
    ghost = np.zeros((mol_length, 3))
    # center 0
    ghost[0][0] = 0
    ghost[0][1] = 0
    ghost[0][2] = 0
    # tail of the next (mol_length-1) beads
    for i in range(1, mol_length, 1):
        ghost[i] = ghost[0]
        ghost[i][2] += i * dist
    # preform random rotation
    ran_rot = Quaternion.random()
    for i in range(mol_length):
        ghost[i] = ran_rot.rotate(ghost[i])
    # preform random displacement
    ran_dis_x = (random() - 0.5e0) * 2.0 * (box_limit)
    ran_dis_y = (random() - 0.5e0) * 2.0 * elongation * (box_limit)
    ran_dis_z = (random() - 0.5e0) * 2.0 * (box_limit)
    for i in range(mol_length):
        ghost[i][0] += ran_dis_x
        ghost[i][1] += ran_dis_y
        ghost[i][2] += ran_dis_z
    return ghost


def plot_all(accepted, n_mol, box_lim):
    """General plotting function for plotting mode"""
    import matplotlib.pyplot as plt

    # import itertools
    import pylab

    # from   itertools import product,imap
    # from   matplotlib.backends.backend_pdf import PdfPages
    import matplotlib.pylab as pylab

    params = {
        "legend.fontsize": "large",
        "figure.figsize": (11, 6),
        "axes.labelsize": "x-large",
        "axes.titlesize": "x-large",
        "xtick.labelsize": "x-large",
        "ytick.labelsize": "x-large",
    }
    pylab.rcParams.update(params)
    import matplotlib.cm as cm
    from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
    from mpl_toolkits.axes_grid1.inset_locator import mark_inset
    from mpl_toolkits.mplot3d.proj3d import proj_transform
```

```
    from matplotlib.text import Annotation
    from mpl_toolkits.mplot3d import Axes3D

    colour = []
    x_list = [row[0] for row in accepted]
    y_list = [row[1] for row in accepted]
    z_list = [row[2] for row in accepted]
    cols = cm.seismic(np.linspace(0, mol_length, mol_length * n_mol) / mol_length)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.scatter(x_list, y_list, z_list, c=cols, marker="o", s=350)
    ax.set_xlim(-box_lim, box_lim)
    ax.set_ylim(-elongation * box_lim, elongation * box_lim)
    ax.set_zlim(-box_lim, box_lim)
    ax.grid(False)
    ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
    ax.set_xlabel("X axis")
    ax.set_ylabel("Y axis")
    ax.set_zlabel("Z axis")
    plt.show()
    return ()


def print_formatted_file(acc, n_molecules, box_limit, mass):
    """Generates the input file for MD simulations"""
    from shutil import copyfile

    with open("input_data.file", "w") as g:
        g.write("LAMMPS nunchunk data file \n\n")
        atoms = mol_length * n_molecules
        bonds = (mol_length - 1) * n_molecules
        angles = (mol_length - 2) * n_molecules
        dihedrals = 0 * n_molecules
        impropers = 0 * n_molecules
        g.write("%d  atoms          \n" % atoms)
        g.write("%d  bonds          \n" % bonds)
        g.write("%d  angles         \n" % angles)
        g.write("%d  dihedrals      \n" % dihedrals)
        g.write("%d  impropers    \n\n" % impropers)

        g.write("%d  atom types          \n" % mol_length)
        g.write("%d  bond types         \n" % (mol_length - 1))
        g.write("%d  angle types         \n" % (mol_length - 2))
        g.write("0  dihedral types \n")
        g.write("0  improper types \n\n")

        g.write("-%f %f xlo xhi  \n" % (box_limit, box_limit))
        g.write("-%f %f ylo yhi  \n" % (elongation * box_limit, elongation * box_limit))
        g.write("-%f %f zlo zhi  \n\n" % (box_limit, box_limit))

        g.write("Masses\n\n")
        for i in range(mol_length):
            g.write("\t %d  %s \n" % ((i + 1), mass))

        g.write("\nAtoms \n\n")
        for i in range(n_molecules):
            for j in range(mol_length):
                # N molecule-tag atom-type q x y z nx ny nz
```

```
                g.write(
                    "\t %d %d %d %s %s %s %d %d %d \n"
                    % (
                        mol_length * i + 1 + j,
                        i + 1,
                        1 + j,
                        acc[mol_length * i + j][0],
                        acc[mol_length * i + j][1],
                        acc[mol_length * i + j][2],
                        0,
                        0,
                        0,
                    )
                )

        g.write("\n\n")
        g.write("Bonds \n\n")
        for i in range(0, n_molecules, 1):
            for j in range(mol_length - 1):
                # N bond-type atom1-atom2
                g.write(
                    "\t %d %d %d %d \n"
                    % (
                        (mol_length - 1) * i + 1 + j,
                        1 + j,
                        mol_length * i + 1 + j,
                        mol_length * i + 2 + j,
                    )
                )

        g.write("\n\n")
        g.write("Angles \n\n")
        for i in range(n_molecules):
            for j in range(mol_length - 2):
                # N angle-type atom1-atom2(central)-atom3
                g.write(
                    "\t %d %d %d %d %d \n"
                    % (
                        (mol_length - 2) * i + 1 + j,
                        1 + j,
                        mol_length * i + 1 + j,
                        mol_length * i + 2 + j,
                        mol_length * i + 3 + j,
                    )
                )

    return ()


# ------------------------------------------------------------


if args.generate:  # ie argument -g

    import numpy as np
    from shutil import copyfile

    # inititalsation
```

```
n_molecules = int(args.generate[0])
mol_length = int(args.generate[1])
box_limit = float(args.generate[2]) / 2.0
rot_threshold = 500
ghost_mol = np.zeros((mol_length, 3))
accpt_mol = np.zeros((mol_length * n_molecules, 3))

start_time = time.time()
# first one is always accepted
ghost_mol = gen_ghost(box_limit, dist)
while within_box(ghost_mol, box_limit) == False:
    ghost_mol = gen_ghost(box_limit, dist)
for i in range(mol_length):
    accpt_mol[i] = ghost_mol[i]
mol = 1
while mol < n_molecules:
    # ----------------------generate ghost molecule--------------------
    ghost_mol = gen_ghost(box_limit, dist)
    while within_box(ghost_mol, box_limit) == False:
        ghost_mol = gen_ghost(box_limit, dist)
    # -------------check overlaping and perform a mx of 200 rots-----------
    flag = they_overlap(ghost_mol, accpt_mol, mol, rad)
    attempt_num = 0
    while flag == True:
        attempt_num += 1
        if attempt_num > rot_threshold:
            mol -= 1
            break
        ghost_mol = perform_rand_rot(ghost_mol)
        while within_box(ghost_mol, box_limit) == False:
            ghost_mol = gen_ghost(box_limit, dist)
        flag = they_overlap(ghost_mol, accpt_mol, mol, rad)
    # -------if not then add them to the list of accepted----------------
    if flag == False:
        for i in range(mol_length):
            accpt_mol[mol_length * mol + i] = ghost_mol[i]
    print(mol)
    # -------------------------move to next mol--------------------------
    mol += 1
end_time = time.time()
print("\n time for execution: " + str(end_time - start_time) + " seconds \n")

# ---------------------------plot all-------------------------------
plot_all(accpt_mol, n_molecules, box_limit)

# ---------------------------print all--------------------------------
print_formatted_file(accpt_mol, n_molecules, box_limit, mass)
print("done")

with open("accepted.dat", "w") as f:
    string_accpt_mol = str(accpt_mol).replace("[", "").replace("]", "")
    f.writelines(string_accpt_mol + "\n")

# --save a copy to be read by load_plot.py----------------------------
target_name = "rawdata_" + str(n_molecules) + "_" + str((box_limit) * 2)
copyfile("accepted.dat", target_name)

# ---------------- rename the input_data.file ----------------------
```

```
    src = "input_data.file"
    dst = (
        "input_data_nunchucks_"
        + str(n_molecules)
        + "_"
        + str(mol_length)
        + "_"
        + str(int((box_limit) * 2))
        + ".file"
    )
    copyfile(src, dst)

if args.replot:  # ie argument -r
    n_molecules, mol_length, box_limit = args.replot
    n_molecules = int(n_molecules)
    mol_length = int(mol_length)
    box_limit = float(box_limit)
    infiles = "rawdata_" + str(n_molecules) + "_" + str((box_limit) * 2)

    # ------------------- initialisation ---------------------
    accepted = np.zeros((mol_length * n_molecules, 3))
    i = 0

    # ------------------ call from functions.py --------------
    with open(infile, "r") as g:
        for row in g.readlines():
            accepted[i][0], accepted[i][1], accepted[i][2] = row.split()
            i += 1
        plot_all(accepted, n_molecules, box_limit)
```

3.2 *nunchucks_crystalline.py*

```
""" Written by candidate to produce crystalline array of molecules
    Based on code for dilute configurations by Iria Pantazi

    Takes 4 input arguments (mol_length, x_num, y_num, z_num),
    as well as flag for mode (-g for generator). I.e. py nunchuck.py -g 15 16 4 16
    Note that i_num is the number of molecules in the i-th axis
    This script automically aligns molecules along the y axis
"""

# ... Unchanged preamble, plot_all() and print_formatted_file() functions ...

if args.generate:  # ie argument -g

    import numpy as np
    from shutil import copyfile

    # inititalisation
    mol_length = int(args.generate[0])
    x_num = int(args.generate[1])
    y_num = int(args.generate[2])
    z_num = int(args.generate[3])
    n_molecules = x_num * y_num * z_num
    spacer = 2 * rad - dist

    end_to_end_length = ((mol_length - 1) * dist) + (2 * rad)
```

```
# ie for 9 bonds, plus two ends of the molecule

accpt_mol = np.zeros((mol_length * n_molecules, 3))
shuffle_molecules = False

start_time = time.time()
for n in range(n_molecules):
    mol_pos_index = [
        (n // y_num) % x_num,
        n % y_num,
        n // (x_num * y_num),
    ]
    # print(mol_pos_index)
    for i in range(mol_length):
        # iterate over atoms in each molecule
        accpt_mol[mol_length * n + i, :] = [
            mol_pos_index[0] * (1 + spacer),
            mol_pos_index[1] * (end_to_end_length + spacer) + i * dist,
            mol_pos_index[2] * (1 + spacer),
        ]
        # aligns all molecules along the long y axis
accpt_mol = accpt_mol + rad  # constant offset

if shuffle_molecules:
    rng = np.random.default_rng()
    print(accpt_mol)
    rng.shuffle(accpt_mol, axis=0)
    print(accpt_mol)

end_time = time.time()
print("\n time for execution: " + str(end_time - start_time) + " seconds \n")

# ----------------------------plot all-------------------------------
# plot_all(accpt_mol, n_molecules, box_limit)

# ----------------------------print all--------------------------------
print_formatted_file(accpt_mol, n_molecules, mass)
print("done")

# ---------------- rename the input_data.file ------------------------
box_volume = (
    (x_num * (1 + spacer))
    * (y_num * (end_to_end_length + spacer))
    * (z_num * (1 + spacer))
)  # for inclusion in filename if desired
src = "input_data.file"
dst = "input_data_nunchucks_" + str(n_molecules) + "_" + str(mol_length) + ".file"
copyfile(src, dst)
```

4 Order Parameter Calculation

There are three main files here. The first is '*phase_plot.py*', which plots the thermodynamic data output during the equilibration periods; significant deviations here typically indicate the occurrence of a phase transition during that equilibration stage. This file also defines a function to determine the volume fraction of the system, which is used throughout our other functions.

Two separate files then determine the nematic and smectic order parameters,using the computational methods

outlined within the main report. Much of the data extraction code in '*smectic_plot.py*' is identical to that of '*nematic_plot.py*', and so not replicated for brevity.

4.1 *phase_plot.py*

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter1d  # for rolling average


def vol_frac(volume_data, mol_length=10, N=1000):
    """Returns array of volume fraction data from volume array

    Input list of volumes to be evaluated (volume data)
    Also takes molecule length (default 10) and number of molecules (default 1000)
    """
    return np.reciprocal(volume_data) * (mol_length * N * (np.pi * 0.56 ** 2))
    # See OneNote details of form to use here


if __name__ == "__main__":
    LOOP_VAR_NAME = "mix_steps"  # allows for loops under different names

    FILE_NAME = "log.lammps"
    start_lines = []
    end_lines = []
    blank_lines = []
    # this gives the number of blank lines encountered by each Loop line

    data_file = open(FILE_NAME, "r")
    blank_lines_count = 0
    for i, line in enumerate(data_file):
        if line.isspace():
            blank_lines_count += 1  # running count of num of blank lines
        if (
            "variable " + LOOP_VAR_NAME in line
        ):  # to extract independant variable values of loop variable
            loop_var_values = []
            for t in line.split():  # separate by whitespace
                try:
                    loop_var_values.append(float(t))
                except ValueError:
                    pass  # any non-floats in this line are ignored
        if "variable N" in line:  # to extract independant variable value of N
            for t in line.split():  # separate by whitespace
                try:
                    N = float(t)
                except ValueError:
                    pass  # any non-floats in this line are ignored
        if "variable len" in line:  # to extract length of molecule
            for t in line.split():  # separate by whitespace
                try:
                    mol_length = float(t)
                except ValueError:
                    pass  # any non-floats in this line are ignored

        if (
            "Step Time" in line
```

```
    ):  # always preceeds data section, can ammend searched string
        start_lines.append(i)  # start of final data readout
    if "Loop time" in line:
        end_lines.append(i)  # end of final data readout
        blank_lines.append(
            blank_lines_count
        )  # end of final data readout without blank lines

if not 2 * len(loop_var_values) == len(start_lines):
    print(
        "Warning: Number of loop variable values does not match"
        + " the number of equillibrium runs. Check whether"
        + " you are reading in the correct loop variable in line 17"
    )
    print("Number of loop variable values: " + str(len(loop_var_values)))
    print("Number of equillibrium runs: " + str(len(start_lines)))

last_line = i  # last line number in file
tot_blank_lines = blank_lines_count  # total blank lines in file
blank_lines_left = [
    tot_blank_lines - b for b in blank_lines
]  # blank lines after each 'Loop' line
end_lines_adj = [
    e_i + b_i for e_i, b_i in zip(end_lines, blank_lines_left)
]  # sum two lists
"""skip_footer doesn't count blank lines, but the iteraction above does.
Therefore, we add in the number of  blank lines below this end line, to
give an adjusted end line that accounts for these extra blank lines"""

data_file.close()

final_values = np.zeros(
    (4, int(len(start_lines) / 2))
)  # in form Temp/Press/PotEng/Volume
# These are the values at equillibrium. Currently output as mean values

total_loop_var = 0
for i in range(1, len(start_lines), 2):
    # taking every other i to only get equillibrium values
    # (each N has a thermalisation and an equillibrium run)

    j = int(
        (i - 1) / 2
    )  # giving range from 0 upwards in integer steps to compare to N_list
    data = np.genfromtxt(
        fname=FILE_NAME,
        names=True,
        skip_header=start_lines[i],
        skip_footer=last_line - end_lines_adj[i] + 1,
        comments=None,
    )

    total_loop_var += loop_var_values[j]
    plt.plot(
        data["Step"] - np.min(data["Step"]),  # so time starts from zero
        # data["Press"],
        uniform_filter1d(data["Press"], size=int(5e2)),  # rolling average
        label=LOOP_VAR_NAME + "= " + str(int(total_loop_var)),
```

```
        )

        end_index = int(len(data["Press"]))
        start_index = int(0.9 * end_index)
        # selecting only final 10% of data to average, so sys has reached equilibrium

        final_values[0, j] = loop_var_values[j]
        final_values[1, j] = np.mean(data["Press"][start_index:end_index])
        final_values[2, j] = np.mean(data["PotEng"][start_index:end_index])
        final_values[3, j] = np.mean(data["Volume"][start_index:end_index])

    vol_frac_data = vol_frac(final_values[3, :])
    print("Volume Fractions: " + str(vol_frac_data))
    print(data.dtype.names)

    plt.xlabel("Time Step")
    plt.ylabel("Pressure (Natural Units)")
    plt.title("Evolution of Thermodynamic Variables at different " + LOOP_VAR_NAME)
    plt.legend()
    plt.savefig("pressureplot_frac.png")
    plt.show()
```

4.2 *director_plot.py*

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter1d  # for rolling average
from phase_plot import vol_frac

FILE_ROOT = "output_T_0.5_time_"  # two underscores to match typo in previous code
SAMPLING_FREQ = 10  # only samples one in X files (must be integer)

plt.rcParams.update({"font.size": 13})  # for figures to go into latex


# READ PARAMETER VALUES FROM LOG FILE

FILE_NAME = "log.lammps"
log_file = open(FILE_NAME, "r")
mix_steps_values = []

for i, line in enumerate(log_file):
    """For loop iteratres over every line in file to find the required variables.

    However, because the dump_interval occurs last (in the main body not the preamble)
    we break at this point to avoid reading the whole file unnecessarily."""

    if "variable N" in line:  # to extract independant variable value of N
        for t in line.split():  # separate by whitespace
            try:
                N_molecules = int(t)
            except ValueError:
                pass  # any non-floats in this line are ignored

    if "variable len" in line:  # to extract length of molecule
        for t in line.split():  # separate by whitespace
            try:
```

```
                mol_length = int(t)
            except ValueError:
                pass  # any non-floats in this line are ignored

    if "dump" and "all custom" in line:  # to extract time interval for dump
        for t in line.split():  # separate by whitespace
            try:
                dump_interval = int(t)
                # interval is the last integer in that line
            except ValueError:
                pass
        break  # got all data, break from for loop

    if "variable mix_steps" in line:  # to extract mix_steps (for shrinking)
        run_num = 0  # counts number of runs
        for t in line.split():
            try:
                mix_steps_values.append(int(t))
                run_num += 1
            except ValueError:
                pass

    if "variable run_steps" in line:  # to extract run_steps (for equilibration)
        for t in line.split():
            try:
                equilibrium_time = int(t)  # time per run
            except ValueError:
                pass
        # this comes up first in file so no searching variable here

log_file.close()

tot_mix_time = sum(mix_steps_values)
run_time = tot_mix_time + run_num * equilibrium_time
time_range = range(0, int(run_time), int(dump_interval * SAMPLING_FREQ))
print(
    "N_molecules, run_time, dump_interval = "
    + str((N_molecules, run_time, dump_interval))
)


def order_param(data):
    """Input data in array of size Molecule Number x 3 x 3

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Second index gives particle number within molecule (first/last)
    Third index gives the component of the position (x,y,z)

    Method for calculation of Order Param given by Eppenga (1984)
    """
    directors = data[:, 1, :] - data[:, 0, :]  # director vector for each molecule
    norm_directors = directors / np.linalg.norm(directors, axis=1).reshape(
        -1, 1
    )  # reshape allows broadcasting
    M_matrix = np.zeros((3, 3))
    for i, j in np.ndindex(M_matrix.shape):
        M_matrix[i, j] = (
```

```
            np.sum(norm_directors[:, i] * norm_directors[:, j]) / N_molecules
        )
    M_eigen = np.linalg.eigvals(M_matrix)
    Q_eigen = (3 * M_eigen - 1) / 2
    return max(Q_eigen)  # largest eigenvalue corresponds to traditional order parameter


# READ MOLECULE POSITIONS

order_param_values = np.zeros(len(time_range))
volume_values = np.full(len(time_range), np.nan)  # new array of NaN
for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(FILE_ROOT + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    box_volume = 1
    rod_positions = np.zeros((N_molecules, 2, 3))
    """Indices are Molecule Number/ First (0) or Last (1) atom/ Positional coord index"""

    for line in data_file:
        if "ITEM: BOX" in line:  # to start reading volume data
            extract_box_data = True
            extract_atom_data = False
            continue  # don't attempt to read this line

        if "ITEM: ATOMS" in line:  # to start reading particle data
            extract_box_data = False
            extract_atom_data = True
            continue

        if extract_box_data and not extract_atom_data:
            # evaluate before particle values
            # each line gives box max and min in a single axis
            box_dimension = []
            for d in line.split():  # separate by whitespace
                try:
                    box_dimension.append(float(d))
                except ValueError:
                    pass  # any non-floats in this line are ignored
            box_volume *= box_dimension[1] - box_dimension[0]
            # multiply box volume by length of this dimension of box

        if extract_atom_data and not extract_box_data:
            # evaluate after box dimension collection
            # each line is in the form "id mol type x y z vx vy vz"
            particle_values = []
            for t in line.split():  # separate by whitespace
                try:
                    particle_values.append(float(t))
                except ValueError:
                    pass  # any non-floats in this line are ignored

            # Save positional coordatinates of end particles
            if int(particle_values[2]) == 1:  # first particle in molecule
                rod_positions[int(particle_values[1]) - 1, 0, :] = particle_values[3:6]
            if int(particle_values[2]) == mol_length:  # last particle in molecule
                rod_positions[int(particle_values[1]) - 1, 1, :] = particle_values[3:6]
```

```
    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume
    order_param_values[i] = order_param(rod_positions)  # evaluate order param at time t
    print("T = " + str(time) + "/" + str(run_time))

fig, ax1 = plt.subplots()

color = "tab:red"
ax1.set_xlabel("Time (arbitrary units)")
ax1.set_ylabel("Order Parameter", color=color)
ax1.plot(time_range, uniform_filter1d(order_param_values, size=int(10)), color=color)
ax1.tick_params(axis="y", labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = "tab:blue"
ax2.set_ylabel(
    "Volume Fraction", color=color
)  # we already handled the x-label with ax1
ax2.plot(time_range, vol_frac(volume_values, mol_length, N_molecules), color=color)
ax2.tick_params(axis="y", labelcolor=color)

plt.title("Evolution of Order Parameter")
fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.savefig("order_and_volfrac.png")
plt.show()
```

4.3 *smectic_plot.py*

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.ndimage import uniform_filter1d  # for rolling average
from phase_plot import vol_frac

file_root = "output_T_0.5_time_"
sampling_freq = 5  # only samples one in X files (must be integer)
plotting_freq = 5  # only plots on in X of the sampled distributions

plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth


def order_param(data):
    """Input data in array of length Molecule Number

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Third index gives the component of the position of CoM (x,y,z)

    Method for calculation given by Polson (1997) 10.1103/PhysRevE.56.R6260
    """
    k = 2 * np.pi / 10  # divide by length of molecule
    exp_values = np.exp(1j * k * data)
    return np.abs(np.sum(exp_values)) / N_molecules
```

```
def CoM_dist(data):
    """Input data in array of size Molecule Number x 3

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Second index gives particle number within molecule (1/5/6/10)
    Third index gives the component of the position (x,y,z)

    """
    return data[:, 1]  # return y coordinate


# ... Same initial extraction as nematic order plot ...


# READ MOLECULE POSITIONS

volume_values = np.full(len(time_range), np.nan)  # new array of NaN
order_param_values = np.full(len(time_range), np.nan)
CoM_mean_values = []
CoM_mean_times = []
for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(file_root + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    # ... same extraction as nematic order plot...

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume
    order_param_values[i] = order_param(rod_positions[:, 1])

    CoM_data = rod_positions[:, 1]  # select y coordinate
    # kde_data = scipy.stats.gaussian_kde(CoM_data)

    plot_num = 0
    tot_plot_num = len(time_range) // plotting_freq
    colors = plt.cm.cividis(np.linspace(0, 1, tot_plot_num))
    if i % plotting_freq == 0 and time != 0:
        if i == plotting_freq or time >= run_time - (
            dump_interval * sampling_freq * plotting_freq
        ):

            my_kde = sns.kdeplot(
                CoM_data,
                label="T = " + str(int(time)),  # start and end times
                color=colors[i // plotting_freq - 1],
                bw_adjust=0.5,  # adjusts smoothing (default is 1)
                cut=0,
                # clip=(0, 50), # cuts off x limit
                # gridsize=50,  #adjusts points in average (default is 200)
            )
        else:
            my_kde = sns.kdeplot(
                CoM_data,
                color=colors[i // plotting_freq - 1],
                alpha=1,
                bw_adjust=0.5,
```

```
                cut=0,
                # clip=(0, 50), # cuts off x limit
                # gridsize=50
            )
            # alpha may be used to adjust transparency
        line = my_kde.lines[plot_num - 1]
        x, y = line.get_data()
        CoM_mean_values.append(max(y) - min(y))
        CoM_mean_times.append(time)  # evaluate order param at time t
        plot_num += 1
        print(max(y), min(y))

    print("T = " + str(time) + "/" + str(run_time))

plt.title("Evolution of CoM distribution over time")
plt.xlabel("Mean CoM")
plt.ylabel("Normalised Frequency")
plt.legend()
```

5 Further Static Analysis

In this section I outline the additional analysis methods used further to the traditional order parameters presented in Section 4. This methods were primarily introduced to account for the additional degrees of freedom when considering the anisotropic nunchuck molecules. Initially we consider the distribution of opening angles in a sample of nunchuck molecules with fixed rigidity, and this distribution is visualised in the '*angle_plot.py*' script.

I present code for the pair-wise orientational correlation function, using the spherical harmonic calculation approach introduced by Daan Frenkel. Further correlation functions, using a more traditional 'brute-force' approach, are also applied to the bisector and normal vectors for the anisotropic molecules in '*correlation_plot_alternative.py*'.

5.1 *angle_plot.py*

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
from phase_plot import vol_frac

file_root = "output_T_0.5_time_"
sampling_freq = 40  # only samples one in X files (must be integer) #30
plotting_freq = 1  # only plots on in X of the sampled distributions

plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth

# ... Same data extraction procedure as phase_plot.py ...


def angle_dist(data, remove_split_mol=True):
    """Input data in array of size Molecule Number x 3 x 3

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Second index gives particle number within molecule (1st/mid/last)
    Third index gives the component of the position (x,y,z)

    Allows option to remove molecules that are split across boundaries
```

```
    """

    if remove_split_mol:
        molecules_removed = 0
        # where molecules span simulation region boundaries, replace with nans
        for i in range(N_molecules):
            if np.linalg.norm(data[i, 2, :] - data[i, 0, :]) > (mol_length + 0.5):
                data[i, :, :].fill(np.nan)
                molecules_removed += 1
                # remove data for molecules that are longer than expected
                # this is due to them spanning the edges of the simulation region
        print("Number of molecules removed is : " + str(molecules_removed))

    rod_1 = data[:, 1, :] - data[:, 0, :]  # director vector for first arm
    norm_rod_1 = rod_1 / np.linalg.norm(rod_1, axis=1).reshape(-1, 1)
    rod_2 = data[:, 1, :] - data[:, 2, :]  # director vector for second arm
    # note this is defined so the two vectors point away from the centre
    norm_rod_2 = rod_2 / np.linalg.norm(rod_2, axis=1).reshape(-1, 1)

    angle_values = np.sum(norm_rod_1 * norm_rod_2, axis=1)
    angle_values = angle_values[~np.isnan(angle_values)]  # remove nans
    return angle_values


# READ MOLECULE POSITIONS

angle_mean_values = np.zeros(len(time_range))
volume_values = np.full(len(time_range), np.nan)  # new array of NaN
for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(file_root + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    # ... Same positional extraction as correlation_plot.py ...

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume

    angle_data = angle_dist(rod_positions)
    angle_mean_values[i] = np.mean(angle_data)  # evaluate order param at time t

    angle_data = np.where(
        angle_data < 0.8, angle_data, np.nan
    )  # remove spurious high values

    tot_plot_num = len(time_range) // plotting_freq
    colors = plt.cm.viridis(np.linspace(0, 1, tot_plot_num))
    if i % plotting_freq == 0 and time != 0:
        print(time)
        sns.kdeplot(
            angle_data,
            color=colors[i // plotting_freq - 1],
            bw_adjust=0.1,  # adjusts smoothing (default is 1)
            # gridsize=50,
            alpha=1,  # adjusts transparency
        )
```

```
    print("T = " + str(time) + "/" + str(run_time))

sm = plt.cm.ScalarMappable(cmap=cm.viridis, norm=plt.Normalize(vmin=0, vmax=run_time))
cbar = plt.colorbar(sm)
cbar.ax.set_ylabel("Number of Time Steps", rotation=270, labelpad=15)

plt.title("Evolution of angle distribution")
plt.xlim([-1, 1])
plt.xlabel(r"Nunchuck Angle ($cos(\theta)$)")
plt.ylabel("Normalised Frequency")
plt.savefig("nun_fr_angledist.eps")
plt.show()
```

5.2 *correlation_plot.py*

```
"""Optimised method to calculate the pair-wise orientational order correlation function.
As detailed by Daan Frenkel, uses spherical harmonics so avoid costly angle
calculations (which are O(N^2)). Uses the end-to-end molecule vector as the director"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
from scipy.ndimage import uniform_filter1d  # for rolling average
from scipy.special import sph_harm
from phase_plot import vol_frac

FILE_ROOT = "output_T_0.5_time_"  # two underscores to match typo in previous code
SAMPLING_FREQ = 20  # only samples one in X files (must be integer)
SEPARATION_BIN_NUM = 20  # number of bins for radius dependance pair-wise correlation

# mol_length = 10  # define explicitly on older datasets

plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth

# ... Same data extraction procedure as phase_plot.py ...


def find_angles(vec_array):
    """Finds spherical angles of cartesian pos. Returns theta and phi in N x 2 array

    vec_array should be of size N x 3, for N angles (ie N particles)
    Choice of (normalised) base vector is arbitrary in order param formulation"""

    assert vec_array.shape[1] == 3, "Vectors should be three dimensional"
    radius = np.linalg.norm(vec_array, axis=1)
    theta = np.arccos(vec_array[:, 2] / radius)
    phi = np.arctan2(vec_array[:, 1], vec_array[:, 0])
    return theta, phi


def find_separations(pos_array, base_pos, box_dim):
    """Finds separation of positions in array from a base position

    This method finds the minimum separation, accounting for the periodic BC
    pos1, pos2 are the position vectors of the two points
    box_dim is a vector (of equal length) giving the dim of the simulation region"""
```

```
    separation_data = pos_array - base_pos
    for n in range(pos_array.shape[0]):
        for i in range(3):  # for 3 dimensional vector
            if np.abs(pos_array[n, i] - base_pos[i]) > box_dim[i] / 2:
                # use distance to ghost instead
                separation_data[n, i] = box_dim[i] - np.abs(
                    pos_array[n, i] - base_pos[i]
                )

    return np.linalg.norm(separation_data, axis=1)


def spherical_harmonic_sum(theta_array, phi_array, l, m):
    """ Returns sum  of spherical harmonics of order l,m"""
    angles_sum = np.sum(sph_harm(m, l, theta_array, phi_array).real)
    return angles_sum


def correlation_func(data, box_dim, bin_num, order):
    """Input data in array of size Molecule Number x 3 x 3

    Input data will be rod_positions array which stores input data
    First index gives molecule number (up to size N_molecules)
    Second index gives particle number within molecule (first/last)
    Third index gives the component of the position (x,y,z)

    Also input  'order' - order of correlation function to compute
                'box_dim' - list of simulation box dimensions
                'bin_num' - integer value for number of radius bins to evaluate

    Returns array of correlation data at each radius"""

    directors = data[:, 2, :] - data[:, 0, :]
    theta_array, phi_array = find_angles(directors)

    max_separation = np.linalg.norm(box_dim) / 2

    bin_width = max_separation / bin_num
    separation_bins = np.linspace(0, max_separation, bin_num, endpoint=False)
    correlation_data = np.zeros_like(separation_bins)

    for n, radius in enumerate(separation_bins):
        order_param_sum = 0
        sample_size = 0

        for i in range(N_molecules):
            running_tot = 0

            separation_array = find_separations(data[:, 2, :], data[i, 2, :], box_dim)
            relevant_theta = np.ma.masked_where(
                np.logical_or(
                    (separation_array < radius),
                    (separation_array > (radius + bin_width)),
                ),
                theta_array,
            )
            relevant_phi = np.ma.masked_where(
```

```
                np.ma.getmask(relevant_theta), phi_array
            )  # applies the mask of theta on phi

            for m in range(-order, order + 1):  # from -l to l
                harmonic_at_i = sph_harm(m, order, theta_array[i], phi_array[i]).real
                harmonics_sum = spherical_harmonic_sum(
                    relevant_theta, relevant_phi, order, m
                )
                # remove i=j term from sum, then multiply by harmonic for molecule i
                running_tot += (harmonics_sum - harmonic_at_i) * harmonic_at_i

            order_param_sum += (4 * np.pi / (2 * order + 1)) * running_tot
            sample_size += relevant_theta.count()  # number of pairs sampled

        correlation_data[n] = order_param_sum / sample_size

        print("    radius = " + str(int(radius)) + "/" + str(int(max_separation)))
    print(correlation_data)
    return separation_bins, correlation_data


# READ MOLECULE POSITIONS

order_param_values = np.zeros(len(time_range))
volume_values = np.full(len(time_range), np.nan)  # new array of NaN
for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(FILE_ROOT + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    box_volume = 1
    box_dimensions = []  # to store side lengths of box for period boundary adjustment
    rod_positions = np.zeros((N_molecules, 3, 3))
    """Indices are Molecule Number; Atom number 1st/mid/last ; Pos coord index"""

    for line in data_file:
        if "ITEM: BOX" in line:  # to start reading volume data
            extract_box_data = True
            extract_atom_data = False
            continue  # don't attempt to read this line

        if "ITEM: ATOMS" in line:  # to start reading particle data
            extract_box_data = False
            extract_atom_data = True
            continue

        if extract_box_data and not extract_atom_data:
            # evaluate before particle values
            # each line gives box max and min in a single axis
            box_limits = []
            for d in line.split():  # separate by whitespace
                try:
                    box_limits.append(float(d))
                except ValueError:
                    pass  # any non-floats in this line are ignored
            box_volume *= box_limits[1] - box_limits[0]
            # multiply box volume by length of this dimension of box
            box_dimensions.append(box_limits[1] - box_limits[0])
```

```
        if extract_atom_data and not extract_box_data:
            # evaluate after box dimension collection
            # each line is in the form "id mol type x y z vx vy vz"
            particle_values = []
            for t in line.split():  # separate by whitespace
                try:
                    particle_values.append(float(t))
                except ValueError:
                    pass  # any non-floats in this line are ignored

            # # Save positional coordatinates of end particles - REGULAR
            # if int(particle_values[2]) == 1:  # first particle
            #     rod_positions[int(particle_values[1]) - 1, 0, :] = particle_values[3:6]
            # if int(particle_values[2]) == int((mol_length + 1) / 2):  # centre particle
            #     rod_positions[int(particle_values[1]) - 1, 1, :] = particle_values[3:6]
            # if int(particle_values[2]) == mol_length:  # last particle
            #     rod_positions[int(particle_values[1]) - 1, 2, :] = particle_values[3:6]

            # Save positional coordatinates of end particles - CLOSE
            centre = (mol_length + 1) / 2
            if int(particle_values[2]) == int(centre - 1):
                rod_positions[int(particle_values[1]) - 1, 0, :] = particle_values[3:6]
            if int(particle_values[2]) == int(centre):  # central particle
                rod_positions[int(particle_values[1]) - 1, 1, :] = particle_values[3:6]
            if int(particle_values[2]) == int(centre + 1):
                rod_positions[int(particle_values[1]) - 1, 2, :] = particle_values[3:6]

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume
    separation_bins, correlation_data = correlation_func(
        rod_positions, box_dimensions, SEPARATION_BIN_NUM, order=2
    )  # evaluate order param at time t

    tot_plot_num = len(time_range)
    colors = plt.cm.cividis(np.linspace(0, 1, tot_plot_num))
    if i == 0:
        continue  # don't plot this case
    plt.plot(
        separation_bins, correlation_data, color=colors[i],
    )

    print("T = " + str(time) + "/" + str(run_time))

sm = plt.cm.ScalarMappable(cmap=cm.cividis, norm=plt.Normalize(vmin=0, vmax=run_time))
cbar = plt.colorbar(sm)
cbar.ax.set_ylabel("Number of Time Steps", rotation=270, labelpad=15)

plt.title("Pairwise Angular Correlation Function")
plt.xlabel("Particle Separation")
plt.ylabel("Correlation Function")
plt.savefig("correlation_func_fast.png")
plt.show()
```

5.3 *correlation_plot_alternative.py*

```
"""Additional approaches to calcuating the pair-wise orientational order
```

```
correlation function, using different director vectors"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
from scipy.ndimage import uniform_filter1d  # for rolling average
from phase_plot import vol_frac

FILE_ROOT = "output_T_0.5_time_"  # two underscores to match typo in previous code
SAMPLING_FREQ = 20  # only samples one in X files (must be integer)
SEPARATION_BIN_NUM = 30  # number of bins for radius dependance pair-wise correlation

mol_length = 15

DIRECTOR_METHOD = "molecule"
# Options are molecule/arm/bisector/normal


plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth

# ... Same data extraction procedure as phase_plot.py ...


def find_director(data, method="molecule"):
    """Obtains director from molecule positions, through a variety of methods

    First index gives molecule number
    Second index gives particle number within molecule
        Corresponds to: start of director/ centre/ end of director
    Third index gives the component of the position (x,y,z)

    'molecule' calculates director between ends of the molecule;
    'arm' calculates director along first arm of molecule;
    'bisector' gives the director along the bisector of the join angle;
    'normal' gives the bisector out of the plane of the molecule
    """

    if method == "molecule":
        return data[:, 2, :] - data[:, 0, :]

    elif method == "arm":
        return data[:, 1, :] - data[:, 0, :]

    elif method == "bisector":
        midpoints = 0.5 * (data[:, 2, :] + data[:, 0, :])
        return data[:, 1, :] - midpoints

    elif method == "normal":
        arm1 = data[:, 1, :] - data[:, 0, :]
        arm2 = data[:, 2, :] - data[:, 1, :]
        return np.cross(arm1, arm2)

    else:
        raise ValueError("Unknown argument to find_director()")


def find_angle(vec1, vec2):
```

```
    """Finds angle between two vectors"""
    assert len(vec1) == len(vec2), "Vectors should be the same dimension"

    vec1 = vec1 / np.linalg.norm(vec1)  # normalise vectors
    vec2 = vec2 / np.linalg.norm(vec2)

    return np.sum(vec1 * vec2)


def find_separation(pos1, pos2, box_dim):
    """Finds separation between two positions

    This method finds the minimum separation, accounting for the periodic BC
    pos1, pos2 are the position vectors of the two points
    box_dim is a vector (of equal length) giving the dim of the simulation region"""
    separation = pos1 - pos2
    for i in range(len(pos1)):  # should be 3 dimensional
        if np.abs(pos1[i] - pos2[i]) > box_dim[i] / 2:
            # use distance to ghost instead
            separation[i] = box_dim[i] - np.abs(pos1[i] - pos2[i])

    return np.linalg.norm(separation)


def eval_angle_array(data, box_dim):
    """Input data in array of size Molecule Number x 3 x 3, and list of box_dim

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Second index gives particle number within molecule
        Corresponds to: start of director/ centre/ end of director
    Third index gives the component of the position (x,y,z)

    Outputs N x N x 2 array, for pairwise values of separation and angle
    Be aware this may generate very large arrays
    """
    angle_array = np.full((N_molecules, N_molecules, 2), np.nan, dtype=np.float32)
    # dtype specified to reduce storgae required

    director = find_director(
        data, method=DIRECTOR_METHOD
    )  # director vector for whole of molecule

    for i in range(N_molecules - 1):
        for j in range(i + 1, N_molecules):
            # Only considering terms of symmetric matrix above diagonal
            # Separation between centres of each molecule:
            angle_array[i, j, 0] = find_separation(
                data[i, 1, :], data[j, 1, :], box_dim
            )
            # Angle between arms of molecule:
            angle_array[i, j, 1] = find_angle(director[i, :], director[j, :])
    angle_array_masked = np.ma.masked_invalid(
        angle_array[:, :, :]
    )  # mask empty values below diagonal
    return angle_array_masked
```

```
def correlation_func(data, box_dim):
    """Input data in array of size Molecule Number x 3 x 3, and list of box_dim

    Input data will be rod_positions array which stores input data
    First index gives molecule number
    Second index gives particle number within molecule (first/last)
    Third index gives the component of the position (x,y,z)

    Returns array of correlation data at each radius"""

    angle_array = eval_angle_array(data, box_dim)
    max_separation = np.max(angle_array[:, :, 0])

    bin_width = max_separation / SEPARATION_BIN_NUM
    separation_bins = np.linspace(0, max_separation, SEPARATION_BIN_NUM, endpoint=False)
    correlation_data = np.zeros_like(separation_bins)

    for n, radius in enumerate(separation_bins):
        # mask data outside the relevant radius range
        relevant_angles = np.ma.masked_where(
            np.logical_or(
                (angle_array[:, :, 0] < radius),
                (angle_array[:, :, 0] > (radius + bin_width)),
            ),
            angle_array[:, :, 1],  # act on angle data
        )

        legendre_polynomials = np.polynomial.legendre.legval(
            relevant_angles[:, :], [0, 0, 1]
        )  # evaluate 2nd order legendre polynomial

        correlation_data[n] = np.mean(legendre_polynomials)

        print("    radius = " + str(int(radius)) + "/" + str(int(max_separation)))

    return separation_bins, correlation_data


# READ MOLECULE POSITIONS

order_param_values = np.zeros(len(time_range))
volume_values = np.full(len(time_range), np.nan)  # new array of NaN
for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(FILE_ROOT + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    box_volume = 1
    box_dimensions = []  # to store side lengths of box for period boundary adjustment
    rod_positions = np.zeros((N_molecules, 3, 3))
    """Indices are Molecule Number; Atom number 1st/mid/last ; Positional coord index"""

    # ... Same position extraction procedure as correlation_plot.py

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume
    separation_bins, correlation_data = correlation_func(
        rod_positions, box_dimensions
```

```
    )  # evaluate order param at time t

    tot_plot_num = len(time_range)
    colors = plt.cm.cividis(np.linspace(0, 1, tot_plot_num))
    if i == 0:
        continue  # don't plot this case
    plt.plot(
        separation_bins, correlation_data, color=colors[i],
    )

    print("T = " + str(time) + "/" + str(run_time))

sm = plt.cm.ScalarMappable(cmap=cm.cividis, norm=plt.Normalize(vmin=0, vmax=run_time))
cbar = plt.colorbar(sm)
cbar.ax.set_ylabel("Number of Time Steps", rotation=270, labelpad=15)

plt.title("Pairwise Angular Correlation Function")
plt.xlabel("Particle Separation")
plt.ylabel("Correlation Function")
image_name = "correlation_func_test" + str(DIRECTOR_METHOD) + ".png"
plt.savefig(image_name)
plt.show()
```

6 Dynamic Analysis

Finally, our analysis is extended to the dynamic behaviour of the system. The total root mean squared (rms) displacement is plotted in '*diffusion_plot.py*', as well as a predictor of the diffusion coefficients. This is then extended in '*diffusion_coeff.py*', where plots of coordinate displacement in logarithmic space allow for phase identification. Directional diffusion coefficients are calculated, as well as the degree of sub-diffusion quantified.

6.1 *diffusion_plot.py*

```
"""Calculates the rms displacement over the simulation, and diffusion
coefficient over different timescales, Plots these over time, including
a directional approach (treating each coordinate separately)."""

import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter1d  # for rolling average
from phase_plot import vol_frac

FILE_ROOT = "output_T_0.5_time_"  # two underscores to match typo in previous code
SAMPLING_FREQ = 1  # only samples one in X files (must be integer)

DIRECTIONAL_COEFF = True

# mol_length = 10  #uncomment on older datasets

plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth

# ... Same data extraction procedure asphase_plot.py ...

sampling_times = np.zeros((len(mix_steps_values), 2))
# Gives start and end times for each equillibrium run
time_counter = 0
```

```
for i in range(len(mix_steps_values)):
    time_counter += mix_steps_values[i]
    sampling_times[i, 0] = time_counter
    time_counter += equilibrium_time
    sampling_times[i, 1] = time_counter

assert time_counter == run_time, "Unexpected result in sampling times"


# CALCULATE THE RMS DISPLACEMENT


def rms_displacement(pos_t, pos_0, box_dim, use_vector=False):
    """Input data in array of size Molecule Number x 3, and list of box_dim

    Input data will be com_positions array which stores input data
    First index gives molecule number
    Second index gives the component of the position (x,y,z)

    If use_vector is false, returns rms displacement from initial displacement
    If use_vector is true, returns average displacement in each coordinate axis"""
    if use_vector:
        rms_vector = np.abs((pos_t - pos_0))
        return np.mean(rms_vector, axis=0)
    else:
        rms_value = np.linalg.norm((pos_t - pos_0))
        return np.mean(rms_value)


# READ MOLECULE POSITIONS

if DIRECTIONAL_COEFF:
    dimension_num = 3
    axis_labels = ["x", "y", "z"]
else:
    dimension_num = 1
    axis_labels = ["RMS"]

displacement_values = np.zeros((len(time_range), dimension_num))
volume_values = np.full(len(time_range), np.nan)  # new array of NaN

# for sampled measurements:
sampled_D_values = np.full((len(mix_steps_values), dimension_num), np.nan)
sampled_vol_values = np.full(len(mix_steps_values), np.nan)

for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(FILE_ROOT + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    box_volume = 1
    box_dimensions = []  # to store side lengths of box for period boundary adjustment
    com_positions = np.zeros((N_molecules, 3))
    """Indices are Molecule Number;  Positional coord index"""

    for line in data_file:
        if "ITEM: BOX" in line:  # to start reading volume data
            extract_box_data = True
```

```
            extract_atom_data = False
            continue  # don't attempt to read this line

        if "ITEM: ATOMS" in line:  # to start reading particle data
            extract_box_data = False
            extract_atom_data = True
            continue

        if extract_box_data and not extract_atom_data:
            # evaluate before particle values
            # each line gives box max and min in a single axis
            box_limits = []
            for d in line.split():  # separate by whitespace
                try:
                    box_limits.append(float(d))
                except ValueError:
                    pass  # any non-floats in this line are ignored
            box_volume *= box_limits[1] - box_limits[0]
            # multiply box volume by length of this dimension of box
            box_dimensions.append(box_limits[1] - box_limits[0])

        if extract_atom_data and not extract_box_data:
            # evaluate after box dimension collection
            # each line is in the form "id mol type x y z vx vy vz"
            particle_values = []
            for t in line.split():  # separate by whitespace
                try:
                    particle_values.append(float(t))
                except ValueError:
                    pass  # any non-floats in this line are ignored

            # Save positional coordatinates of end particles - CLOSE
            centre = (mol_length + 1) / 2
            if int(particle_values[2]) == int(centre):  # central particle
                com_positions[int(particle_values[1]) - 1, :] = particle_values[3:6]

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume

    if time == 0:
        initial_positions = com_positions
        displacement_values[0, :] = np.nan
    else:
        displacement_values[i, :] = rms_displacement(
            com_positions,
            initial_positions,
            box_dimensions,
            use_vector=DIRECTIONAL_COEFF,
        )  # evaluate <x^2> at time t

    # For specific sample measurement

    if time in sampling_times:
        where_output = np.where(sampling_times == time)
        indices = (where_output[0][0], where_output[1][0])
        if indices[1] == 0:  # start of sampling period
            initial_sample = com_positions
            sampled_vol_values[indices[0]] = box_volume
```

```
        else:  # end of sampling period
            sampled_rms = rms_displacement(
                com_positions,
                initial_sample,
                box_dimensions,
                use_vector=DIRECTIONAL_COEFF,
            )  # initial sample taken from previous iteration in if clause
            sampled_D_values[indices[0], :] = sampled_rms / (6 * equilibrium_time)
            # D value for i-th equillibration period

    print("T = " + str(time) + "/" + str(run_time))

time_range[0] = 1  # avoid divide by zero error, will be ignored anyway
diffusion_coeff_values = (1 / 6) * np.divide(displacement_values.T, time_range).T

print("Mean Diffussion Coefficients: " + str(np.nanmean(diffusion_coeff_values)))

for i in range(dimension_num):
    plt.plot(time_range, displacement_values[:, i], label=axis_labels[i])
plt.xlabel("Time (arbitrary units)")
plt.ylabel("RMS displacement")
plt.legend()
plt.savefig("rms_displacement.png")
plt.show()

for i in range(dimension_num):
    plt.plot(
        vol_frac(sampled_vol_values, mol_length, N_molecules),
        sampled_D_values[:, i],
        "x",
        label=axis_labels[i],
    )
plt.ylabel("Diffusion Coefficient")
plt.xlabel("Volume Fraction")
plt.legend()
plt.savefig("order_vs_diffusion_sampled.png")
plt.show()
```

6.2 *diffusion_coeff.py*

```
"""Calculates the diffusion coefficient over each equillibration run.
Accounts for additional displacement when crossing the periodic boundary conditions"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter1d  # for rolling average
from scipy.stats import linregress  # for linear regression
from phase_plot_future import vol_frac

FILE_ROOT = "output_T_0.5_time_"  # two underscores to match typo in previous code
DIRECTIONAL_COEFF = True

mol_length = 10  # uncomment on older datasets

plt.rcParams.update({"font.size": 13})  # for figures to go into latex at halfwidth

# ... Same data extraction procedure as phase_plot.py ...
```

```
# GENERATE LIST OF TIME STEPS TO SAMPLE

sampling_times = np.zeros((len(mix_steps_values), 2))
# Gives start and end times for each equillibrium run
time_counter = 0
for i in range(len(mix_steps_values)):
    time_counter += mix_steps_values[i]
    sampling_times[i, 0] = time_counter
    time_counter += equilibrium_time
    sampling_times[i, 1] = time_counter

assert time_counter == run_time, "Unexpected result in sampling times"

# CALCULATE THE RMS DISPLACEMENT
def periodic_bc_displacement(current_pos, previous_pos, box_dim, tolerance=0.5):
    """Input position data in arrays of size Molecule Number x 3, and list of box_dim

    Input data will be com_positions array which stores input data
    First index gives molecule number
    Second index gives the component of the position (x,y,z)
    'tolerance' is the maximum allowed step size (without assuming boundary crossing)
        It is measured as a fraction of the box dimensions, default is 0.5

    Returns an additional displacement vector which track disp across boundaries"""
    delta_x = current_pos - previous_pos
    output_displacement = np.zeros_like(delta_x)
    for i in range(len(box_dim)):
        # adds displacement equal to box dimension in direction of boundary crossing
        output_displacement[:, i] += np.where(
            delta_x[:, i] > tolerance * box_dim[i], -box_dim[i], 0
        )  # crossings in negative axis direction (to give large +ve delta_x)
        output_displacement[:, i] += np.where(
            delta_x[:, i] < -1 * tolerance * box_dim[i], box_dim[i], 0
        )  # crossings in positive axis direction
    return output_displacement


def rms_displacement(pos_t, pos_0, use_vector=False):
    """Input data in array of size Molecule Number x 3, and list of box_dim

    Input data will be com_positions array which stores input data
    First index gives molecule number
    Second index gives the component of the position (x,y,z)

    If use_vector is false (default), returns rms displacement from initial disp
    If use_vector is true, returns average displacement in each coordinate axis"""
    if use_vector:
        rms_vector = np.abs((pos_t - pos_0))
        return np.mean(rms_vector, axis=0)
    else:
        rms_value = np.linalg.norm((pos_t - pos_0))
        return np.mean(rms_value)


# READ MOLECULE POSITIONS

if DIRECTIONAL_COEFF:
```

```
    dimension_num = 3
    axis_labels = ["x", "y", "z"]
else:
    dimension_num = 1
    axis_labels = ["RMS"]

volume_values = np.full(len(time_range), np.nan)  # new array of NaN

# for ongoing measurements:
rms_disp_values = np.full((run_num_tot, len(eq_range), dimension_num), np.nan)
time_origin = 0
run_num = 0

# for sampled measurements:
sampled_D_values = np.full((len(mix_steps_values), dimension_num), np.nan)
sampled_vol_values = np.full(len(mix_steps_values), np.nan)
equilibrium_flag = False  # denotes whether system is currently in an equillibrium run

extra_displacement = np.zeros((N_molecules, 3))
# additional values to account for crossing the boundaries of the box

for i, time in enumerate(time_range):  # interate over dump files
    data_file = open(FILE_ROOT + str(time) + ".dump", "r")
    extract_atom_data = False  # start of file doesn't contain particle values
    extract_box_data = False  # start of file doesn't contain box dimension

    # ... Same positional data extraction as in diffusion_plots.py ...

    data_file.close()  # close data_file for time step t
    volume_values[i] = box_volume

    if time != 0:  # GIVE PREVIOUS DISPLACEMENT
        extra_displacement += periodic_bc_displacement(
            com_positions, previous_positions, box_dimensions
        )

    if equilibrium_flag:  # MEASURE ONGOING RMS DISPLACEMENT
        rms_disp_values[run_num, i - run_origin, :] = rms_displacement(
            com_positions + extra_displacement,  # takes current values
            initial_sample,  # reset for each eq run
            use_vector=DIRECTIONAL_COEFF,
        )

    if time in sampling_times:  # MEASURE SAMPLING POINTS
        print("T = " + str(time) + "/" + str(run_time))
        where_output = np.where(sampling_times == time)
        indices = (where_output[0][0], where_output[1][0])
        if indices[1] == 0:  # start of sampling period
            initial_sample = com_positions
            sampled_vol_values[indices[0]] = box_volume

            extra_displacement = np.zeros_like(com_positions)  # reset for each sample

            run_origin = i  # so ongoing measurement starts from zero each time
            equilibrium_flag = True
        else:  # end of sampling period
            # print(extra_displacement) # check you aren't getting multiple crossings
            sampled_rms = rms_displacement(
```

```
                com_positions + extra_displacement,
                initial_sample,
                use_vector=DIRECTIONAL_COEFF,
            )  # initial sample taken from previous iteration in if clause
            sampled_D_values[indices[0], :] = sampled_rms / (6 * equilibrium_time)
            # D value for i-th equillibration period

            run_num += 1
            equilibrium_flag = False

    previous_positions = com_positions

plot_list = range(0, run_num_tot, 1)  # runs to plot

sampled_vol_frac = vol_frac(sampled_vol_values, mol_length, N_molecules)

fig, axs = plt.subplots(nrows=1, ncols=len(plot_list), sharey=True, figsize=(10, 5))
for plot_index, data_index in enumerate(plot_list):
    axs[plot_index].set_title(
        r"$\phi =$" + "{:.2f}".format(sampled_vol_frac[data_index])
    )
    # print(rms_disp_values[data_index, :, 0])
    rms_disp_values[data_index, 0, :] = rms_disp_values[data_index, 1, :]  # remove nan
    eq_time_values = np.array(eq_range)
    eq_time_values[0] = eq_time_values[1]  # remove zero so log can be evaluated

    slope_x, intercept_x, r_value_x, p_value_x, std_err_x = linregress(
        np.log10(eq_time_values), np.log10(rms_disp_values[data_index, :, 0])
    )  # consider x axis for purpose of this
    slope_y, intercept_y, r_value_y, p_value_y, std_err_y = linregress(
        np.log10(eq_time_values), np.log10(rms_disp_values[data_index, :, 1])
    )  # consider x axis for purpose of this

    plot_best_fit = True

    print(
        "X: For vol frac = "
        + "{:.4f}".format(sampled_vol_frac[data_index])
        + ", x_slope = "
        + "{:.4f}".format(slope_x)
        + ", y_slope = "
        + "{:.4f}".format(slope_y)
        + ", y_error = "
        + "{:.4f}".format(std_err_y)
        + ", intercept ratio = "
        + "{:.4f}".format(10 ** intercept_x / 10 ** intercept_y)
    )

    for j in range(dimension_num):
        if plot_index == 0:  # for legend
            axs[plot_index].loglog(
                eq_range, rms_disp_values[data_index, :, j], label=axis_labels[j]
            )
            if plot_best_fit == True and j == 2:  # only needs to be plotted once
                axs[plot_index].plot(
                    eq_time_values,
                    (eq_time_values ** slope_x) * (10 ** intercept_x),
                    label="Best fit (x)",
```

```
                    linestyle="dashed",
                )
                axs[plot_index].plot(
                    eq_time_values,
                    (eq_time_values ** slope_y) * (10 ** intercept_y),
                    label="Best fit (y)",
                    linestyle="dashed",
                )
        else:  # no legend entries
            axs[plot_index].loglog(eq_range, rms_disp_values[data_index, :, j])

            if plot_best_fit == True and j == 2:
                axs[plot_index].plot(
                    eq_time_values,
                    (eq_time_values ** slope_x) * (10 ** intercept_x),
                    linestyle="dashed",
                )
                axs[plot_index].plot(
                    eq_time_values,
                    (eq_time_values ** slope_y) * (10 ** intercept_y),
                    linestyle="dashed",
                )


axs[int(len(plot_list) / 2)].set_xlabel(
    "Time (Arbitrary Units)"
)  # use median of plot_list
axs[0].set_ylabel("RMS displacement")
fig.legend(loc="center right")
plt.savefig("rms_displacement_runwise2.png")
plt.show()

for i in range(dimension_num):
    plt.plot(
        sampled_vol_frac, sampled_D_values[:, i], "x", label=axis_labels[i],
    )
plt.ylabel("Diffusion Coefficient")
plt.xlabel("Volume Fraction")
plt.legend()
plt.savefig("order_vs_diffusion_with_bc.png")
plt.show()

print("Volume fraction = " + str(sampled_vol_frac))
print("D_x/D_y = " + str(sampled_D_values[:, 0] / sampled_D_values[:, 1]))
```