CUED - Engineering Tripos Part IIB 2024-2025 **Module Coursework**

Module	4G5	Title of report	Investigating the Elasticity of Rubber
Date submitted:	19/03/2025	Assessment for this module is X 100% / 25% coursework of which this assignment forms 50 %	

UNDERGRADUATE and POST GRADUATE STUDENTS				
Name:	Mac Walker	College:	Magdalene	X Undergraduate Post graduate

Feedback to the student **See also comments in the text**		Very good	**Good**	Needs improvmt
CONTENT	**Completeness, quantity of content:** Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly?			
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?			
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?			
	Comments:			
PRESENTATION	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			
	Comments:			

Marker: Date:

Investigating the Elasticity of Rubber

BGN: 2265V

March 19, 2025

Contents

1 Introduction

This project explores the elasticity of rubber, specifically investigating the molecular mechanisms underlying its stretching and restoring behavior. Rubber (polyisoprene) is a molecule known for its elasticity, but what exactly is going on under the hood? How and why does it stretch back? What occurs at the molecular level when it stretches? How can we understand what is happening?

This project addresses these questions using molecular dynamics (MD) simulations. Molecular Dynamics is a computational technique in which the time evolution of interacting atoms is simulated by numerically integrating their equations of motion [6, 4]. By employing specialised simulation software, we model the dynamics of polyisoprene chains and record key parameters such as atomic positions, velocities, forces, and energies over time [1, 4].

The objectives of this project are to (1) measure the restoring force as a function of displacement, (2) separate entropic from enthalpic contributions, and (3) apply statistical analysis methods to ensure the validity of results. Through constrained molecular dynamics and time-series analysis, we quantify how restoring force scales with extension and investigate the role temperature plays in the balance between entropy and internal energy.

2 Theory and Background

2.1 Molecular Dynamics and Entropic Elasticity

Molecular Dynamics (MD) numerically integrates Newton's equations of motion to simulate particle trajectories. This technique provides insight into molecular processes, allowing us to relate atomic-level interactions to macroscopic properties [6, 4]. Rubber elasticity is governed largely by changes in configurational entropy. Stretching polymer chains reduces their accessible microstates, decreasing entropy and creating a restoring force opposing deformation [2, 3].

Simulations typically employ thermostats (e.g., Langevin) to maintain constant temperature, ensuring realistic sampling of the polymer's configurational space according to the Boltzmann distribution [4]. This approach allows clear distinction between entropic contributions and small enthalpic adjustments during moderate strains.

2.2 Autocorrelation and Statistical Error Estimation

In MD, there are inherent correlations between successive measurements in molecular dynamics simulations. Hence, standard statistical error analysis must account for the non-independence of data points. This is done using the autocorrelation function, which

quantifies the correlation between values at different times along a trajectory [5]. The integrated autocorrelation time, τ_{int}, determines how quickly these correlations decay. Therefore, knowledge of τ_{int} allows one to understand the effective number of independent samples available for estimating properties.

Autocorrelation Function: The autocorrelation function $C(t)$ measures how correlated force measurements are over time:

$$C(t) = \frac{\langle F(0)F(t)\rangle - \langle F\rangle^2}{\langle F^2\rangle - \langle F\rangle^2}$$

where $F(t)$ represents the force at time t, and the brackets $\langle\cdot\rangle$ denote ensemble averages over time.

Integrated Autocorrelation Time: To quantify how long force values remain correlated, we compute the integrated autocorrelation time τ_{int} as:

$$\tau_{\text{int}} = 0.5 + \sum_{t=1}^{M} C(t)$$

where M is the summation cutoff, chosen such that the tail contributions of $C(t)$ are negligible.

Estimating the statistical uncertainty of averages requires accurately determining τ_{int}, as an underestimated correlation time would lead to artificially small error bars. This measure is thus critical for ensuring statistical robustness and enabling proper interpretation of the observed restoring forces and energies [5].

2.3 Ergodicity and Entropy

The concept of ergodicity underpins MD simulations. It asserts that a system, over long enough time periods, explores as accesible microstates. Hence, averages over sufficiently long trajectories correspond to ensemble averages across all accessible states. Ergodicity ensures the meaningful extraction of thermodynamic properties such as entropy from microscopic states [2, 3]. In the Methods section, we utilise this concept to perform analysis of the restoring force as a function of displacement.

The statistical thermodynamics perspective of entropy is a quantification of the number of accessible microstates. Temperature emerges naturally as the derivative of entropy with respect to energy [2]. This formulation bridges the gap between atomic-level details and macroscopic thermodynamics, which is critical for understanding polymer elasticity [2, 3].

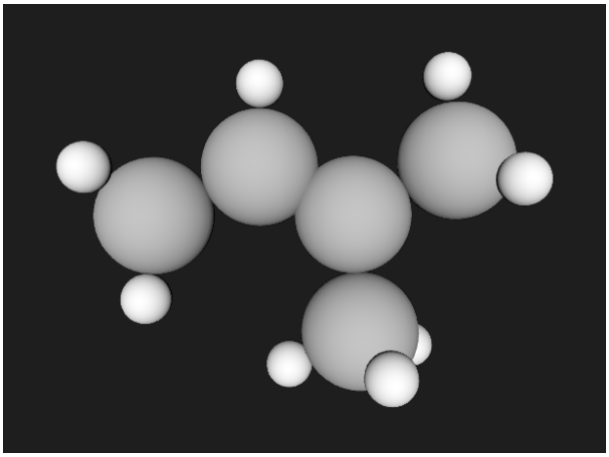

Figure 1: Isoprene Molecule, visualised using ase

3 Methods

3.1 Dynamics and Autocorrelation

We begin our investigation by using the software package `ase` for the molecular dynamics simulation of isoprene. In Figure 1, we see a visualisation of this molecule. In Figures 2 and 3 we display the distance between atom 8 and 13 over a simulation run of 500 fico seconds (fs).

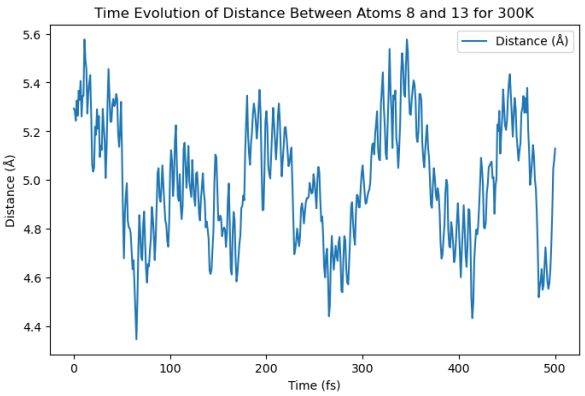

Figure 2: Time Evolution of Distance at 300K

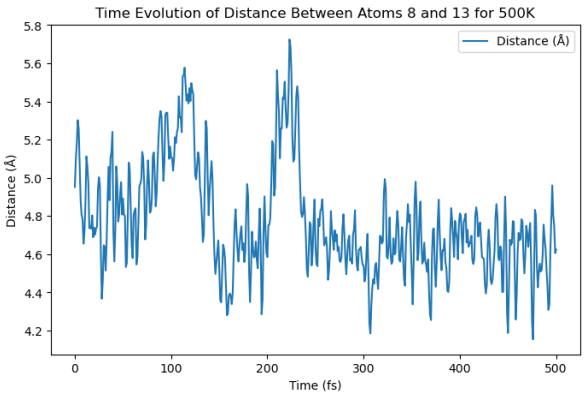

Figure 3: Time Evolution of Distance at 500K

To investigate the behaviour of the distance between atoms 8 and 13 in isoprene at different temperatures, we computed the autocorrelation function (ACF) of the distance time series at 300K and 500K (Figure 4).

To understand the underlying structural differences, we analysed the distribution of distances sampled throughout the trajectory (Figure 5). At 300K, the distances exhibit a broader distribution, suggesting transitions between multiple states. In contrast, at 500K, the distribution is more confined, indicating smoother fluctuations within a more

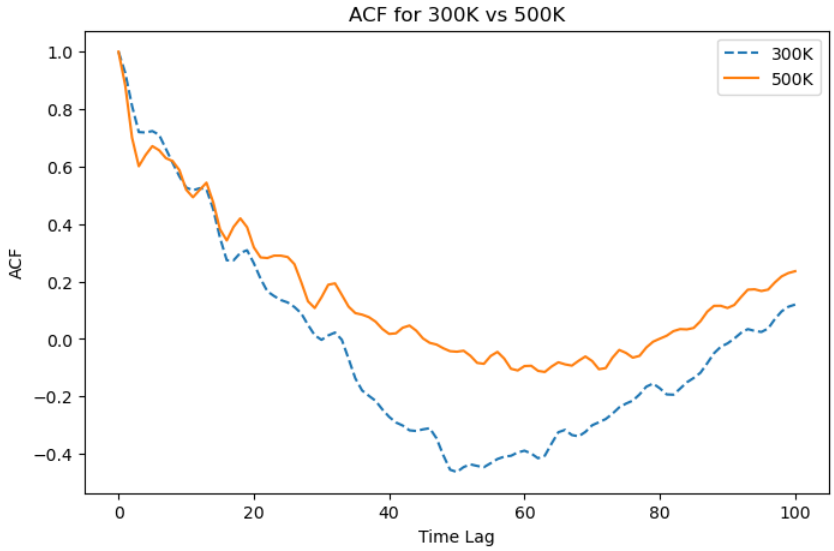

Figure 4: Autocorrelation function of the distance between atoms 8 and 13 at 300K and 500K. The 300K system loses correlation more rapidly than the 500K system.

restricted range of distances.

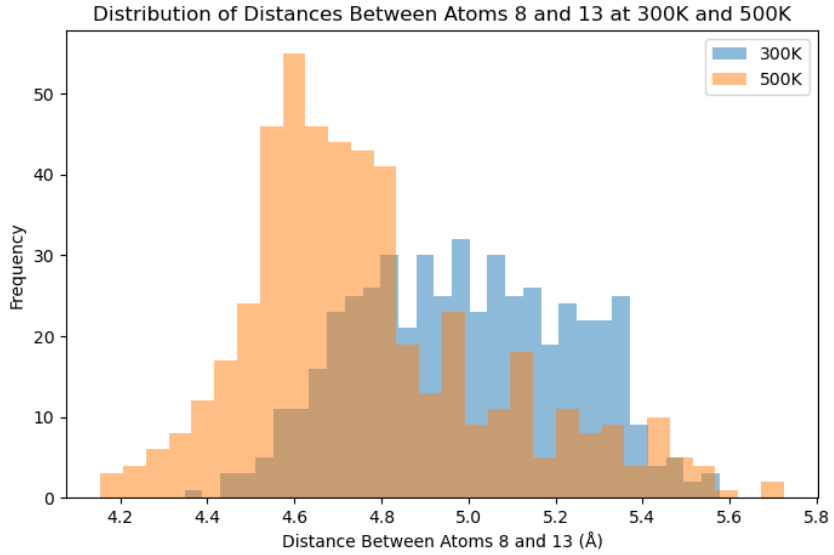

Figure 5: Histogram of distances between atoms 8 and 13 at 300K and 500K. The broader spread at 300K suggests discrete transitions, while the 500K system exhibits a more continuous distribution.

These results indicate that the reason the system at 300K decorrelates more rapidly is likely due to sharper transitions between distinct conformational states. Conversely, at 500K, the smoother molecular motion results in a longer correlation time. This suggests that higher temperature does not necessarily lead to faster decorrelation but rather a transition to a different dynamical regime.

4 Main Task

The objective of this study is to investigate the effect of displacement on the restoring force in a polymer system and to understand how temperature influences this effect. We use the `lammps` software package. We use the NVT emsemble scheme with a Langevin thermostat at 300K and 600K to maintain constant temperature - modelling a chain of polyisoprene. Further details of implementation may be found in the Appendix.

To achieve this, we conducted a total of 16 molecular dynamics (MD) simulations, systematically varying displacement and temperature. The simulations were performed at two temperatures (300K and 600K) with displacement values ranging from 130 to 200 $\mathring{A}$ in increments of 10 $\mathring{A}$.

4.1 Force vs. Displacement Relationship

In Figure 6, we measure the restoring force as a function of displacement. Let us first analyse the plot encompassing mean force at 300K. An obeservation is that clearly two relationships exist. For values between 130$\mathring{A}$ and 170$\mathring{A}$, the force remains relatively low and increases linearly with displacement. This is due to the polymer exhbiting entropic elasticity. For larger displacements $L \geq 170$ $\mathring{A}$, we see a sharper rise in the relationship between displacement and mean restoring force, most likely due to enthalpic contributions. Hence, there exists a shift from an entropic regime to an enthalpic regime at $L \approx 170$ $\mathring{A}$.

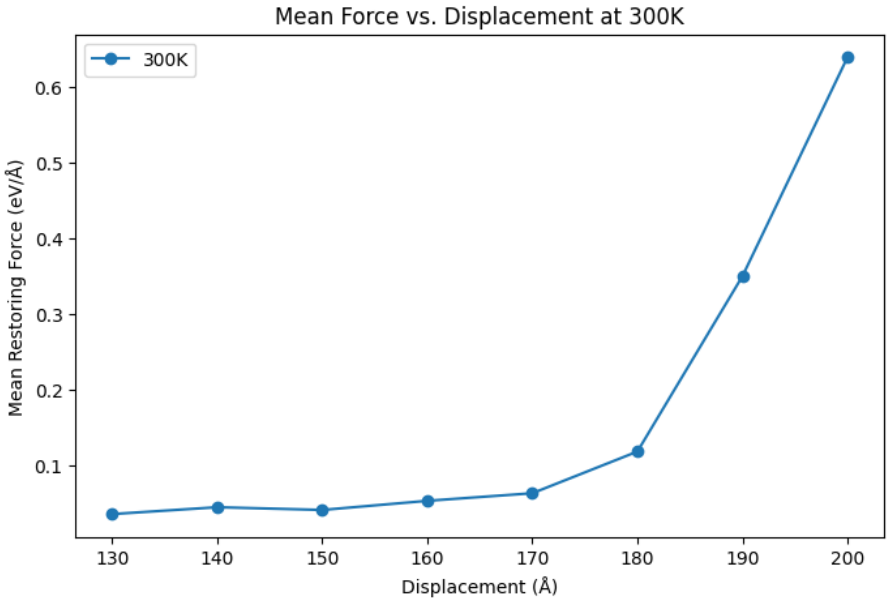

Figure 6: The average restoring force measured against displacement for 300K

4.2 Temperature Dependence of Force

In Figure 7, we see plot the temperature dependence of restorative force.

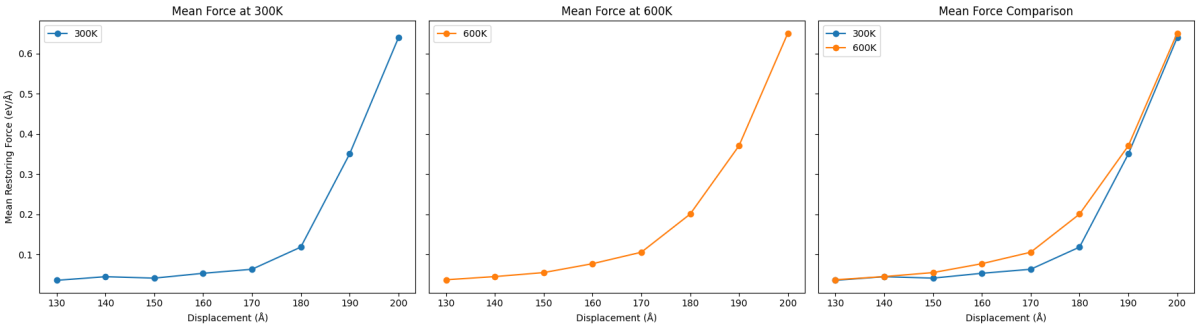

Figure 7: The average restoring force measured against displacement for both 300K and 600K

Analysing Figure 7, there are two key observations. First, at lower displacements ($130\mathring{A} \leq L \leq 170\mathring{A}$), both temperature curves exhibit a low-force, approximately linear increase with displacement. This suggests that in this regime, entropic elasticity dominates, and the polymer retains configurational freedom.

At $L = 170\ \mathring{A}$, we observe a sharper increase in restoring force, likely due to bond stretching.

Importantly, the force at 600K is consistently higher than at 300K across all displacements. This is expected from entropic elasticity theory, which predicts that restoring force scales with temperature [7]. The higher thermal energy at 600K reduces the number of available configurations at a given extension, increasing the resistance to deformation.

4.3 Error Bar Analysis

Before we complete the error bar analysis, we should update our theoretical framework established in the earlier section to deal with autocorrelation on a discretised, molecular dynamics simulation. Because force measurements in MD are correlated, the number of effective independent samples is smaller than the total number of data points. This necessitates using the integrated autocorrelation time, τ_{int}, to compute the true standard error.

Physical Autocorrelation Time: Since force measurements are collected every `tsamp` timesteps, the physical autocorrelation time, which determines how frequently truly independent samples occur, is:

$$\tau_{\text{phys}} = \tau_{\text{int}} \times \texttt{tsamp}$$

where `tsamp` is the number of simulation steps between recorded force values.

Effective Number of Independent Samples: Given the total number of recorded force samples N_{total}, the number of effectively independent samples is:

$$N_{\text{eff}} = \frac{N_{\text{total}}}{2\tau_{\text{phys}}}$$

This corrects for the fact that correlated measurements provide less statistical information than fully independent ones.

Corrected Standard Error: To estimate the statistical uncertainty of force averages, we use the corrected standard error formula:

$$\text{Standard Error} = \frac{\sigma}{\sqrt{N_{\text{eff}}}}$$

where σ is the standard deviation of the force values. This adjustment ensures that error bars account for the underlying correlations in the data, preventing underestimation of uncertainties.

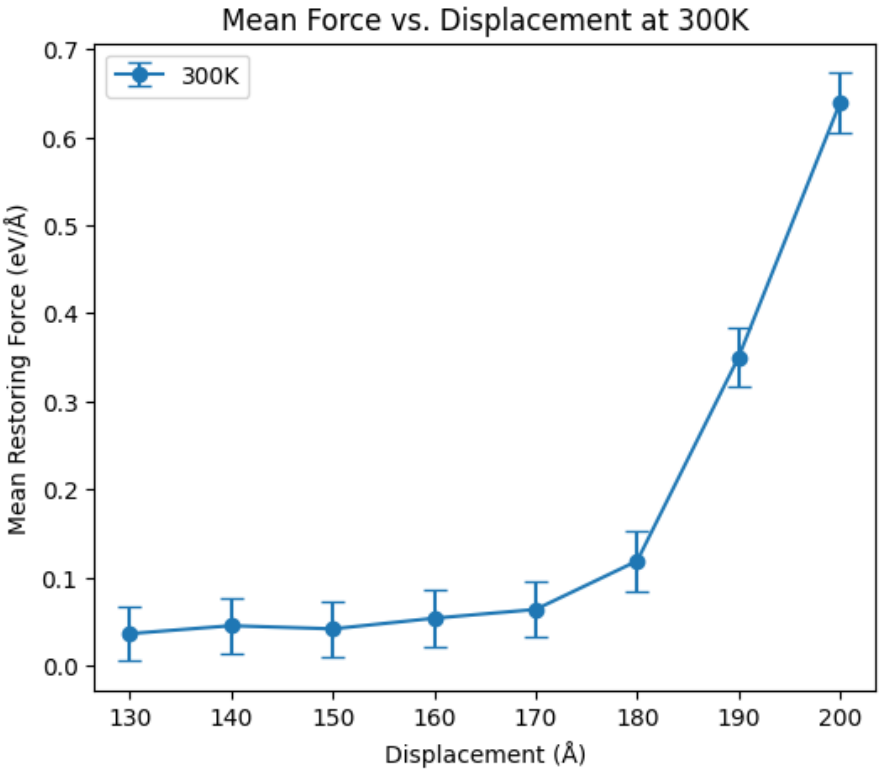

Figure 8: Restoring force vs. displacement at 300K with error bars.

Figure 8 provides a focused view of the mean restoring force at 300K with corresponding error bars. In the entropic regime ($130\mathring{A} \leq L \leq 170\mathring{A}$), the force remains low and increases gradually. However, the error bars show a clear trend of increasing with displacement. This increase arises because, as the polymer is stretched further, the number of accessible configurations decreases, leading to greater fluctuations in force as fewer

pathways remain for molecular motion. This reinforces the idea that entropic elasticity dominates in this region, with the force primarily driven by configurational entropy rather than direct bond stretching.

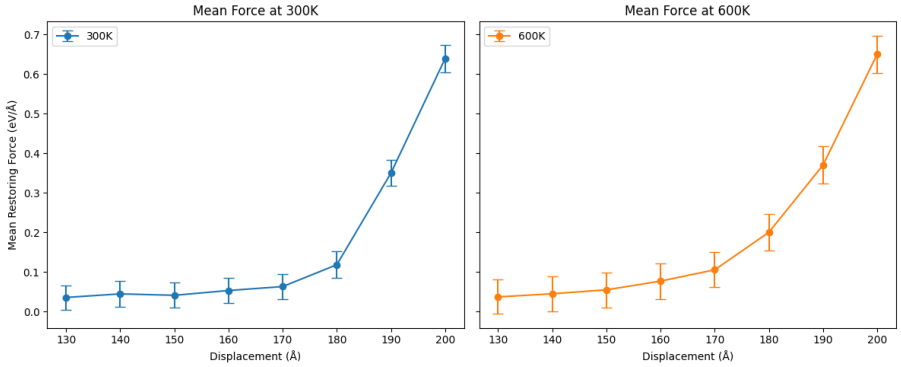

Figure 9: Comparison of restoring force vs. displacement at 300K and 600K, including error bars.

In Figure 9, we compare force-displacement relationships at both 300K and 600K, incorporating error bars for both datasets. A key observation is that while force increases with temperature, the relative error also grows at 600K. This is consistent with expectations from entropic elasticity: at higher temperatures, thermal fluctuations are amplified, leading to greater force variance. We further confirm this by plotting histograms for all restoring forces (Figure 13). The increasing error with displacement, shown later in Figure 10 is similarly observed at both temperatures, further highlighting that entropic effects become increasingly constrained as the polymer is stretched, causing more pronounced fluctuations in restoring force.

4.4 Temperature Scaling of Force Fluctuations

In Figure 10, we examine how the standard deviation of force fluctuations changes with temperature. As the fluctuations follow entropic elasticity, the standard deviation should scale as $\sigma_F \propto \sqrt{T}$ [8]. This is supported by the plot, where we see this relationship as approximately correct.

4.5 Internal Energy vs. Displacement

Figure 11 presents the variation of internal energy with displacement for both 300K and 600K. The first key observation is that the internal energy remains relatively stable for most displacements, showing only minor fluctuations. This is expected, as internal energy in the entropic regime should not vary significantly with extension, reinforcing the idea that molecular rearrangements rather than energetic contributions dominate elasticity in this range.

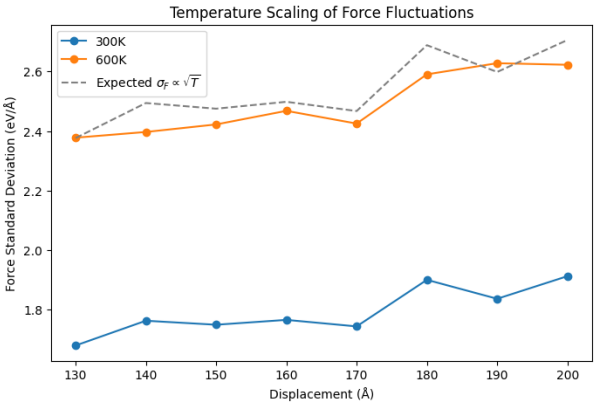

Figure 10: Standard deviation of restoring forces

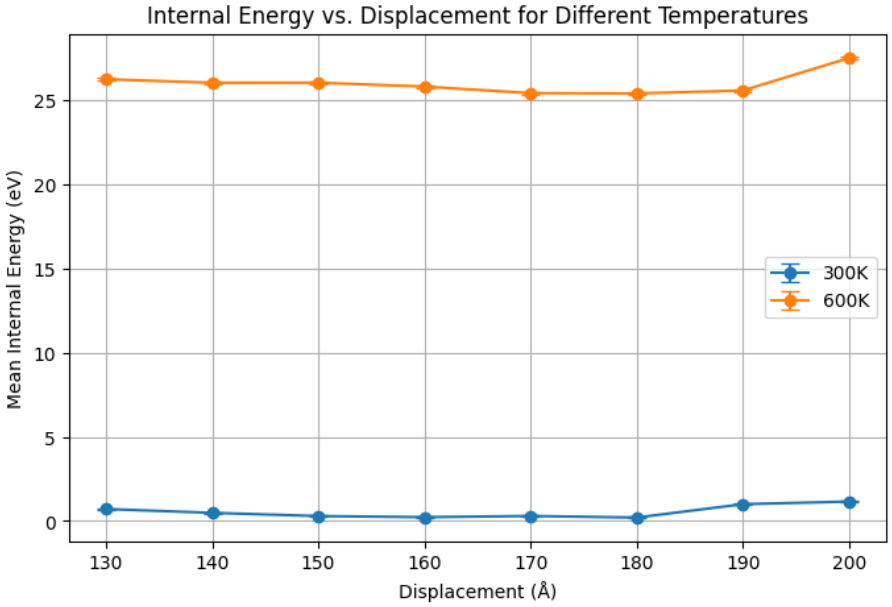

Figure 11: Mean internal energy vs. displacement at 300K and 600K with error bars.

At 600K, the internal energy is consistently higher than at 300K, as expected from thermal energy scaling. The error bars remain small across the displacement range, indicating that the internal energy measurements are statistically stable despite autocorrelations. Since the error bars for internal energy are much smaller compared to those for the restoring force, this suggests that internal energy measurements contain more independent samples over the simulation time.

A slight increase in internal energy at the largest displacements is observed, beginning at $180\mathring{A}$ for the 300K temperature and $190\mathring{A}$ for the 600K temperature. This is because at high strains, bond stretching contributes to the restoring force.

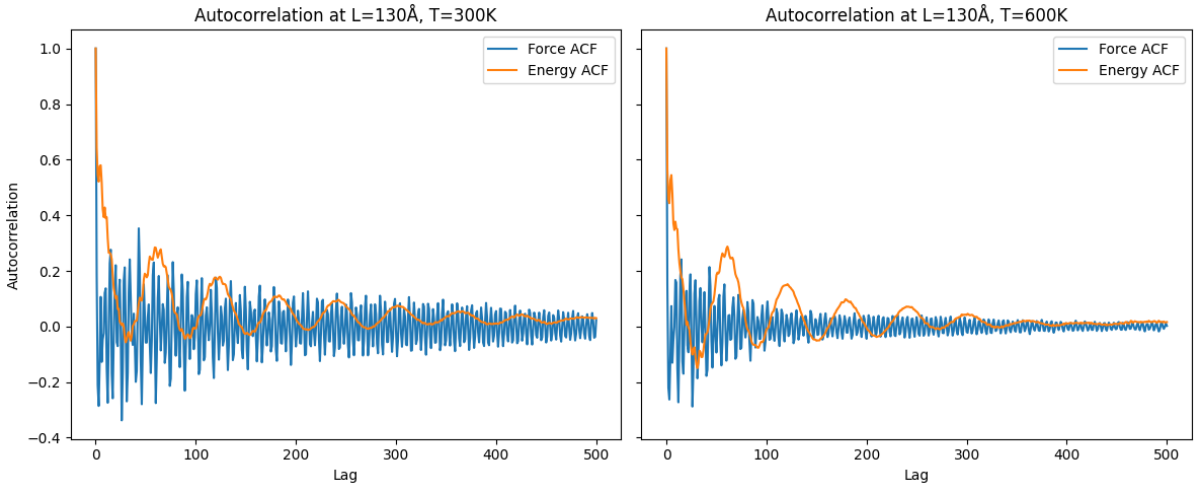

Figure 12: Comparison of autocorrelation functions (ACF) for force and internal energy at $L = 130$Å for both temperatures.

4.6 Autocorrelation of Force vs. Energy Fluctuations

Figure 12 compares the autocorrelation functions (ACF) of force and internal energy at $L = 130$Å for both 300K and 600K. Let us consider the significant difference in autocorrelation decay between force and energy. The force ACF oscillates more rapidly and exhibits a faster decay, indicating that force fluctuations decorrelate quickly. This suggests that force measurements experience rapid fluctuations over short timescales, likely due to immediate responses to molecular vibrations and interactions.

In contrast, the energy ACF shows a slower decay with higher-amplitude oscillations, suggesting that energy fluctuations persist for longer timescales. This aligns with the idea that internal energy variations are governed by slower conformational rearrangements rather than instantaneous fluctuations. The longer correlation time for energy fluctuations also explains the slightly smaller error bars in energy measurements compared to force measurements, as the effective sample size remains larger. Force fluctuations occur on shorter timescales than energy fluctuations, reflecting their different molecular origins.

In Figure 12, we note that at 600K, the autocorrelation function (ACF) for force exhibits larger oscillation peaks but lower amplitude compared to 300K. This suggests increased thermal fluctuations leading to more rapid decorrelation of force measurements. This is because at higher temperatures, thermal fluctuations increase, allowing the polymer to explore a larger number of configurations. This leads to stronger energy fluctuations, which persist longer before decorrelating, resulting in larger peaks in the energy ACF. Conversely, force fluctuations decorrelate more quickly at higher temperatures due to increased molecular randomness, causing the force ACF to have lower and more damped peaks. This distinction arises because energy evolves over longer timescales, while force is a rapidly fluctuating, local response to molecular interactions.

5 Conclusion

In this study, we investigated the elasticity of rubber using molecular dynamics simulations, focusing on the relationship between displacement, restoring force, and internal energy. Our results confirm that in the entropic regime, restoring force scales approximately linearly with displacement. The statistical uncertainty in force measurements increased with displacement, highlighting the need for autocorrelation corrections.

Overall, this study demonstrates the utility of molecular dynamics in probing polymer elasticity and provides a framework for statistically robust measurements. Future work could explore higher strain regimes to further characterise the transition between entropic and enthalpic elasticity, as well as the effects of cross-linking and chain length on restoring force behavior.

References

[1] Gábor Csányi. Lecture 1. Lecture notes, Materials and Molecules: Simulation, Modelling and Machine Learning, University of Cambridge, 2025. Accessed: 2025-03-18.

[2] Gábor Csányi. Lecture 2. Lecture notes, Materials and Molecules: Simulation, Modelling and Machine Learning, University of Cambridge, 2025. Accessed: 2025-03-18.

[3] Gábor Csányi. Lecture 3. Lecture notes, Materials and Molecules: Simulation, Modelling and Machine Learning, University of Cambridge, 2025. Accessed: 2025-03-18.

[4] Gábor Csányi. Lecture 4. Lecture notes, Materials and Molecules: Simulation, Modelling and Machine Learning, University of Cambridge, 2025. Accessed: 2025-03-18.

[5] Gábor Csányi. Lecture 6. Lecture notes, Materials and Molecules: Simulation, Modelling and Machine Learning, University of Cambridge, 2025. Accessed: 2025-03-18.

[6] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, 2nd edition, 2002.

[7] L. R. G. Treloar. *The Physics of Rubber Elasticity*. Oxford University Press, 1975.

[8] James T. Waters and Harold D. Kim. Calculation of a fluctuating entropic force by phase space sampling. *Physical Review E*, 92(1):013308, 2015.

6 Appendix

6.1 Histograms displaying restoring forces

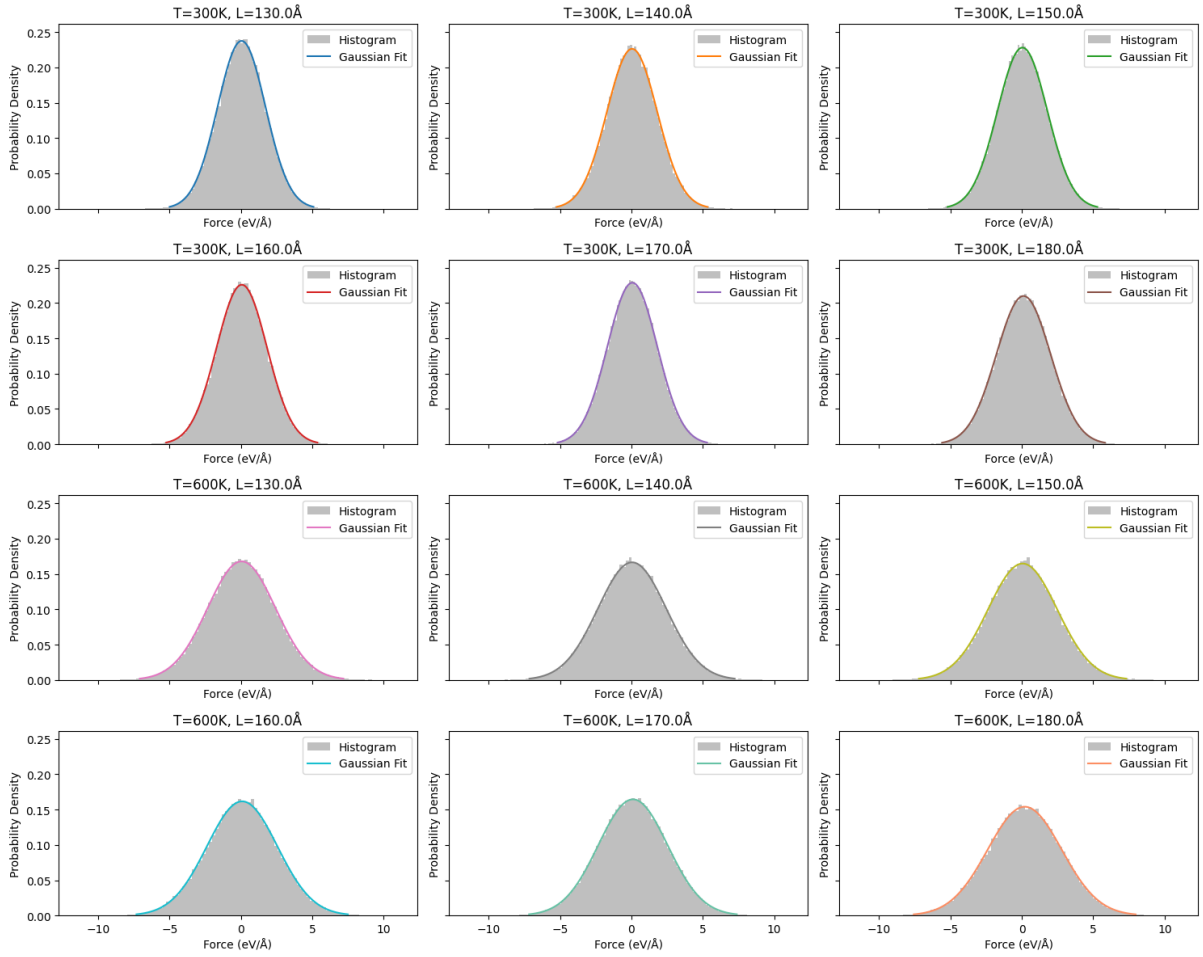

Figure 13: Histograms displaying the distribution of restoring forces

6.2 Code - Polyisoprene

Polyisoprene Code

```
# %% [markdown]
# 4G5 Coursework Computer Project: rubber elasticity
# ====
#
# <!--<img src="./isoprene.png" height=200>-->
# <img src="./polyisoprene.png" height=200>
#
```

```
# In this project, you are going to investigate the elastic properties
↪  of rubber. There are many different types of rubber, both synthetic
↪  and
# natural, the latter mostly derived from the fluids of the [rubber
↪  tree](https://en.wikipedia.org/wiki/Hevea_brasiliensis). All of
↪  them have a
# common structure: extremely long chains of flexible polymers.
↪  Different materials vary in the composition of the polymers,
↪  whether and how
# cross-linked they are. In this project, we are going to consider the
↪  polyisoprene molecule, which is the major constitutent of
# natural rubber. Isoprene is a simple hydrocarbon, a naturally very
↪  abundant molecule, even humans produce some! To keep things simple,
↪  you
# will study only a single polymer chain, and to keep the computational
↪  effort low, only a short one. This is already sufficient to display
# the main phenomenology of elasticity.

# %% [markdown]
# The main objective of the project is to demonstrate the linear
↪  restoring force as a function of displacement for the polyisoprene
↪  molecule.
# This will be accomplished by _constrained_ molecular dynamics at
↪  constant temperature, in which the two ends of the molecule will be
↪  kept at a
# given distance apart, and the molecular motion simulated as it
↪  explores the allowable conformations. After sufficient
# data is accumulated, the average force on the end points is recorded,
↪  and a new, larger distance is set, repeating the previous
↪  procedure. You will
# need to use error analysis to determine how long a simulation to do
↪  at a fixed displacement before moving on to the next one.
#
# __Your report__ should contain the measured average restoring force
↪  as a function of displacement, with appropriate error bars, and
↪  brief commentary on what
# you have found. Also consider the average internal energy of molecule
↪  (also as a function of displacement), and comment on its
↪  relationship to the
```

```
# restoring force.
#
# Notes:
#
# - If you find that you are spending more than 6 hours on the
↪  coursework, seek help. This does _not_ include the runtime of the
↪  simulation that you use to gather
# your final data (after you've done shorter exploratory work), it is
↪  recommended that you run it overnight. Always make an estimate on
↪  how
# long a given run will take, never start a simulation for which you
↪  have no idea when it will finish!
# - Don't forget that the coursework is marked anonymously, so make
↪  sure that you include a [coursework cover
↪  sheet](http://teaching.eng.cam.ac.uk/node/4171) as the
# first page of your report that you upload to Moodle.
# - The marking will be focused on your understanding of the modelling
↪  and data analysis, rather than on programming. If you are stuck,
↪  seek help.
#

# %% [markdown]
# Preliminaries
# ----

# %%
# Pre - Preliminaries
#!pip install numpy # who hasn't got numpy installed?
!pip install ase
!pip install torchani

# %%
#
# import basic atomistic simulation modules
#
import numpy as np
import ase
from ase.build import bulk
from ase.data.pubchem import pubchem_atoms_search
```

```
from ase.visualize import view
import matplotlib.pyplot as plt
import pandas as pd
from statsmodels.tsa.stattools import acf


# %%
# get the structure of isoprene
isoprene = pubchem_atoms_search(smiles="CC=C(C)C")
view(isoprene, viewer='x3d')

# %%
# import an energy model
import sys
sys.path.insert(0, "ANI")
import ani

# %%
isoprene.calc = ani.calculator
isoprene.get_potential_energy()

# %%
#
# now import the modules we need to run molecular dynamics
#
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase.md.verlet import VelocityVerlet
from ase import units
from ase.md.langevin import Langevin
from ase.io.trajectory import Trajectory


# %%
# this initialises the velocities by drawing from the appropriate
↪  distribution at T=300K
MaxwellBoltzmannDistribution(isoprene, temperature_K=300)

# We now create a molecular dynamics object that can be used to run
↪  Langevin dynamics
```

```
# the parameters after the structure are the time step, the
↪  temperature, and the friction constant (in units of picoseconds)
dynamics = Langevin(isoprene, temperature_K=300, timestep=0.5*units.fs,
↪  friction=0.01)


# While the dynamics is running, we want to collect some data! This is
↪  achieved by creating a function to the dynamics object
# which gets called after some number of steps, and it can report to us
↪  what is happenning to the molecule, record its trajectory, etc.
xyzfile = open('isoprene.xyz', 'w') # the file we are going to record
↪  the structures to, the visualiser application "Ovito" can read such
↪  XYZ files.
def report():
    print("Time: {:.3f} fs  |  Potential Energy: {:.3f} eV  |  Kinetic
    ↪  Energy: {:.3f} K".format(dynamics.get_time()/units.fs,
    ↪  isoprene.get_potential_energy(),

        ↪  isoprene.get_kinetic_energy()/(len(isoprene)*(3.0/2.0)*units.kB)))
    ase.io.write(xyzfile, isoprene, format="extxyz")

# notice how we print the kinetic energy in units of Kelvin, but it is
↪  not the thermodynamic temperature (which is kept constant)

dynamics.attach(report, interval=1) # the interval argument specifies
↪  how many steps of dynamics are run between each call to record the
↪  trajectory
dynamics.run(100)
xyzfile.close()
del dynamics

# After execution, you will see a new file called "isoprene.xyz". You
↪  can look at it, and see what it records. Download it, and view the
↪  molecular
# motion using the "Ovito" application.

# %%
# Ovito also helps you to select atoms by dragging the
↪  "Particles/GLobal Attributes" tab upwards a bit,
```

```
# and selecting the little target crosshairs on the left, then clicking
↪  on individual atoms. The "ParticleIndex" variable tells you the
↪  order
# of the atoms in the file and the ASE atoms object.

# For example, particle indices 8 and 13 correspond to two H atoms on
↪  opposite ends of the isoprene molecule, and we can get their
↪  distance:
dr = (isoprene.get_positions()[8,:]-isoprene.get_positions()[13,:])
print("relative displacement vector:", dr)
r = np.linalg.norm(dr)
print("distance: ", r)
dru = dr/np.linalg.norm(dr)
print("unit vector in the same direction:", dru)

# %% [markdown]
# Exercise 1 : dynamics and autocorrelation
# ----
#
# Run molecular dynamics of isoprene, and record the distance between
↪  atoms 8 and 13 (two hydrogens) every 10 steps. Plot the time
↪  evolution
# of this distance, and calculate the autocorrelation function. Run the
↪  trajectory long enough that you sample the methyl groups rotating
↪  around
# the C-C bond. Repeat the exercise at higher temperature (e.g. 500 K),
↪  what happens to the autocorrelation function ?
#
# Note: you can calculate the autocorrelation yourself, or learn to use
↪  the `acf` function in  `statsmodels.tsa.stattools`

# %%
# ---------------------------------- Initial Notes
↪  ----------------------------------
'''

We need to:
1. run molecular dynamics of isoprene
2. record the distance between atoms 8 and 13 every 10 steps
```

```
3. plot the time evolution of this distance
4. calculate the autocorrelation function (on what)

Initial Notes

We have the code which currently:
- initialises velocity of molecules and particles
- creates the molecular dynamics object
- have a 'report' function which gets called after a certain number of
↪  steps to report
- dynamics.run(100): runs 100 steps of our molecular simulation
- dynamics.attach(report, interval = n): reports every n steps of our
↪  simulation
- .get_positions()[k,:] gets the position (in 3-dimensions) of our
↪  atoms in our molecules

- we need to investigate the acf function


- first steps - lets run the dynamics and report every 10 time steps
↪  the distance between the two atoms

'''
# ------------------------------- Initial Code
↪  --------------------------------


# this initialises the velocities by drawing from the appropriate
↪  distribution at T=300K
MaxwellBoltzmannDistribution(isoprene, temperature_K=300)

# We now create a molecular dynamics object that can be used to run
↪  Langevin dynamics
# the parameters after the structure are the time step, the
↪  temperature, and the friction constant (in units of picoseconds)
dynamics = Langevin(isoprene, temperature_K=300, timestep=0.5*units.fs,
↪  friction=0.01)

xyz_file = open('exercise_1_300K.xyz', 'w')
```

```
distance_df_300 = []
full_distance_df_300 = []

def report():

    pos_8 = isoprene.get_positions()[8,:] # retrieve position 8
    pos_13 = isoprene.get_positions()[13,:] # retrieve position 13

    distance = pos_8 - pos_13 # find distance (3 dim vector)
    scalar_distance = np.linalg.norm(distance) # norm of distance (1
    ↪ dim)

    distance_df_300.append(scalar_distance) # add to list of values for
    ↪ plotting
    full_distance_df_300.append(distance) # add to list of values for
    ↪ plotting

    # maybe you don't wanna see this - looks kinda cool tho
    print("Time: {:.3f} fs  |  Potential Energy: {:.3f} eV  |  Kinetic
    ↪ Energy: {:.3f} K | Distance btw. atoms 8 and 13: {:.3f}".format(
        dynamics.get_time()/units.fs, isoprene.get_potential_energy(),
        isoprene.get_kinetic_energy()/(len(isoprene)*(3.0/2.0)*units.kB),
        scalar_distance))

    # writes to the xyz file
    ase.io.write(xyz_file, isoprene, format="extxyz")


dynamics.attach(report, interval = 10)
dynamics.run(5000)

xyzfile.close()
del dynamics

# Convert data into Pandas DataFrame for analysis
df_300 = pd.DataFrame(full_distance_df_300, columns=['dx', 'dy', 'dz'])
df_300['distance'] = distance_df_300
```

```
# %%
# ----  PLOTTING TIME EVOLUTION OF DISTANCE ----
plt.figure(figsize=(8, 5))
plt.plot(df_300['distance'], label="Distance (Å)")
plt.xlabel("Time (fs)")
plt.ylabel("Distance (Å)")
plt.title("Time Evolution of Distance Between Atoms 8 and 13 for 300K")
plt.legend()
plt.show()

# ----  COMPUTE AND PLOT AUTOCORRELATION FUNCTION (ACF) ----
acf_values = acf(df_300['distance'], nlags=40, fft=True)  # Compute ACF
↪  with FFT
lags = np.arange(len(acf_values)) * 10  # Convert index to time lag (fs)

plt.figure(figsize=(8, 5))
plt.plot(lags, acf_values, label="Autocorrelation")
plt.xlabel("Time Lag (fs)")
plt.ylabel("ACF")
plt.title("Autocorrelation of Distance Between Atoms 8 and 13 for 300K")
plt.legend()
plt.show()

# %%
# ---------------------------------- Initial Notes
↪  ----------------------------------
'''

We need to:
1. run molecular dynamics of isoprene
2. record the distance between atoms 8 and 13 every 10 steps
3. plot the time evolution of this distance
4. calculate the autocorrelation function (on what)

Initial Notes

We have the code which currently:
- initialises velocity of molecules and particles
- creates the molecular dynamics object
```

```
- have a 'report' function which gets called after a certain number of
↪  steps to report
- dynamics.run(100): runs 100 steps of our molecular simulation
- dynamics.attach(report, interval = n): reports every n steps of our
↪  simulation
- .get_positions()[k,:] gets the position (in 3-dimensions) of our
↪  atoms in our molecules

- we need to investigate the acf function


- first steps - lets run the dynamics and report every 10 time steps
↪  the distance between the two atoms

'''
# ---------------------------------- Initial Code
↪  ----------------------------------


# this initialises the velocities by drawing from the appropriate
↪  distribution at T=500K
MaxwellBoltzmannDistribution(isoprene, temperature_K=500)

# We now create a molecular dynamics object that can be used to run
↪  Langevin dynamics
# the parameters after the structure are the time step, the
↪  temperature, and the friction constant (in units of picoseconds)
dynamics = Langevin(isoprene, temperature_K=500, timestep=0.5*units.fs,
↪  friction=0.01)

xyz_file = open('exercise_1_500K.xyz', 'w')

distance_df_500 = []
full_distance_df_500 = []

def report():

    pos_8 = isoprene.get_positions()[8,:] # retrieve position 8
    pos_13 = isoprene.get_positions()[13,:] # retrieve position 13
```

```
    distance = pos_8 - pos_13 # find distance (3 dim vector)
    scalar_distance = np.linalg.norm(distance) # norm of distance (1
    ↪ dim)

    distance_df_500.append(scalar_distance) # add to list of values for
    ↪ plotting
    full_distance_df_500.append(distance) # add to list of values for
    ↪ plotting

    # maybe you don't wanna see this - looks kinda cool tho
    print("Time: {:.3f} fs  |  Potential Energy: {:.3f} eV  |  Kinetic
    ↪ Energy: {:.3f} K | Distance btw. atoms 8 and 13: {:.3f}".format(
        dynamics.get_time()/units.fs, isoprene.get_potential_energy(),
        isoprene.get_kinetic_energy()/(len(isoprene)*(3.0/2.0)*units.kB),
        scalar_distance))

    # writes to the xyz file
    ase.io.write(xyz_file, isoprene, format="extxyz")


dynamics.attach(report, interval = 10)
dynamics.run(5000)

xyzfile.close()
del dynamics

# Convert data into Pandas DataFrame for analysis
df_500 = pd.DataFrame(full_distance_df_500, columns=['dx', 'dy', 'dz'])
df_500['distance'] = distance_df_500

# %%
# ----  PLOTTING TIME EVOLUTION OF DISTANCE ----
plt.figure(figsize=(8, 5))
plt.plot(df_500['distance'], label="Distance (Å)")
plt.xlabel("Time (fs)")
plt.ylabel("Distance (Å)")
plt.title("Time Evolution of Distance Between Atoms 8 and 13 for 500K")
plt.legend()
```

```
plt.show()

# ----  COMPUTE AND PLOT AUTOCORRELATION FUNCTION (ACF) ----
acf_values = acf(df_500['distance'], nlags=40, fft=True)  # Compute ACF
↪  with FFT
lags = np.arange(len(acf_values)) * 10  # Convert index to time lag (fs)

plt.figure(figsize=(8, 5))
plt.plot(lags, acf_values, label="Autocorrelation")
plt.xlabel("Time Lag (fs)")
plt.ylabel("ACF")
plt.title("Autocorrelation of Distance Between Atoms 8 and 13 for 500K")
plt.legend()
plt.show()

# %%
plt.figure(figsize=(8, 5))
plt.hist(df_300['distance'], bins=30, alpha=0.5, label="300K")
plt.hist(df_500['distance'], bins=30, alpha=0.5, label="500K")
plt.xlabel("Distance Between Atoms 8 and 13 (Å)")
plt.ylabel("Frequency")
plt.title("Distribution of Distances Between Atoms 8 and 13 at 300K and
↪  500K")
plt.legend()
plt.show()


# %%
acf_300 = acf(df_300['distance'], nlags=100, fft=True)
acf_500 = acf(df_500['distance'], nlags=100, fft=True)

plt.figure(figsize=(8, 5))
plt.plot(acf_300, label="300K", linestyle="--")
plt.plot(acf_500, label="500K", linestyle="-")
plt.xlabel("Time Lag")
plt.ylabel("ACF")
plt.title("ACF for 300K vs 500K")
plt.legend()
plt.show()
```

```
# %%
from scipy.optimize import curve_fit

def exp_decay(t, tau):
    return np.exp(-t / tau)

lags = np.arange(len(acf_300)) * 10  # Convert index to time lag in fs

tau_300, _ = curve_fit(exp_decay, lags[:50], acf_300[:50])  # Fit first
↪  part of the decay
tau_500, _ = curve_fit(exp_decay, lags[:50], acf_500[:50])  # Fit first
↪  part of the decay

print(f"Correlation Time 300K: {tau_300[0]:.2f} fs")
print(f"Correlation Time 500K: {tau_500[0]:.2f} fs")


# %% [markdown]
# Polyisoprene
# ---

# %%
# we now create a polyisoprene molecule
polyisoprene =
↪  pubchem_atoms_search(smiles="CC=C(C)CCC=C(C)CCC=C(C)CCC=C(C)CCC=C(C)CCC=C(C)C")
view(polyisoprene, viewer='x3d')

# %% [markdown]
# Main Task
# ---
# Create a moleculary dynamics simulation in which you stretch the
↪  molecule very slowly, and measure the restoring force as a function
# of displacement (I suggest using method 2 from above).
# Notes:
# - You are  interested in the force _between_ the two molecules held
↪  fixed (i.e. the force difference) and only in the component of the
↪  force along
```

```
# the line connecting the two fixed atoms.
# - It is enough to record the structure every 100 steps or so, and be
↪  prepared to run the simulation for 100,000 steps or more for each
↪  fixed distance.
# - I recommend that you create and record the trajectory first, and
↪  the analyse it afterwards, because the main computational cost is
↪  creating the trajectory.
# - When analysing the data, do not include data immediately after each
↪  stretch, allow the system to relax towards the equilibrium
↪  distribution for a few
# thousand steps before collecting the force data.
# - Calculate the autocorrelation of the restoring force value and use
↪  it to compute error bars on your measurement

# %% [markdown]
# Optional extension
# ---
#
# The fact that the restoring force is entropic in origin has
↪  implications for the temperature dependence of the restoring force.
↪  Repeat the main task for
# different temperature settings of the Langevin dynamics, and compare
↪  the results!
```

6.3 Code - Data Analysis

Data Analysis Code

```
# %%
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
from statsmodels.tsa.stattools import acf

# %% [markdown]
# # Data Analysis
```

```
#
# Here we conduct data analysis of the molecular dynamics simulation

# %% [markdown]
# # FINAL RESULTS

# %%
# Define all displacement (L) and temperature (T) values
L_values = list(range(130, 210, 10))  # L = 130, 140, ..., 200
T_values = [300, 600]  # T = 300K, 600K
Ns = 100000
Neq = 25000
Nrun = 1000000
ts = 10
td = 1000

# Base directory
base_dir = "lammps_project"

# %%
# Create an empty list to store data
data_list = []

# Loop over each L and T
for L in L_values:
    for T in T_values:
        folder_name =
        ↪  f"L{L}_T{T}_Ns{Ns}_Neq{Neq}_Nrun{Nrun}_ts{ts}_td{td}"
        forces_file = os.path.join(base_dir, folder_name,
        ↪  f"lmp{L}.forces")
        log_file = os.path.join(base_dir, folder_name, f"lmp{L}.log")

        # Load force and displacement data
        displacement, restoring_force = load_forces(forces_file)
        energy_data = load_energy(log_file)

        # Store as a dictionary
        if displacement is not None and restoring_force is not None:
```

```
            for i in range(len(displacement)):  # Loop over each data
            ↪ point
                data_list.append({
                    "L": L,
                    "T": T,
                    "displacement": displacement[i],
                    "force": restoring_force[i],
                    "energy": energy_data[i] if energy_data is not None
                    ↪  and i < len(energy_data) else np.nan
                })

# Convert to Pandas DataFrame
df = pd.DataFrame(data_list)


# %%
import matplotlib.pyplot as plt
import numpy as np

# Compute mean force per displacement & temperature
mean_force_df = df.groupby(["displacement",
↪  "T"])["force"].mean().reset_index()

plt.figure(figsize=(8,5))

for T in [300, 600]:
    temp_df = mean_force_df[mean_force_df["T"] == T]
    plt.plot(temp_df["displacement"], temp_df["force"], marker="o",
    ↪  linestyle="-", label=f"{T}K")

plt.xlabel("Displacement (Å)")
plt.ylabel("Mean Restoring Force (eV/Å)")
plt.title("Mean Force vs. Displacement for Different Temperatures")
plt.legend()
plt.show()



# %%
```

```
import matplotlib.pyplot as plt
import numpy as np

# Compute mean force per displacement at 300K
mean_force_300K = df[df["T"] ==
↪  300].groupby("displacement")["force"].mean().reset_index()

# Create the plot
plt.figure(figsize=(8, 5))
plt.plot(mean_force_300K["displacement"], mean_force_300K["force"],
         marker="o", linestyle="-", color="tab:blue", label="300K")

# Labels and title
plt.xlabel("Displacement (Å)")
plt.ylabel("Mean Restoring Force (eV/Å)")
plt.title("Mean Force vs. Displacement at 300K")
plt.legend()

# Show the plot
plt.show()


# %%
import matplotlib.pyplot as plt
import numpy as np

# Compute mean force per displacement & temperature
mean_force_df = df.groupby(["displacement",
↪  "T"])["force"].mean().reset_index()

# Define colors (blue for 300K, orange for 600K)
colors = {300: "tab:blue", 600: "tab:orange"}

# Create subplots
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

for i, T in enumerate([300, 600]):
    temp_df = mean_force_df[mean_force_df["T"] == T]
    axes[i].plot(temp_df["displacement"], temp_df["force"],
```

```
                marker="o", linestyle="-", color=colors[T],
                ↪  label=f"{T}K")

    axes[i].set_xlabel("Displacement (Å)")
    axes[i].set_title(f"Mean Force at {T}K")
    axes[i].legend()

# Set common Y-axis label
axes[0].set_ylabel("Mean Restoring Force (eV/Å)")

# Adjust layout and show the plot
plt.tight_layout()
plt.show()



# %%
import matplotlib.pyplot as plt
import numpy as np

# Compute mean force per displacement & temperature
mean_force_df = df.groupby(["displacement",
↪  "T"])["force"].mean().reset_index()

# Define colors (blue for 300K, orange for 600K)
colors = {300: "tab:blue", 600: "tab:orange"}

# Create subplots (3 plots in a row)
fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

# Plot for 300K
temp_df_300 = mean_force_df[mean_force_df["T"] == 300]
axes[0].plot(temp_df_300["displacement"], temp_df_300["force"],
             marker="o", linestyle="-", color=colors[300], label="300K")
axes[0].set_xlabel("Displacement (Å)")
axes[0].set_ylabel("Mean Restoring Force (eV/Å)")
axes[0].set_title("Mean Force at 300K")
axes[0].legend()

# Plot for 600K
```

```
temp_df_600 = mean_force_df[mean_force_df["T"] == 600]
axes[1].plot(temp_df_600["displacement"], temp_df_600["force"],
             marker="o", linestyle="-", color=colors[600], label="600K")
axes[1].set_xlabel("Displacement (Å)")
axes[1].set_title("Mean Force at 600K")
axes[1].legend()

# Overlaid plot for comparison
axes[2].plot(temp_df_300["displacement"], temp_df_300["force"],
             marker="o", linestyle="-", color=colors[300], label="300K")
axes[2].plot(temp_df_600["displacement"], temp_df_600["force"],
             marker="o", linestyle="-", color=colors[600], label="600K")
axes[2].set_xlabel("Displacement (Å)")
axes[2].set_title("Mean Force Comparison")
axes[2].legend()

# Adjust layout and show the plot
plt.tight_layout()
plt.show()


# %% [markdown]
# ### Tau_int

# %%
import numpy as np
import pandas as pd
from statsmodels.tsa.stattools import acf

# Define function to compute tau_int
def compute_tau_int(series, nlags=1000):
    """Compute the integrated autocorrelation time (tau_int) for a
    ↪  given time series."""
    acf_values = acf(series, nlags=nlags, fft=True)

    # Find cutoff M (where ACF first crosses zero)
    M = np.argmax(acf_values < 0) if np.any(acf_values < 0) else nlags  #
    ↪  Use nlags if no zero-crossing
```

```
    # Compute tau_int using the sum formula
    tau_int = 0.5 + np.sum(acf_values[:M])
    return tau_int

# Compute tau_int for each displacement-temperature pair
tau_int_results = []
for (L, T), group in df.groupby(["displacement", "T"]):
    tau = compute_tau_int(group["force"])  # No extra multiplication
    ↪ here!
    tau_int_results.append({"displacement": L, "T": T, "tau_int": tau *
    ↪ 10})  # Apply correction only once!

# Convert to DataFrame
tau_int_df = pd.DataFrame(tau_int_results)


# Convert to DataFrame
tau_int_df = pd.DataFrame(tau_int_results)

# Show computed tau_int values
print(tau_int_df)


# %% [markdown]
# ### Error Bars

# %%
# Compute tau_int for each displacement-temperature pair
tau_int_results = []
for (L, T), group in df.groupby(["displacement", "T"]):
    tau = compute_tau_int(group["force"])  # No extra multiplication
    ↪ here!
    tau_int_results.append({"displacement": L, "T": T, "tau_int": tau *
    ↪ 10})  # Apply correction only once!

# Convert to DataFrame
tau_int_df = pd.DataFrame(tau_int_results)
```

```
# Convert to DataFrame
tau_int_df = pd.DataFrame(tau_int_results)

# Recompute mean and standard deviation of force per displacement &
↪ temperature
force_stats_df = df.groupby(["displacement", "T"])["force"].agg(["mean",
↪ "std"]).reset_index()
force_stats_df.rename(columns={"mean": "force_mean", "std": "force_std"},
↪ inplace=True)

# Merge the exact N_total values and tau_int values
force_stats_df = force_stats_df.merge(N_total_df, on=["displacement",
↪ "T"])
force_stats_df = force_stats_df.merge(tau_int_df, on=["displacement",
↪ "T"])

# Compute effective sample size using actual tau_int values
force_stats_df["N_eff"] = force_stats_df["N_total"] / (2 *
↪ force_stats_df["tau_int"])

# Compute standard error using the correct formula
force_stats_df["error"] = force_stats_df["force_std"] /
↪ np.sqrt(force_stats_df["N_eff"])

# Define colors (blue for 300K, orange for 600K)
colors = {300: "tab:blue", 600: "tab:orange"}

# Create subplots for 300K and 600K
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

for i, T in enumerate([300, 600]):
    temp_df = force_stats_df[force_stats_df["T"] == T]
    axes[i].errorbar(temp_df["displacement"], temp_df["force_mean"],
                     yerr=temp_df["error"], fmt="o-", color=colors[T],
                     ↪ label=f"{T}K",
                     capsize=5, capthick=1)

    axes[i].set_xlabel("Displacement (Å)")
    axes[i].set_title(f"Mean Force at {T}K")
```

```
    axes[i].legend()

# Set common Y-axis label
axes[0].set_ylabel("Mean Restoring Force (eV/Å)")

# Adjust layout and show the plot
plt.tight_layout()
plt.show()

# Display the computed errors for verification
force_stats_df[["displacement", "T", "tau_int", "error"]]



# %%
# Compute effective sample size (N_eff) using the formula:
# N_eff = N_total / (2 * (10 * tau_int))

# Ensure tau_int is correctly assigned from previous computations
force_stats_df["N_eff"] = force_stats_df["N_total"] / (2 *
↪  force_stats_df["tau_int"])

# Display the computed N_eff values
force_stats_df[["displacement", "T", "tau_int", "N_total", "N_eff"]]



# %%
# Compute tau_int for each displacement-temperature pair (ONLY FOR
↪  300K)
tau_int_results_300K = []
for (L, T), group in df.groupby(["displacement", "T"]):
    if T == 300:  # Filter for only 300K
        tau = compute_tau_int(group["force"])  # No extra multiplication
        ↪  here!
        tau_int_results_300K.append({"displacement": L, "T": T,
        ↪  "tau_int": tau * 10})  # Apply correction only once!

# Convert to DataFrame
tau_int_df_300K = pd.DataFrame(tau_int_results_300K)
```

```
# Recompute mean and standard deviation of force per displacement for
↪  300K
force_stats_df_300K = df[df["T"] == 300].groupby(["displacement",
↪  "T"])["force"].agg(["mean", "std"]).reset_index()
force_stats_df_300K.rename(columns={"mean": "force_mean", "std":
↪  "force_std"}, inplace=True)

# Merge the exact N_total values and tau_int values for 300K
force_stats_df_300K = force_stats_df_300K.merge(N_total_df,
↪  on=["displacement", "T"])
force_stats_df_300K = force_stats_df_300K.merge(tau_int_df_300K,
↪  on=["displacement", "T"])

# Compute effective sample size using actual tau_int values for 300K
force_stats_df_300K["N_eff"] = force_stats_df_300K["N_total"] / (2 *
↪  force_stats_df_300K["tau_int"])

# Compute standard error using the correct formula for 300K
force_stats_df_300K["error"] = force_stats_df_300K["force_std"] /
↪  np.sqrt(force_stats_df_300K["N_eff"])

# Define color for 300K
color_300K = "tab:blue"

# Create plot for 300K only
plt.figure(figsize=(6, 5))
plt.errorbar(force_stats_df_300K["displacement"],
↪  force_stats_df_300K["force_mean"],
             yerr=force_stats_df_300K["error"], fmt="o-",
              ↪  color=color_300K, label="300K",
             capsize=5, capthick=1)

# Labels and title
plt.xlabel("Displacement (Å)")
plt.ylabel("Mean Restoring Force (eV/Å)")
plt.title("Mean Force vs. Displacement at 300K")
plt.legend()

# Show the plot
```

```
plt.show()

# Display the computed errors for verification
force_stats_df_300K[["displacement", "T", "tau_int", "error"]]


# %% [markdown]
# ## Force Distributions

# %%
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# Select displacement values to plot
L_values = list(range(130, 210, 10))  # L = 130, 140, ..., 200

fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=True)  # Two
↪  subplots for 300K & 600K

for i, T in enumerate([300, 600]):
    ax = axes[i]

    for L in L_values:
        temp_df = df[(df["T"] == T) & (df["displacement"] == L)]["force"]

        # Compute Gaussian fit parameters
        mu, sigma = np.mean(temp_df), np.std(temp_df)
        x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)

        # Plot Gaussian curve
        ax.plot(x, stats.norm.pdf(x, mu, sigma), label=f"L={L}Å")

    # Labels and title
    ax.set_xlabel("Force (eV/Å)")
    ax.set_title(f"Force Distribution at {T}K")
    ax.legend()

# Common y-axis label
```

```
fig.supylabel("Probability Density")
plt.tight_layout()
plt.show()


# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

# Compute standard deviation of force fluctuations per displacement &
↪  temperature
force_fluctuations = df.groupby(["displacement",
↪  "T"])["force"].std().reset_index()
force_fluctuations.rename(columns={"force": "force_std"}, inplace=True)

# Create subplots for force fluctuation distributions
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

# Plot force distributions with Gaussian fits
for i, T in enumerate([300, 600]):
    ax = axes[i]
    for L in df["displacement"].unique():
        subset = df[(df["displacement"] == L) & (df["T"] == T)]["force"]
        mu, sigma = np.mean(subset), np.std(subset)

        x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
        ax.plot(x, stats.norm.pdf(x, mu, sigma), label=f"L={L}Å")

    ax.set_title(f"Force Distribution at {T}K")
    ax.set_xlabel("Force (eV/Å)")
    ax.set_ylabel("Probability Density")
    ax.legend()

plt.tight_layout()
plt.show()
```

```
# ---  TESTING TEMPERATURE SCALING OF FLUCTUATIONS ---
plt.figure(figsize=(8,5))

# Compute expected scaling from sqrt(T)
force_fluctuations["expected_scaling"] = force_fluctuations["force_std"]
↪  * np.sqrt(600 / 300)  # Scaling by sqrt(T)

for T in [300, 600]:
    temp_df = force_fluctuations[force_fluctuations["T"] == T]
    plt.plot(temp_df["displacement"], temp_df["force_std"], marker="o",
    ↪  linestyle="-", label=f"{T}K")

# Plot expected entropic scaling
plt.plot(temp_df["displacement"],
↪  force_fluctuations[force_fluctuations["T"] ==
↪  300]["expected_scaling"],
         linestyle="dashed", color="gray", label="Expected $\sigma_F
         ↪  \propto \sqrt{T}$")

plt.xlabel("Displacement (Å)")
plt.ylabel("Force Standard Deviation (eV/Å)")
plt.title("Temperature Scaling of Force Fluctuations")
plt.legend()
plt.show()


# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats

# Define displacements to plot (6 for each temp)
selected_displacements = sorted(df["displacement"].unique())[:6]  # First
↪  6 displacements

# Create a color cycle for Gaussians (12 unique colors)
colors = plt.cm.tab10.colors + plt.cm.Set2.colors  # Combine 2 colormaps
```

```
# Create a 4x3 subplot grid (2 rows per temperature)
fig, axes = plt.subplots(4, 3, figsize=(15, 12), sharex=True,
↪  sharey=True)

# Set global x and y limits for consistency
force_min, force_max = df["force"].min(), df["force"].max()
y_max = 0  # Placeholder for max y-axis value

# Loop over temperatures and displacements
for i, T in enumerate([300, 600]):
    for j, L in enumerate(selected_displacements):
        ax = axes[i * 2 + (j // 3), j % 3]  # Position in 4x3 grid

        # Get subset of force data
        subset = df[(df["displacement"] == L) & (df["T"] == T)]["force"]

        # Compute Gaussian fit parameters
        mu, sigma = np.mean(subset), np.std(subset)
        x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 100)
        y = stats.norm.pdf(x, mu, sigma)

        # Update max y-axis value for consistency
        y_max = max(y_max, max(y))

        # Plot histogram (same color)
        ax.hist(subset, bins=100, density=True, alpha=0.5, color="gray",
        ↪  label="Histogram")

        # Plot Gaussian fit (unique color per subplot)
        ax.plot(x, y, label="Gaussian Fit", color=colors[i * 6 + j])

        # Titles & labels
        ax.set_title(f"T={T}K, L={L}Å")
        ax.set_xlabel("Force (eV/Å)")
        ax.set_ylabel("Probability Density")
        ax.legend()

# Set the same axis limits for all subplots
for ax in axes.flat:
```

```
    ax.set_xlim(force_min, force_max)
    ax.set_ylim(0, y_max * 1.1)  # Add a bit of padding

# Adjust layout for clarity
plt.tight_layout()
plt.show()


# %% [markdown]
# # Energy

# %%
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from statsmodels.tsa.stattools import acf

# Define function to compute tau_int for energy
def compute_tau_int(series, nlags=1000):
    """Compute the integrated autocorrelation time (tau_int) for a
    ↪ given time series."""
    acf_values = acf(series, nlags=nlags, fft=True)

    # Find cutoff M (where ACF first crosses zero)
    M = np.argmax(acf_values < 0) if np.any(acf_values < 0) else nlags  #
    ↪ Use nlags if no zero-crossing

    # Compute tau_int using the sum formula
    tau_int = 0.5 + np.sum(acf_values[:M])
    return tau_int

# Compute tau_int for energy per displacement-temperature pair
tau_int_energy_results = []
for (L, T), group in df.groupby(["displacement", "T"]):
    tau_energy = compute_tau_int(group["energy"]) * 10  # Apply the 10x
    ↪ correction for time steps
    tau_int_energy_results.append({"displacement": L, "T": T,
    ↪ "tau_int_energy": tau_energy})
```

```
# Convert to DataFrame
tau_int_energy_df = pd.DataFrame(tau_int_energy_results)

# Compute mean and standard deviation of internal energy per
↪  displacement & temperature
energy_stats_df = df.groupby(["displacement",
↪  "T"])["energy"].agg(["mean", "std"]).reset_index()
energy_stats_df.rename(columns={"mean": "energy_mean", "std":
↪  "energy_std"}, inplace=True)

# Merge the exact N_total values and tau_int values for energy
energy_stats_df = energy_stats_df.merge(N_total_df, on=["displacement",
↪  "T"])
energy_stats_df = energy_stats_df.merge(tau_int_energy_df,
↪  on=["displacement", "T"])

# Compute effective sample size for energy using the correct formula:
energy_stats_df["N_eff_energy"] = energy_stats_df["N_total"] / (2 *
↪  energy_stats_df["tau_int_energy"])

# Compute standard error using the correct formula
energy_stats_df["error_energy"] = energy_stats_df["energy_std"] /
↪  np.sqrt(energy_stats_df["N_eff_energy"])

# Define colors for temperature
colors = {300: "tab:blue", 600: "tab:orange"}

# Create a single plot
plt.figure(figsize=(8, 5))

for T in [300, 600]:
    temp_df = energy_stats_df[energy_stats_df["T"] == T]
    plt.errorbar(temp_df["displacement"], temp_df["energy_mean"],
                 yerr=temp_df["error_energy"], fmt="o-", color=colors[T],
                 ↪  label=f"{T}K",
                 capsize=5, capthick=1)

# Labels and title
plt.xlabel("Displacement (Å)")
```

```
plt.ylabel("Mean Internal Energy (eV)")
plt.title("Internal Energy vs. Displacement for Different Temperatures")
plt.legend()
plt.grid(True)

# Show plot
plt.show()


# %%
print(energy_stats_df[["displacement", "T", "energy_std",
↪  "tau_int_energy", "error_energy"]])


# %%
print(energy_stats_df[["displacement", "T", "N_eff_energy",
↪  "error_energy"]])


# %%
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf

L_check = 130  # Pick a displacement
T_check = 300  # Pick a temperature

# Extract energy and force timeseries
force_series = df[(df["displacement"] == L_check) & (df["T"] ==
↪  T_check)]["force"]
energy_series = df[(df["displacement"] == L_check) & (df["T"] ==
↪  T_check)]["energy"]

# Compute autocorrelations
acf_force = acf(force_series, nlags=500, fft=True)
acf_energy = acf(energy_series, nlags=500, fft=True)

# Plot
plt.figure(figsize=(8, 5))
plt.plot(acf_force, label="Force ACF", color="tab:blue")
```

```
plt.plot(acf_energy, label="Energy ACF", color="tab:orange")
plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.title(f"Autocorrelation Comparison at L={L_check}Å, T={T_check}K")
plt.legend()
plt.show()


# %%
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf

L_check = 130  # Pick a displacement
temperatures = [300, 600]  # Analyze for both temperatures

# Create subplots for comparison
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

for i, T_check in enumerate(temperatures):
    # Extract energy and force timeseries
    force_series = df[(df["displacement"] == L_check) & (df["T"] ==
    ↪  T_check)]["force"]
    energy_series = df[(df["displacement"] == L_check) & (df["T"] ==
    ↪  T_check)]["energy"]

    # Compute autocorrelations
    acf_force = acf(force_series, nlags=500, fft=True)
    acf_energy = acf(energy_series, nlags=500, fft=True)

    # Plot
    ax = axes[i]
    ax.plot(acf_force, label="Force ACF", color="tab:blue")
    ax.plot(acf_energy, label="Energy ACF", color="tab:orange")
    ax.set_xlabel("Lag")
    ax.set_title(f"Autocorrelation at L={L_check}Å, T={T_check}K")
    ax.legend()

# Set common Y-axis label
axes[0].set_ylabel("Autocorrelation")
```

```
# Adjust layout and show the plot
plt.tight_layout()
plt.show()


# %%
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf

# Select a single temperature (300K)
selected_temp = 600

# Define displacement values
displacements = sorted(df[df["T"] ==
↪  selected_temp]["displacement"].unique())

# Create a plot for energy ACF across all displacements at 300K
plt.figure(figsize=(10, 6))

for L in displacements:
    subset = df[(df["displacement"] == L) & (df["T"] ==
    ↪  selected_temp)]["energy"]
    acf_values = acf(subset, nlags=500, fft=True)

    plt.plot(acf_values, label=f"L = {L}Å")

plt.xlabel("Lag")
plt.ylabel("Autocorrelation")
plt.title(f"Energy Autocorrelation Function at {selected_temp}K for
↪  Different Displacements")
plt.legend()
plt.show()
```

6.4 LAMMPS Input File - Simulation Setup

LAMMPS Input Script

```
# length between chain ends
# (Removed the below line so L is set only by -var)
# variable        L equal ${L}

# temperature in K (set dynamically by the shell script)
variable        temp index ${temp}

# number of steps over which to stretch polymer from the initial state to
↪  length L
variable        Nstretch equal 1e5

# number of extra steps before data collection is started (25000)
variable        Nequilib equal 25000

# number of steps during which data is collected
variable        Nrun equal 1e6

# number of steps between force sample collection
variable        tsamp equal 10

# number of steps between dumping coordinates
variable        tdump equal 1000

# Define a folder name that includes metadata (chain length, temperature,
↪  etc.)
variable folder string
↪  L${L}_T${temp}_Ns${Nstretch}_Neq${Nequilib}_Nrun${Nrun}_ts${tsamp}_td${tdump}

# Create the directory (requires that the shell command is available in
↪  your LAMMPS Docker image)
shell mkdir -p ${folder}



#####################################
# Do not change settings below this
```

```
#####################################

# number of monomers in chain (needs to match lmp.data file)
variable        Nmonomers equal 50

# identity of endpoint atoms
variable        head_id equal 2
variable        tail_id equal 5+(v_Nmonomers-1)*13


units           metal
atom_style      full
boundary        f f f
timestep        0.001

read_data       lmp.data
include         lmp.param

log             ${folder}/lmp${L}.log
thermo          ${tsamp}
thermo_style    custom step temp pe
dump            write_traj all xyz ${tdump} ${folder}/lmp${L}.xyz
dump_modify     write_traj sort id element C C H

group           head_atom id ${head_id}
group           tail_atom id ${tail_id}

#-----------------------------------------------#
#           stretching to target L              #
#-----------------------------------------------#

variable        xh  equal x[${head_id}]
variable        yh  equal y[${head_id}]
variable        zh  equal z[${head_id}]
variable        xt  equal x[${tail_id}]
variable        yt  equal y[${tail_id}]
variable        zt  equal z[${tail_id}]
variable        vxh equal ((v_xt-v_xh)-v_L)/(2*v_Nstretch*0.001)
variable        vyh equal (v_yt-v_yh)/(2*v_Nstretch*0.001)
```

```
variable        vzh equal (v_zt-v_zh)/(2*v_Nstretch*0.001)
variable        vxt equal -v_vxh
variable        vyt equal -v_vyh
variable        vzt equal -v_vzh

fix             run_traj all nvt temp ${temp} ${temp} 0.1
fix             move_head head_atom move linear ${vxh} ${vyh} ${vzh}
fix             move_tail tail_atom move linear ${vxt} ${vyt} ${vzt}
run             ${Nstretch}
unfix           move_head
unfix           move_tail

print           "head atom coordinates: ${xh} ${yh} ${zh}"
print           "tail atom coordinates: ${xt} ${yt} ${zt}"

#-----------------------------------------------#
#            equilibrating system               #
#-----------------------------------------------#

set             atom ${head_id} vx 0.0 vy 0.0 vz 0.0
set             atom ${tail_id} vx 0.0 vy 0.0 vz 0.0

fix             store_frcs all store/force
fix             fix_head head_atom setforce 0.0 0.0 0.0
fix             fix_tail tail_atom setforce 0.0 0.0 0.0
run             ${Nequilib}

#-----------------------------------------------#
#               production run                  #
#-----------------------------------------------#

variable        fxh equal f_store_frcs[${head_id}][1]
variable        fyh equal f_store_frcs[${head_id}][2]
variable        fzh equal f_store_frcs[${head_id}][3]
variable        fxt equal f_store_frcs[${tail_id}][1]
variable        fyt equal f_store_frcs[${tail_id}][2]
variable        fzt equal f_store_frcs[${tail_id}][3]

log             ${folder}/lmp${L}.log
```

```
fix             log_frcs all print ${tsamp} "${fxh} ${fyh} ${fzh} ${fxt}
↪  ${fyt} ${fzt} ${xh} ${xt}" file ${folder}/lmp${L}.forces screen no
run             ${Nrun}
```

6.5 Shell Script - Running LAMMPS Simulation

Shell Script for Running LAMMPS

```
#!/bin/bash

# Loop through both temperature values (300K and 600K)
for T in 300 600; do
    for L in 130 140 150 160 170 180 190 200; do
        echo "Starting LAMMPS simulation for L=$L at T=$T..."

        docker run --rm \
            -v "$(pwd)":/data \
            -w /data \
            -e OMP_NUM_THREADS=2 \
            lammps/lammps:latest \
            mpirun -np 2 lmp_mpi -in lmp_temp_L.input -var L $L -var temp
            ↪  $T

        echo "Completed simulation for L=$L at T=$T."
    done
done
```