Uppsala University

Bachelor's program in Physics

Degree project C in Physics

Towards a better understanding of protein structures – assessing the sulfur bridge in Cystine through photofragmentation

Author: Emma Danielsson
Supervisors: Oscar Grånäs and Carl Caleman
Subject reader: Mattias Klintenberg

July 20, 2020

Abstract

This work aims to investigate the fragmentation of an ionized Cystine molecule, as simulated in the framework of molecular dynamics and quantum mechanics. Cystine is viewed as a model system for larger sets of peptides – ultimately contributing to the understanding of protein photofragmentation, which is crucial for determining the structure of a protein using new methods. The analysis software was written in Python, partly in conjunction with another student. The photofragmentation of the molecule is analyzed in terms of bond integrity versus time and mass-to-charge ratios for the resulting fragments. Generally, the molecule disintegrates into more and smaller fragments the higher the degree of ionization is.

Sammanfattning

I det föreliggande arbetet undersöks fragmenteringen av en joniserad molekyl Cystin, som simulerats medelst molekyldynamik och kvantmekanik. Cystin betraktas som ett modellsystem för större peptidstrukturer – något som i längden kan bidra till större förståelse för fotofragmentering av proteiner, vilket i sin tur är avgörande inom nya metoder för strukturbestämning. Analysprogrammet skrevs i Python och delvis i samarbete med en annan student. Molekylens fotofragmentering analyseras med avseende på bindningsintegritet över tid, samt mass-laddningskvot hos de resulterande fragmenten. I allmänhet sönderfaller molekylen till fler och mindre fragment ju högre joniseringsnivån är.

Contents

1 Introduction

Proteins serve a vast array of purposes in living organisms. For instance, they act as catalysts in many biochemical reactions that regulate the metabolism of a cell – to the extent that the basic processes of life wouldn't be feasible without them. The biochemical function of a protein is determined both by the sequence of amino acids that comprises it and the three-dimensional structure that it folds into. Thus, structure determination at atomic resolution has been a major subject of interest for researchers [1]. The most common method that researchers have used historically has been X-ray crystallography, a method which requires the protein to be in crystalline form. This way the energy of the X-ray is absorbed throughout the crystal, allowing for a comparatively long exposure time and a clear diffraction pattern. One limitation of this method is that many types of proteins are difficult to consolidate into large crystals. Since a few years back, though, a new type of radiation source has emerged – the X-ray free electron laser, XFEL.

XFEL uses intense, femtosecond-length pulses of X-rays to gather data from a sample. Due to the high energy content of the pulses, the molecule being studied will disintegrate into a plasma during the X-ray exposure [2]. Calculating the structure of the molecule after such a measurement is only possible if one has some knowledge of how its fragmentation tends to happen.

1.1 Aim

This project will use data from simulations of the dipeptide Cystine to analyse its process of fragmentation due to ionization. In particular, focus will lie on the behaviour of its disulfide bond. Integrity of individual bonds will be displayed in heat maps, while the mass-to-charge ratio of each fragment at the end of the simulation will be displayed in histograms.

2 Background

2.1 Determination of protein structure

In all DNA- and RNA-based forms of life, proteins can be said to constitute the basic biochemical tools of the cell. They fill a variety of functions – for example information transfer, transport and catalysis of chemical reactions. Despite of this large variation, most biologically relevant proteins consist of the same 20 amino acids connected in different sequences. The exact biochemical function of a protein cannot be identified without knowing its three-dimensional structure. [1]

The first experiments to determine the three-dimensional structure of a protein were carried out with the help of X-rays in the beginning of the twentieth century. They led to Max von Laue being awarded the Nobel Prize in Physics in 1914, and were the beginnings of a research field called X-ray crystallography. [3] The basic principle of X-ray crystallography is to shine X-rays through the sample, which should be in the form of a fairly large and stable crystal, and to detect the resulting diffraction pattern. The diffuse energy absorption by the crystal is what allows the beam to continue for some period of time without the crystalline structure disintegrating. After the necessary time of exposure, the structure of the individual molecules in the crystal can be calculated. [4]

Many biologically relevant proteins are difficult to consolidate into large enough crystals to be imaged with X-ray crystallography, and have thus been out of reach for structure determination. Towards the end of the 2000's, a new type of radiation source – the X-ray free electron laser (XFEL) – would come to change this. XFEL structure determination is based on the idea of collecting a lot of diffraction data during a very short period of time. The central challenge is to collect enough data before the molecule disintegrates into a plasma. The first facility to achieve atomic-scale resolution with this technique was LCLS at Stanford University, USA. [2] Although the usual timescale of the pulses in XFEL is in the range of femtoseconds, this is not enough to approximate the examined molecule as static during the pulse. On the contrary, the sample usually starts to decompose before the pulse has ended. This means that the resulting diffraction data contains information about several stages of disintegration, not just the molecule in its intact state. To be able to interpret results from this kind of structure determination, then, one needs information about how the molecule of interest disintegrates due to radiation. [5] [6]

There exists a number of different methods for simulation of biomolecules. Two examples of software are GROMACS and Siesta, which are based on molecular dynamics and quantum mechanics respectively. Siesta uses a variety of underlying quantum mechanical models to simulate the dynamics of the molecular system. Several parameters, like ionisation, spatial orientation and the surrounding medium can be varied. To create a comprehensive picture of the fragmentation process, several combinations of these parameters can be run. [7]

The number of amino acids that comprise a single protein varies between 51 and around 34000. [1] With the soft- and hardware available today, simulating these systems at a quantum level is generally not feasible. For that reason, researchers often choose to simulate a so-called model system – shorter peptides or single amino acids. Despite of their limited scope, the results from these simulations can be used to better understand the compound system.

2.2 Cystine

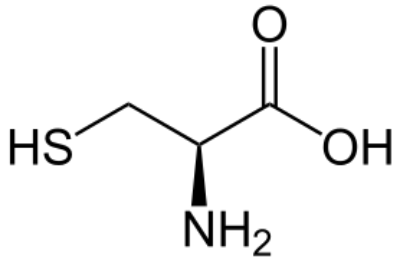


Figure 1: Skeletal formula of Cysteine [8]

Within the scope of this project, the fragmentation of the dipeptide Cystine will be examined. Cystine is the result of a condensation reaction between two molecules of Cysteine, a hydrophobic amino acid which is commonly occurring in proteins. Like all amino acids, Cysteine has one carboxyl and one amine group. In Figure 1, they are shown at the top right and the bottom respectively. These groups can form peptide bonds with other amino acids, thus locking the molecule into a peptide sequence which could form a protein. The side chain of Cysteine, shown to the left in Fig. 1, consists of a thiol group – that is, one hydrogen and one sulfur atom. It is using this side chain that two Cysteine molecules can react to form Cystine.

Two Cysteine molecules in different parts of a protein can react to create a Cystine molecule. This is of great significance for the structure of the protein, since the sulfur bond interlocks two parts of the peptide chain. Its bonding strength is higher than that of van der Waals-interactions but somewhat lower than in the covalent C-C bonds that make up the backbone of the peptide chain. The sulfur bond occurs particularly frequently in keratin, a class of proteins that, among other things, make up hair, fur, claws, beaks, scales and skin. The different variants of keratin have many different tertiary structures, but generally the harder types contain more sulfur bonds. [1] [9]

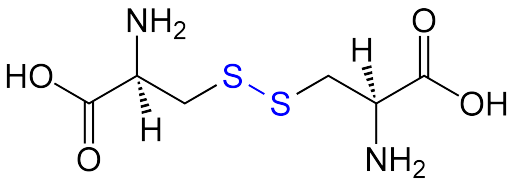


Figure 2: Skeletal formula of Cystine with Natta projection. The sulfur bond is marked in blue. [10]

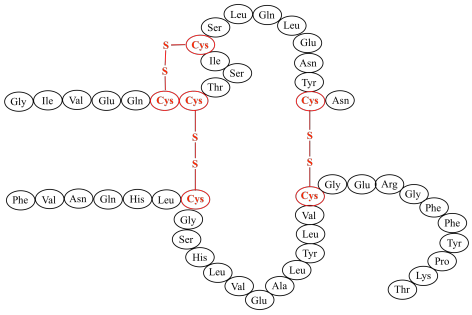


Figure 3: A schematic represenation of insulin, the smallest known protein, with the Cystine sulfur bridges marked in orange. [11]

3 Method

The analysis of simulation data will be carried out in the programming language Python, in particular using the Jupyter Notebook platform. For some parts of the software construction, I collaborated with another bachelor student, Ebba Koerfer, who does a similar project. When writing the code, our overarching goal was to make it as general and transferable as possible. The quantum mechanical simulations analyzed in the project were run by Oscar Grånäs who also provided a set of functions for data processing – namely, the script `analyze_trajectories.py`. During the course of the project, Ebba and I wrote and added the functions `bond_broken_2`, `mean_distance_dict` and `frags_from_dists` to this file. An overview of the analysis procedure is given below – for details about the structure and function of the code, please refer to the appendix.

The project can be divided into two main parts – the pre-processing of thermalization data to investigate how neighboring atoms bond under usual conditions, and the analysis of how these bonds develop once a high level of ionization (like the one resulting from radiation) is applied. The first task is to import the necessary data from the thermalization runs into the script, and to package it into a useful format. This is done with the help of the function `parse_timestep` from the script `analyze_trajectories.py`, which extracts the position of each atom in the molecule for each time step in one simulation. The process is repeated for every thermalization, and the resulting data is put into a list. To be able to investigate the bonds of the molecule from this data, it is essential to not confuse the atoms for one another. Therefore, the atoms were assigned both an index, indicating their exact position in the molecule, and a label indicating the type of element. For example, one of the nitrogen atoms was assigned the index 0 and the label N1. This was also accomplished with a function from `analyze_trajectories.py`.

To facilitate the analysis of bonds in the molecule, a list of "neighbors" was created. For each atom in the molecule, this overarching list contains one list of the atoms that would be considered bonded to it. The list is constructed using the function `get_neighborlist`, which returns all atoms within a certain radius of a given atom at a specific time step of the simulation. Thus, it is a crude measure of what atoms can be said to be bonded to each other.

With these structures done, it was time to determine the *mean* distances between the supposedly bonded atoms in each thermalization run. For this purpose the function `mean_distance_dict` was written. Through a series of loops, it constructs a dictionary with the names of the bonded atoms as a key, and a list of the mean distance between them in all thermalization runs as the value.

The next part of the code aims to analyse the bond integrities over time in a highly ionized molecule of Cystine. Bond integrity is defined as follows:

$$\mathscr{B}_I(A,B,t) = \frac{1}{N_{MD}} \sum_{i=1}^{N_{MD}} \left(1 + e^{\lambda(|d_i[A,B](t) - d_i[A,B](0)| - 0.5)}\right)^{-1} \quad [6]$$

In the above expression, $d_i[A,B](t)$ is the distance between the atoms at time t, λ is a smearing parameter and N_{MD} is the number of molecular dynamics simulations. In the script, the equation was implemented in the function `bond_broken_2` with the slight difference that no averaging over different runs is made – $N_{MD} = 1$. Rather, data on one bond from its ionized simulation and

from the corresponding thermalization yields one value of $\mathscr{B}_I$ for each time step in the simulation. New files, based on simulations with a variety of initial ionizations and configurations, are provided at this stage. Data regarding positions of the atoms are extracted from these files and loaded into a set of lists. From these, bond integrity for each bond in each configuration could be calculated using the function `bond_broken_2` . Plots of bond integrity versus time were constructed for each bond, averaged over all ionization levels and starting geometries.

Since one aim of the project is to determine the fragmentation of the molecule it was necessary to determine if, when and under what conditions the bonds in the molecule were broken. Being able to calculate bond integrity from distance between two atoms, this was fairly straightforward. By studying how the value of the bond integrity oscillated for different bonds in the thermalization runs, we could establish a "stable range" which would encompass all normal oscillations. The limit for when a bond would be considered broken was then set well outside of this range – we chose $B_I = 0.5$. To clarify: since the nature of a chemical bond is continuous rather than discrete, this limit is somewhat arbitrary and only to be taken as an approximation for when the bond is to be considered broken. Together with the previously loaded information about distances between bonded atoms, this limit was fed into the function `frags_from_dists`. It returns the state of the molecule at the last timestep of each simulation. Each run is represented by a set of lists that display the atoms present in a particular fragment. In the cases where no fragmentation occurs, there will simply be one fragment containing all of the atoms.

Having calculated what fragments are formed in each run of the simulation, some analysis of the mass-to-charge ratio would also need to be performed. For this purpose the script `ElementData`, containing information about the mass of each element, was used. The charges on each atom for each timestep in the simulation were obtained using the function `get_hirsh`. Information from the two sources were combined using simple division, and displayed in histograms. To get a plot more similar to what would arise from experiment, a kernel density plot was also constructed using the seaborn library. A gaussian kernel was used, and the smearing parameter bw was set to 0.125.

4 Results

4.1 Molecular structure

In the present work, the atoms of the Cystine molecule have been assigned labels according to their species and location in the molecule. Of course, Cystine is symmetric around the central sulfur bridge, so the choices are somewhat arbitrary. The molecular structure, together with the chosen labels is demonstrated in the illustration below:

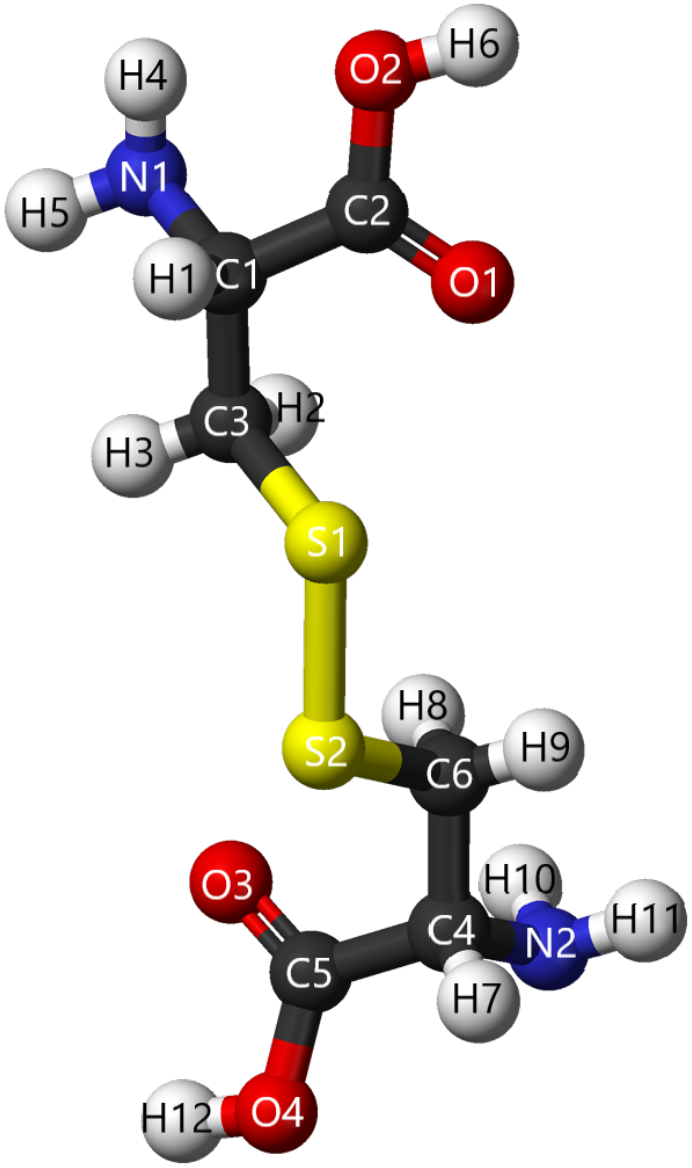


Figure 4: A ball-and-stick model of Cystine with the labels assigned to each atom in the present work written out. [12]

4.2 Bond integrity

As mentioned in the Method section, the integrity of each bond over the course of the fragmentation was to be calculated. The results are displayed in heatmaps, where the y-axis shows time and the x-axis shows the value of $\bar{z} = {}^{e}/_{N}$. The bond integrity data, displayed with a certain color at a point (x,y), is compiled from all of the starting geometries. Blue color indicates a value of $\mathscr{B}_I$ close to 0, while yellow indicates $\mathscr{B}_I \approx 1$.

The following are heatmap plots of a few bonds that occur in Cystine - for brevity, the rest are displayed in the Appendix. Figures 5-7 occur in and around the central sulfur bridge, while the rest are examples of the C-C, N-H and C-H bonds that occur elsewhere in the molecule.

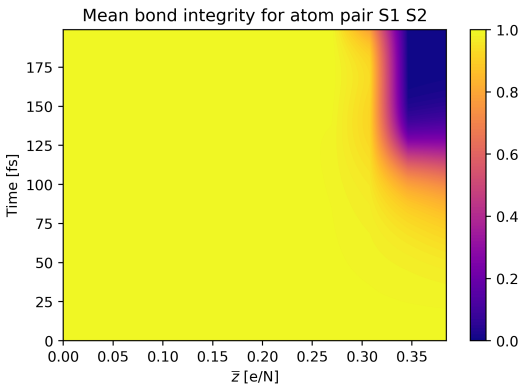


Figure 5

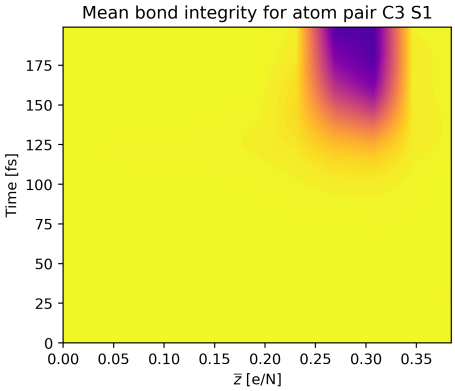


Figure 6

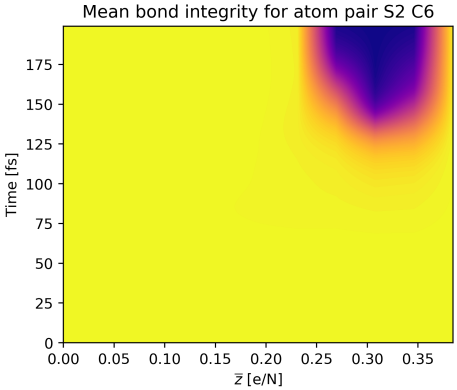


Figure 7

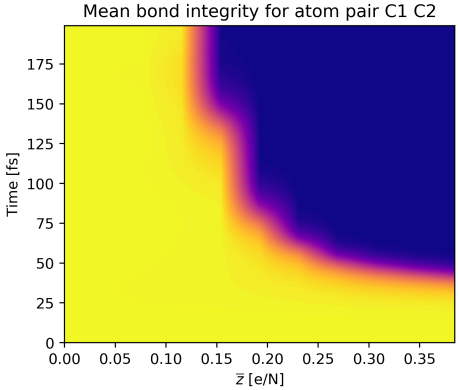


Figure 8

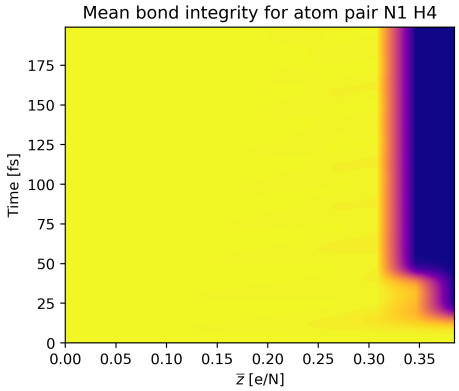


Figure 9

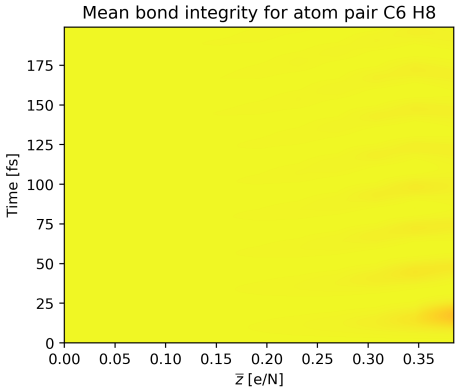


Figure 10

4.3 Mass-to-charge ratio of fragments

The plots in this section display mass-to-charge ratios of the fragments that form in each ionization level of the simulation. Data from all geometric configurations is shown in each plot.

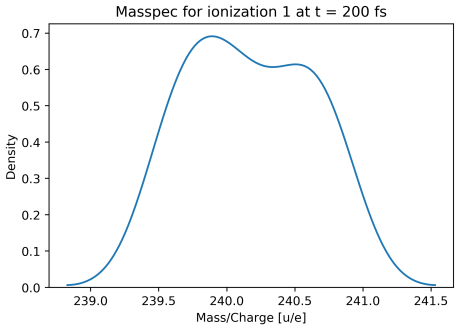


Figure 11

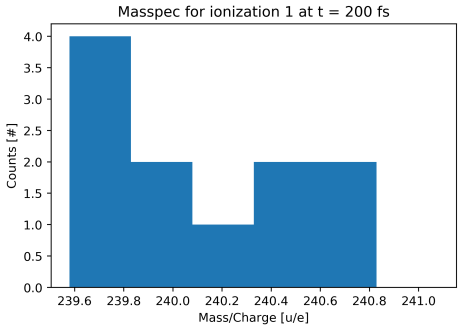


Figure 12

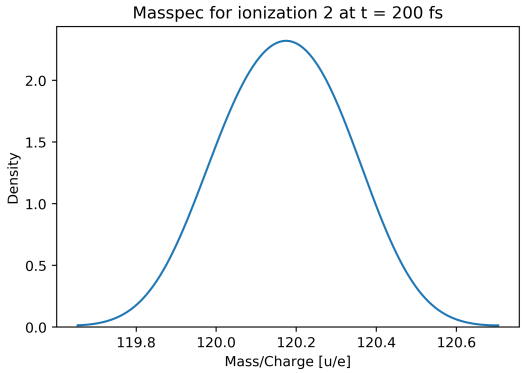


Figure 13

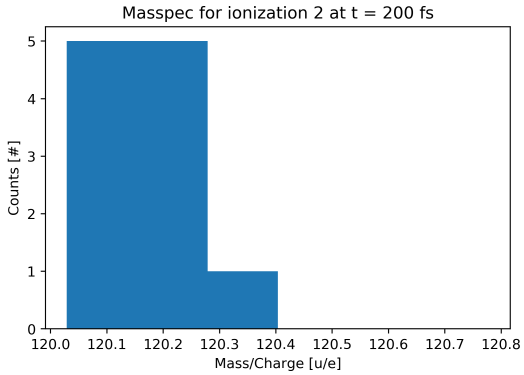


Figure 14

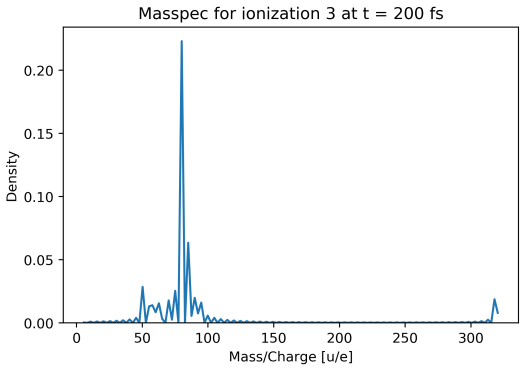


Figure 15

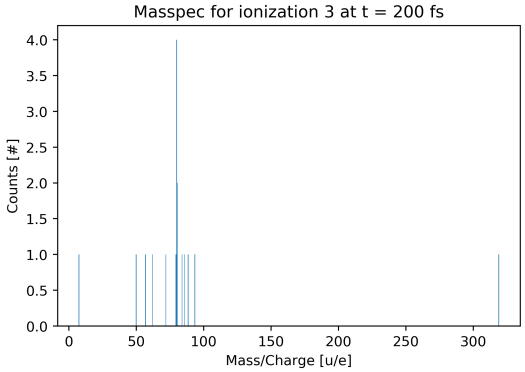


Figure 16

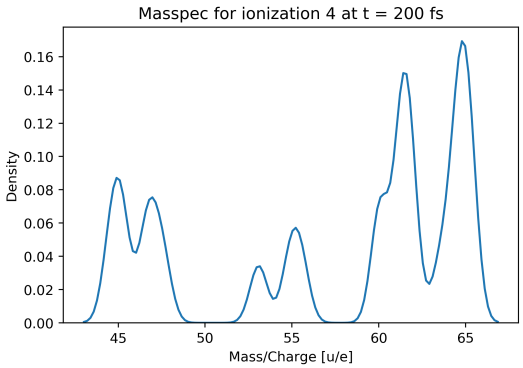


Figure 17

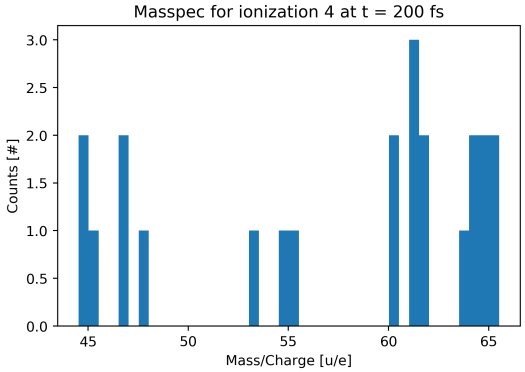


Figure 18

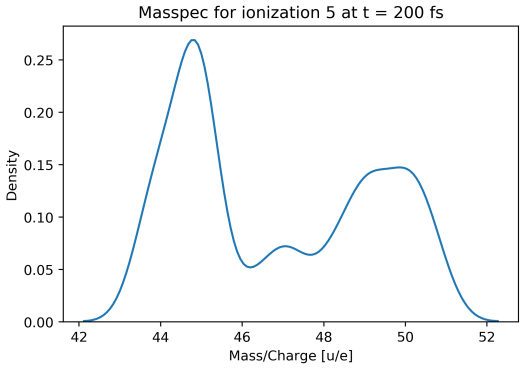


Figure 19

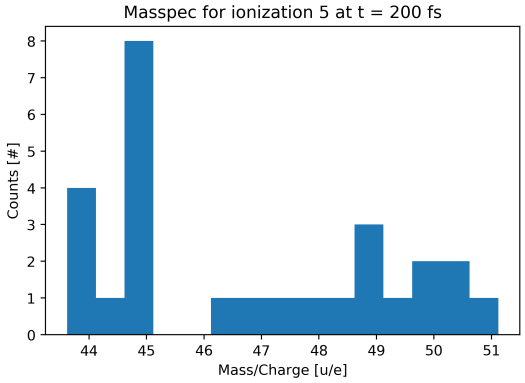


Figure 20

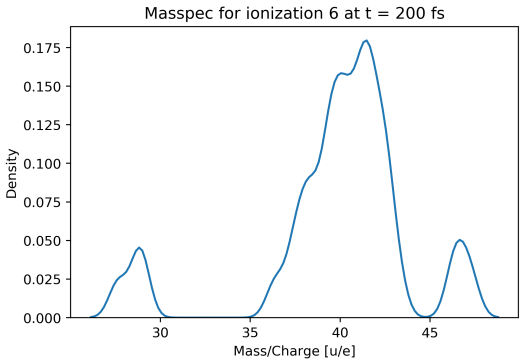


Figure 21

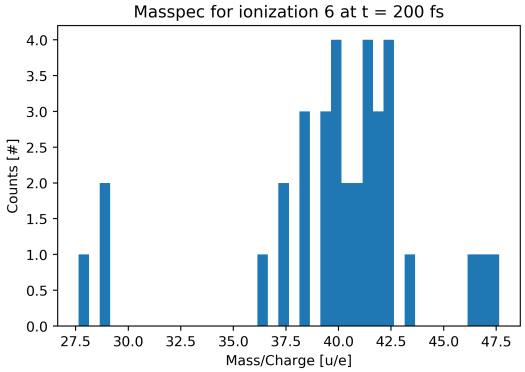


Figure 22

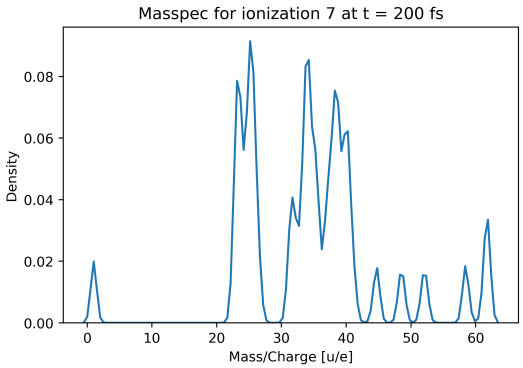


Figure 23

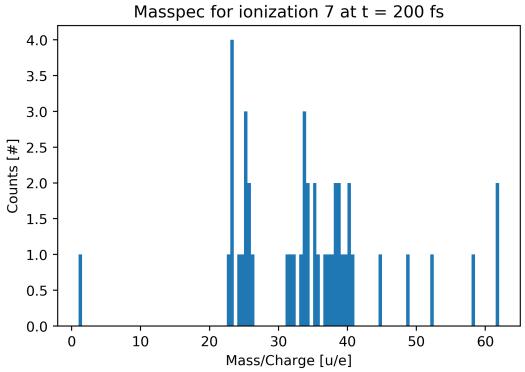


Figure 24

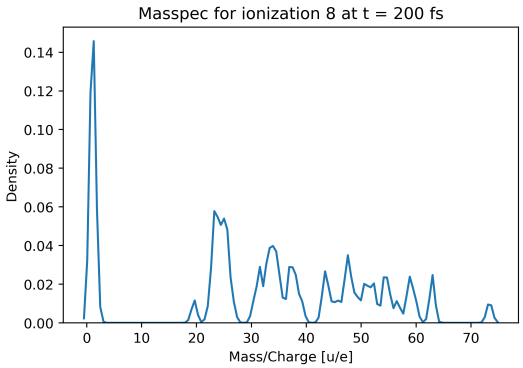
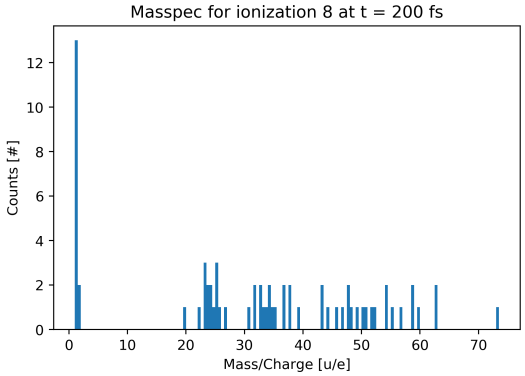
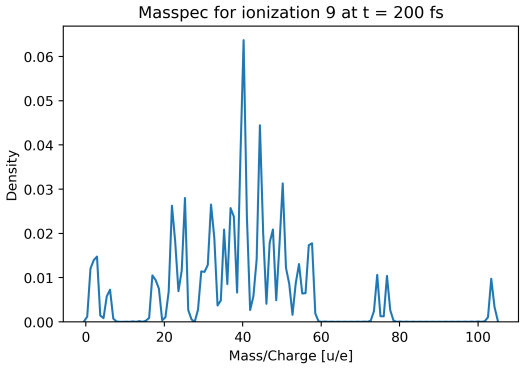


Figure 25



(a) Figure 26



(a) Figure 27

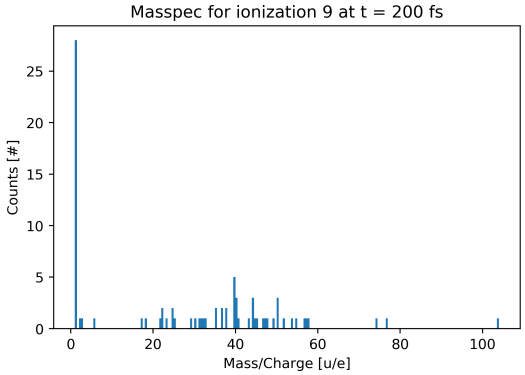


Figure 28

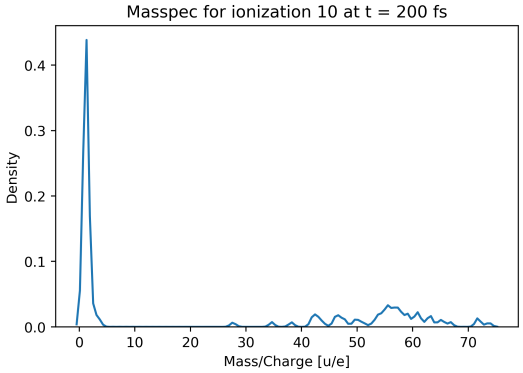


Figure 29

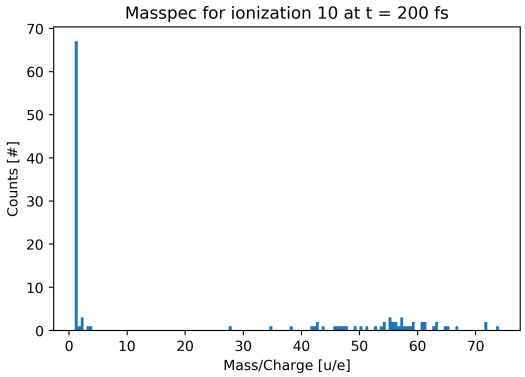


Figure 30

5 Discussion

As shown in the results, the fragmentation process generally follows a pattern: the more ionization occurs in the molecule, the less stable it becomes and the more fragments we can observe at the end of the simulation. Apart from this, though, there are many aspects that warrant discussion.

For example, from the heatmaps (Fig. 8, 36) we can see that the two bonds in the molecule most prone to breaking are both between carbon atoms that form the backbone of the molecule's structure. For low ionization levels, they tend to stay stable for the whole simulation, but between $\bar{z} = 0.10$ and $\bar{z} = 0.15$ both bonds start disintegrating before the end of the simulation. I initially found these results quite surprising, since there are plenty of bonds in Cystine with a longer average bonding distance – like the S-S bond. Though, the sensitivity of the C-C bond is consistent with earlier results. [6]

Another interesting feature of the results is the behaviour of the C-H bonds (Fig. 10, 44-48). At higher ionization levels, they tend to oscillate in bond integrity. Judging by the colour scale, the bond integrity seems to stay in the range $[0.8, 1]$ for all of them, which is well above the limit that was set for bond breaking, $BI = 0.5$. We can see a somewhat similar, though not as pronounced, behaviour for some of the other bonds (for example N1,H4).

As stated earlier, the general trend in the fragmentation of the molecule is that higher ionizations yield a larger amount of small fragments. This is apparent in most consecutive plots of mass-to-charge ratio, even though their appearance is different due to the bin sizes of the histograms. Though, there is one detail in the mass-to-charge plot for ionization level 3 (Fig. 15,16) that seems discrepant – the rightmost bin in the histogram is placed at $^u/_e \approx 310$. Since the total weight of the Cystine molecule is around $u = 240$, this must have been caused by an error in the code. Despite quite some searching and checking the script, I have been unable to locate it. The conclusion must thus be that this unknown error in the code might have affected the other mass-to-charge plots as well.

One central objective of this project was to study the S-S bond, which proved somewhat difficult during code construction. More specifically, the long bonding length between the sulfur atoms (between 2.0 and 2.2 Å) posed a hurdle when trying to assign all atoms their correct "neighbors". Our initial attempt at trying to set a firm distance limit beyond which atoms would not be bonded to each other didn't work at all – if the limit was set high enough to include the sulfur bond, an extra 15-20 bonds would also be included. Even after thorough pruning, the high distance limit would cause illegitimate bonds to be included. The solution, though rather crude, was to simply enter the sulfur bond manually into the list of neighboring atoms.

Once the sulfur bond was included in the data set, though, its behaviour could be visualised in a heat map (Fig. 5). Comparing it to the heat maps for the sulfur atoms' respective bonds to carbon atoms (Fig. 6, 7) revealed some interesting patterns. Firstly, the S-S bond seems to start breaking up around $t = 110$ for ionisation levels above $\bar{z} = 0.30$, while the C-S bonds generally dissolve later in the simulation and for lower ionisation levels. Though there is some difference between the overall volatility of the two, both bonds will start disintegrating at some point between $\bar{z} = 0.20$ and $\bar{z} = 0.25$. This will usually happen around the time $t = 130$, but it varies with ionisation level. From these trends we can draw some conclusions about the fragmentation of the molecule.

For ionization levels between $\bar{z} \approx 0.20$ and $\bar{z} = 0.30$ the C-S bonds break while the S-S bond stays intact, which implies that the two sulfur atoms form a fragment. When $0.30 < \bar{z} < 0.35$ all of the three bonds tend to break, which results in the two sulfur atoms dropping off of the molecule independently. Interestingly, for $\bar{z} > 0.35$, we see an increasing bond integrity for the C-S bonds but no such behaviour in the S-S bonds – which would imply that the molecule breaks right down the middle. Though, due to the slight difference between the two C-S bonds there also exists "mixed" cases.

As stated in the Method section, the goal when writing the code for this project has been to make it general and transferable so it can be used in subsequent projects. One example of this is the function `bond_broken_2`, which calculates bond integrity. It takes the mean value and standard deviation of the distance between two atoms from a thermalization run as parameters and uses them as a baseline for the ionized case – instead of setting an arbitrary limit for all bonds. This way, bond integrity is automatically calculated differently for each bond.
One underlying source of error in this project is the pre-written code that was provided – specifically, the functions in `analyze_trajectories.py`. Due to my own lack of experience in loading and processing molecular simulations, understanding how these functions worked proved quite a challenge. Even though several of them are used in the final script, and I understand the in- and outgoing values, a lot of their contents remain a "black box" to me. Not only does this mean that systematic errors might be introduced to the code, but also that I have no way of estimating them. Though I understand the necessity of using provided functions and programs, I find this somewhat troubling from a scientific point of view – because it means I can't fully guarantee the quality of the calculations.

Similarly, I have very little insight into the Siesta simulations from which the data originates. Although I trust my supervisor to have made reasonable assumptions when setting up the system, this may also be a source of systematic error in the project. Since I have very little experience with the theoretical groundwork of Siesta, though, that is as concrete a conclusion I can draw at this point. Beyond that there is only speculation.

Outlook

To deepen the understanding of Cystine photofragmentation, I have some suggestions for future works. First and foremost: to run simulations width a larger amount of starting configurations and ionization levels. As could be seen in the Results section the fragmentation process is highly dependent on both factors, which is a compelling reason to collect better statistics. I would not necessarily expand the range of ionization levels – since the experimentally probable levels lie well within it – but rather increase its density. Of course, running more configurations will be more expensive in terms of time and resources, but it would give a fuller view of the fragmentation process.

My second suggestion to improve upon the present work would be to simulate Cystine in an aqueous environment instead of vacuum. Although more cumbersome to simulate, this would represent the molecule's usual biochemical environment much better.

Conclusions

Since I couldn't analyze the sources of error in the simulations and the provided functions, the reliability of my results can't really be evaluated. Though, some patterns – like the ionization level at which most bonds start disintegrating – are in line with earlier results. In the context of other works on the subject, the general trends of the fragmentation process weren't anomalous either.

6 Appendix

Bond integrity plots

The plots of bond integrity versus time that were not included in the Results are displayed here.

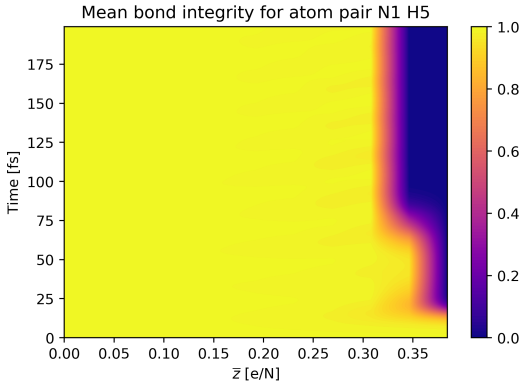


Figure 31

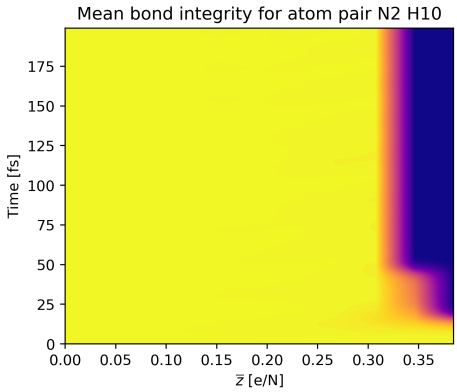


Figure 32

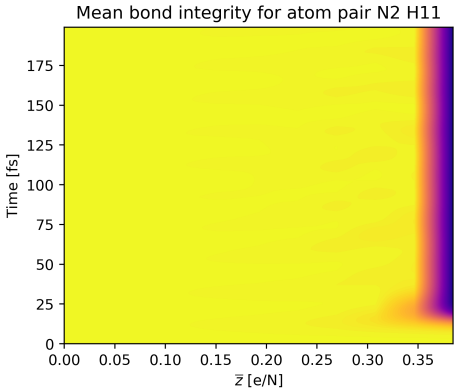


Figure 33

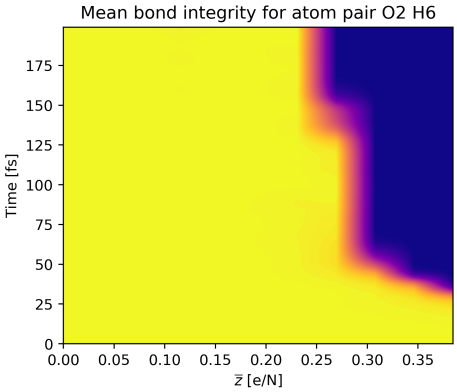


Figure 34

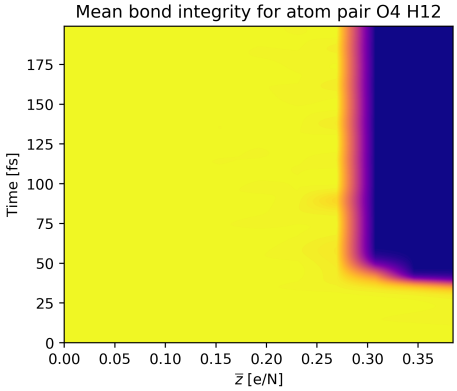


Figure 35

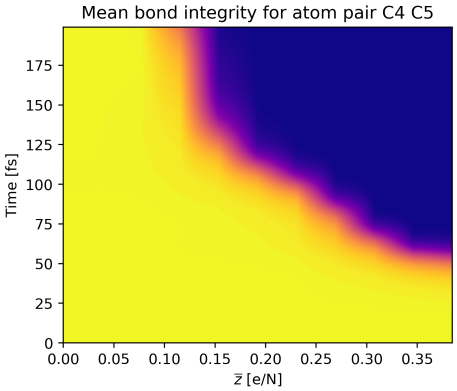


Figure 36

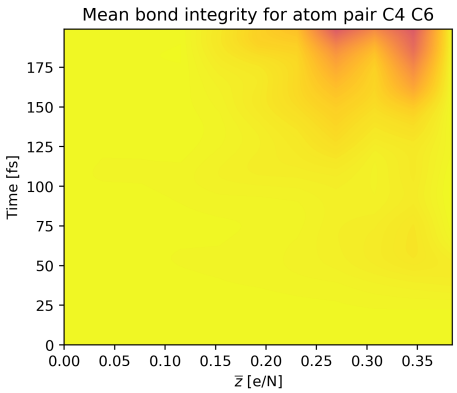


Figure 37

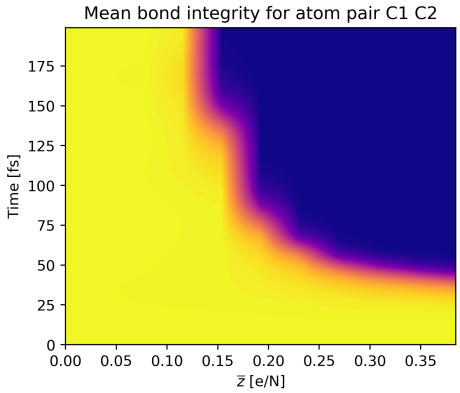


Figure 38

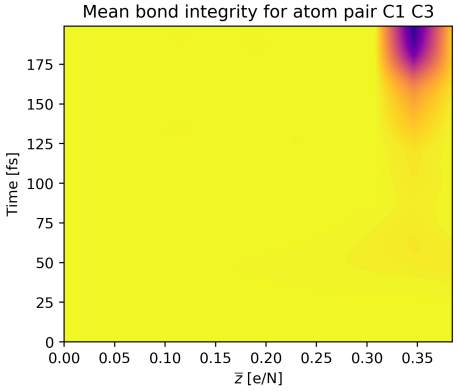


Figure 39

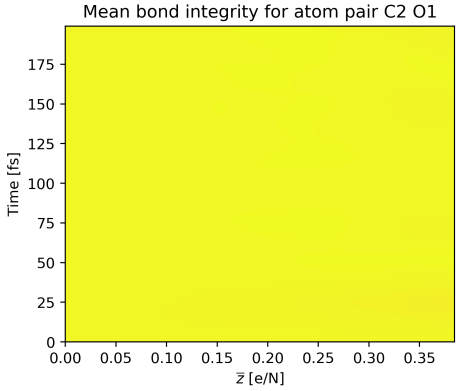


Figure 40

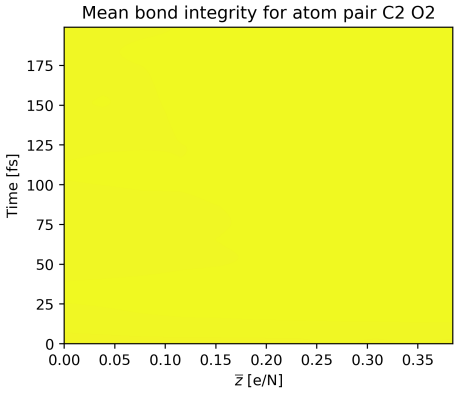


Figure 41

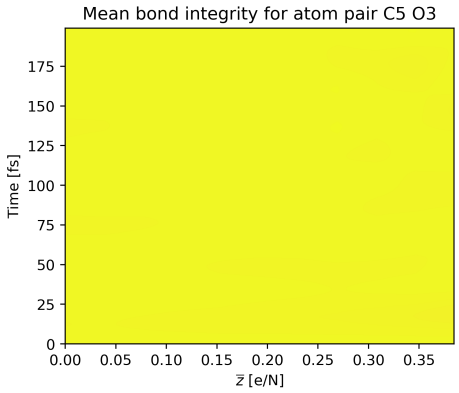


Figure 42

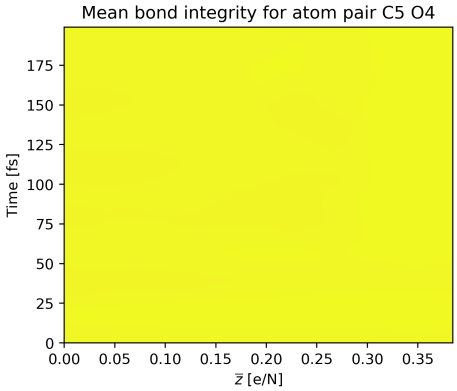


Figure 43

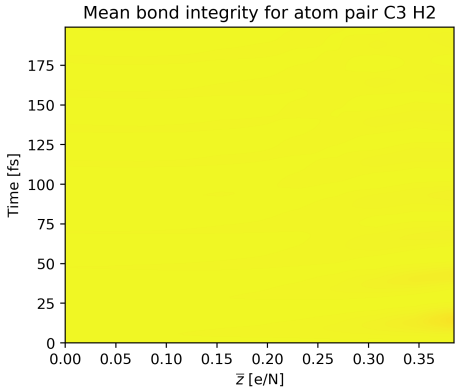


Figure 44

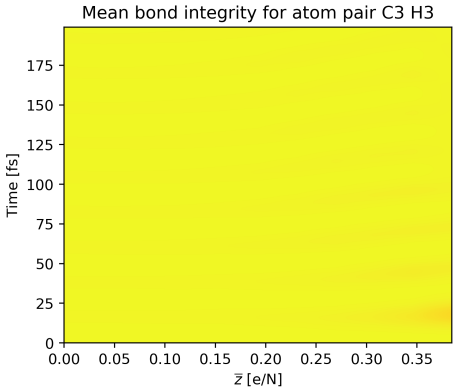


Figure 45

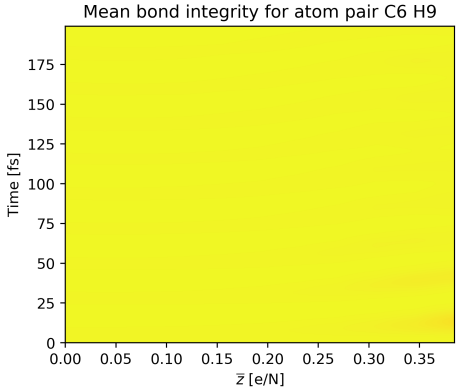


Figure 46

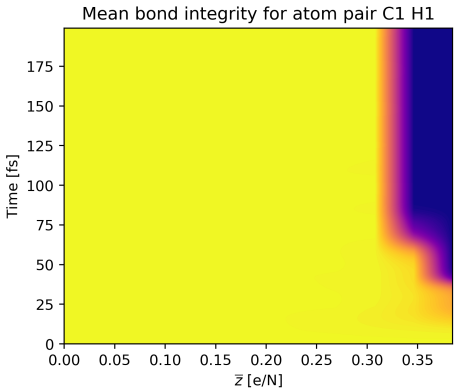


Figure 47

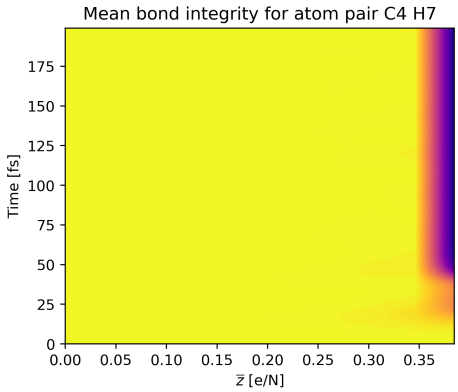


Figure 48

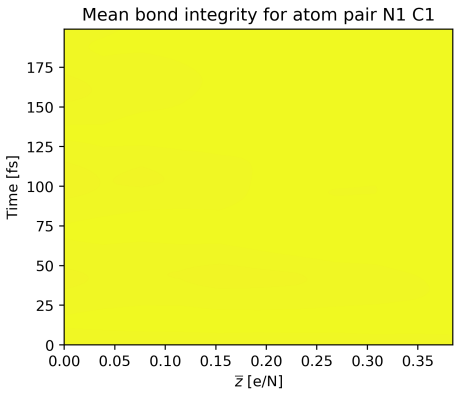


Figure 49

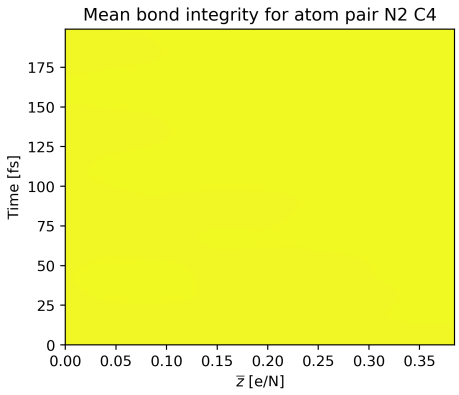


Figure 50

loadv6.ipynb

The following is the code contents of the IPython Notebook file `loadv6.ipynb`. Some basic commands were provided by Oscar at the beginning of the project, but the majority of the code has been constructed by Ebba Koerfer and me.

```
import numpy as np
import scipy as sp
from statistics import mean, stdev
from analyze_trajectories import *
import matplotlib.pyplot as plt
from elementdata import *
import seaborn as sns

#The names of the data files must be entered explicitly
runs=['thermalize_short_0.out','thermalize_short_1.out','thermalize_short_2.out','thermalize_short_3.out',
      'thermalize_short_4.out','thermalize_short_5.out','thermalize_short_6.out','thermalize_short_7.out',
      'thermalize_short_8.out','thermalize_short_9.out','thermalize_short_10.out']

#The data is extracted and added to thermalisation_list
thermalization_list=[]
for run in runs:
    time_pos, timeserie, orblegend, specieslegend, numberlegend = parse_timestep(run)
    thermalization_list.append(time_pos)

index_to_atom, atom_to_index=make_atom_dictionary_from_timeserie(time_pos)

#A first, crude estimate of the bonding is made
neighbors_list = get_neighborlist(time_pos[80],1.8)
neighbors_list[7].extend([20])
neighbors_list[20].extend([18,7])

print(f'Neighbor list[k][j]: {neighbors_list}')

#Based on the estimate made above, the distances between the "bonded" atoms for each time step in the simulations
# is calculated. The mean bonding distance is also calculated for each bond.
mean_distances_dict, distance_list = mean_distance_dict(thermalization_list, index_to_atom, neighbors_list)

for k in range(len(neighbors_list)):
    for j in neighbors_list[k]:
        print(f"Mean distance between {index_to_atom[str(k)]} and {index_to_atom[str(j)]}: \t"
              f"{mean(mean_distances_dict[str((index_to_atom[str(k)],index_to_atom[str(j)]))])} Å")
        print(f'Standard deviation: \t\t\t{stdev(mean_distances_dict[str((index_to_atom[str(k)],index_to_atom[str(j)]))])} Å',
              end='\n\n')

#Plotting some examples
i=7
time = [x for x in range(len(thermalization_list[5]))]
for j in neighbors_list[i]:
    fig, ax = plt.subplots()
    ax.plot(time, distance_list[i][str(j)])
    ax.set_ylim(0.8, 2.3)
    ax.set_xlim(0, 102)
    ax.set(xlabel='Time [fs]', ylabel='Distance [Å]', title=f'Distance between atom pair {index_to_atom[str(i)]}, {index_to_atom[str(j)]}')
    plt.show()

#Testing the bond_broken_2 function, which calculates the integrity of a bond
fig, ax = plt.subplots()
ax.plot(distance_list[0][str(1)], bond_broken_2(distance_list[0][str(1)],100,1.445,0.03,10))
ax.set_ylim(0, 1.1)
ax.set_xlim(1.25, 3)
ax.set(xlabel='Bond distance [Å]', ylabel='Bond integrity', title=f'Bond integrity for atom pair N1 {index_to_atom[str(1)]}')
plt.show()
```

```
#Removing bonds that are copies of other bonds with the names of the atoms swapped, like C1,H1 vs. H1,C1. They contain
# the exact same information, which is redundant.
all_keys = list(mean_distances_dict.keys())
sorted_keys = [sorted(e) for e in list(mean_distances_dict.keys())]

index=[]
for i, key in enumerate(sorted_keys):
    index.append([i for i, keyi in enumerate(sorted_keys) if key==keyi])

for indexpair in index:
    if len(indexpair) > 1:
        if all_keys[indexpair[1]] in mean_distances_dict:
            del mean_distances_dict[all_keys[indexpair[1]]]

print(list(mean_distances_dict.keys()))

#This code tries to remove any bonds between hydrogen atoms (which doesn't happen in this molecule)
for k in index_to_atom.values():
    neighlist= [n for n in mean_distances_dict.keys() if str(k) in n]
    mainatom=str(k[0])
    if mainatom=="H":
        #print(k)
        for n in neighlist:
            #print(n)
            if n.count("H")>1 and n in mean_distances_dict.keys():
                del mean_distances_dict[n]

#The data files for the cases with ionization must also be entered manually. There are 11 starting geometric
# configurations and 11 ionization levels, which makes for a total of 121 initial setups that each correspond to
# a separate simulation.
runs2=[['startgeo0_ionization0.out', 'startgeo0_ionization1.out', 'startgeo0_ionization2.out', 'startgeo0_ionization3.out',
      'startgeo0_ionization4.out', 'startgeo0_ionization5.out', 'startgeo0_ionization6.out', 'startgeo0_ionization7.out',
      'startgeo0_ionization8.out', 'startgeo0_ionization9.out', 'startgeo0_ionization10.out'],
       ['startgeo1_ionization0.out','startgeo1_ionization1.out', 'startgeo1_ionization2.out', 'startgeo1_ionization3.out',
      'startgeo1_ionization4.out', 'startgeo1_ionization5.out', 'startgeo1_ionization6.out', 'startgeo1_ionization7.out',
      'startgeo1_ionization8.out', 'startgeo1_ionization9.out', 'startgeo1_ionization10.out'],
      ['startgeo2_ionization0.out', 'startgeo2_ionization1.out', 'startgeo2_ionization2.out', 'startgeo2_ionization3.out',
      'startgeo2_ionization4.out', 'startgeo2_ionization5.out', 'startgeo2_ionization6.out', 'startgeo2_ionization7.out',
      'startgeo2_ionization8.out', 'startgeo2_ionization9.out', 'startgeo2_ionization10.out'],
      ['startgeo3_ionization0.out', 'startgeo3_ionization1.out', 'startgeo3_ionization2.out', 'startgeo3_ionization3.out',
      'startgeo3_ionization4.out', 'startgeo3_ionization5.out', 'startgeo3_ionization6.out', 'startgeo3_ionization7.out',
      'startgeo3_ionization8.out', 'startgeo3_ionization9.out', 'startgeo3_ionization10.out'],
      ['startgeo4_ionization0.out', 'startgeo4_ionization1.out', 'startgeo4_ionization2.out', 'startgeo4_ionization3.out',
      'startgeo4_ionization4.out', 'startgeo4_ionization5.out', 'startgeo4_ionization6.out', 'startgeo4_ionization7.out',
      'startgeo4_ionization8.out', 'startgeo4_ionization9.out', 'startgeo4_ionization10.out'],
      ['startgeo5_ionization0.out', 'startgeo5_ionization1.out', 'startgeo5_ionization2.out', 'startgeo5_ionization3.out',
      'startgeo5_ionization4.out', 'startgeo5_ionization5.out', 'startgeo5_ionization6.out', 'startgeo5_ionization7.out',
      'startgeo5_ionization8.out', 'startgeo5_ionization9.out', 'startgeo5_ionization10.out'],
      ['startgeo6_ionization0.out', 'startgeo6_ionization1.out', 'startgeo6_ionization2.out', 'startgeo6_ionization3.out',
      'startgeo6_ionization4.out', 'startgeo6_ionization5.out', 'startgeo6_ionization6.out', 'startgeo6_ionization7.out',
      'startgeo6_ionization8.out', 'startgeo6_ionization9.out', 'startgeo6_ionization10.out'],
      ['startgeo7_ionization0.out', 'startgeo7_ionization1.out', 'startgeo7_ionization2.out', 'startgeo7_ionization3.out',
      'startgeo7_ionization4.out', 'startgeo7_ionization5.out', 'startgeo7_ionization6.out', 'startgeo7_ionization7.out',
      'startgeo7_ionization8.out', 'startgeo7_ionization9.out', 'startgeo7_ionization10.out'],
      ['startgeo8_ionization0.out', 'startgeo8_ionization1.out', 'startgeo8_ionization2.out', 'startgeo8_ionization3.out',
      'startgeo8_ionization4.out', 'startgeo8_ionization5.out', 'startgeo8_ionization6.out', 'startgeo8_ionization7.out',
      'startgeo8_ionization8.out', 'startgeo8_ionization9.out', 'startgeo8_ionization10.out'],
      ['startgeo9_ionization0.out', 'startgeo9_ionization1.out', 'startgeo9_ionization2.out', 'startgeo9_ionization3.out',
      'startgeo9_ionization4.out', 'startgeo9_ionization5.out', 'startgeo9_ionization6.out', 'startgeo9_ionization7.out',
      'startgeo9_ionization8.out', 'startgeo9_ionization9.out', 'startgeo9_ionization10.out'],
      ['startgeo10_ionization0.out', 'startgeo10_ionization1.out', 'startgeo10_ionization2.out', 'startgeo10_ionization3.out',
      'startgeo10_ionization4.out', 'startgeo10_ionization5.out', 'startgeo10_ionization6.out', 'startgeo10_ionization7.out',
      'startgeo10_ionization8.out', 'startgeo10_ionization9.out', 'startgeo10_ionization10.out']]
ionization_list=[]
for geo in runs2:
    for run in geo:
        time_pos, timeserie, orblegend, specieslegend, numberlegend = parse_timestep(run)
        ionization_list.append(time_pos)

#Data from the above files is added to ion_dict, which stores the distances between two atoms for each simulation as a value
# to a key of the form "atom1, atom2".
ion_dict = {}
for key in list(mean_distances_dict.keys()):
    index_i = int(atom_to_index[key.split("'")[1]])
    index_j = int(atom_to_index[key.split("'")[3]])
    for geo in range(0,11):
        for ion in range(0,11):
            current_run=(11*geo)+ion
            ion_dist_list = [dist_timestep(ionization_list[current_run][t],index_i,index_j)
                             for t in range(len(ionization_list[current_run]))]
            if key not in ion_dict:
                ion_dict[key]=[None]*11
                ion_dict[key]=[[None]*11 for x in ion_dict[key]]
            else:
                pass
            ion_dict[key][geo][ion]=ion_dist_list

#Plotting the bond integrity vs. bonding distance
atom_pair = "('C1', 'C2')"
g = 6
ion = 10
fig, ax = plt.subplots()
ax.plot(ion_dict[atom_pair][g][ion], bond_broken_2(ion_dict[atom_pair][g][ion],len(ion_dict[atom_pair][g][ion]),
                                        mean(mean_distances_dict[atom_pair]), stdev(mean_distances_dict[atom_pair]),10))
i = atom_pair.split("'")[1]
j = atom_pair.split("'")[3]
```

```
ax.set(xlabel='Bond distance [Å]', ylabel='Bond integrity', title=f'Bond integrity for atom pair {i} {j} with g={g} and i={ion}')
plt.show()

#Plotting the bond integrity vs. time in the simulations for a few bonds
atom_pairs = ["('N1', 'H4')", "('C1', 'C3')", "('C5', 'O4')", "('C2', 'O1')", "('N1', 'C1')", "('S1', 'S2')"]
g = 7
ion = 9
time = [t for t in range(len(ionization_list[0]))]
for atom_pair in atom_pairs:
    fig, ax = plt.subplots()
    ax.plot(time, bond_broken_2(ion_dict[atom_pair][g][ion],len(ion_dict[atom_pair][g][ion]),
                                mean(mean_distances_dict[atom_pair]), stdev(mean_distances_dict[atom_pair]),10))
    i = atom_pair.split("'")[1]
    j = atom_pair.split("'")[3]
    ax.set(xlabel='Time [fs]', ylabel='Bond integrity', title=f'Bond integrity for atom pair {i} {j} with g={g} and i={ion}')
    plt.show()

#Generating heatmap plots for each bond that display their bond integrity as a function of ionization level and time
for atom_pair in mean_distances_dict.keys():
    i = atom_pair.split("'")[1]
    j = atom_pair.split("'")[3]

    mean_g_dist = [[] for _ in range(11)]
    all_g = [[] for _ in range(11)]
    for ion in range(11):
        all_g[ion] = [ion_dict[atom_pair][g][ion] for g in range(11) if len(ion_dict[atom_pair][g][ion]) == len(time)]

        for t in range(len(time)):
            mean_g_dist[ion].append(mean([all_g[ion][g][t] for g in range(len(all_g[ion]))]))


    z_mesh = np.divide(np.linspace(0,10,11),np.float(26))
    time_mesh = [t for t in range(len(ionization_list[0]))]
    all_mean_integrity = np.transpose([bond_broken_2(mean_g_dist[current_i], len(mean_g_dist[current_i]),
                    mean(mean_distances_dict[atom_pair]), stdev(mean_distances_dict[atom_pair]),10) for current_i in range(11)])

    fig, ax = plt.subplots()

    p = plt.contourf(z_mesh, time_mesh, all_mean_integrity, levels=100, vmin=0., vmax=1.0,
             alpha=1, cmap='plasma')

    fig.colorbar(p, ticks=[0,0.2,0.4,0.6,0.8,1])
    plt.clim(0,1)
    ax.set(xlabel='$\overline{z}$ [e/N]', ylabel='Time [fs]', title=f'Mean bond integrity for atom pair {i} {j}')
   # plot_filename=f'mBI_{i}_{j}.png'
   # plt.savefig(plot_filename, bbox_inches='tight', format='png', dpi=400)
    plt.show()

#Using the function total_fragments to calculate the fragmentation of the molecule at each timestep in every simulation.
BI_cutoff=0.5
lamda=10
total_fragments=frags_from_dists(mean_distances_dict, atom_to_index, ion_dict, lamda, BI_cutoff)

#To make a dictionary of the mass-to-charge ratio for each fragment in all of the simulations, we first extract the charge
# data for each atom.
all_filenames = []
for geo in runs2:
    for run in geo:
        all_filenames.append(run)

atom_charge_dict = {}
for file in all_filenames:
    charges = parse_hirsh(file)
    for atom in range(len(charges[0])):
        if atom_charge_dict.get(index_to_atom[str(atom)]) != None:
            atom_charge_dict[index_to_atom[str(atom)]].append([charges[t][atom] for t in range(len(charges))])
        else:
            atom_charge_dict[index_to_atom[str(atom)]] = []
            atom_charge_dict[index_to_atom[str(atom)]].append([charges[t][atom] for t in range(len(charges))])

#The mass data is loaded into a list and used to calculate the weight of each fragment.Then their mass-to-charge ratio
# can be calculated, and collected into one histogram + one kde plot per ionization level.
ed=ElementData()

frag_weights=[0]*11
frag_weights=[[0]*11 for x in frag_weights]
for geo in range(0,11):
    for ion in range(0,11):
        frag_weights[geo][ion]=[0]*len(total_fragments[geo][ion])


for geo in range (len(frag_weights)):
    for ion in range(len(frag_weights[geo])):
        for frag in range(len(frag_weights[geo][ion])):
            for atm in range(len(total_fragments[geo][ion][frag])):
                frag_weights[geo][ion][frag]+=ed.elementweight[total_fragments[geo][ion][frag][atm][0]]


frags_charges=[0]*11
frags_charges=[[0]*11 for x in frags_charges]
for geo in range(11):
    for run in range(11):
        frags_charges[geo][run] = [0 for x in total_fragments[geo][run]] #The same # of frags and charges
        for frag in range(len(total_fragments[geo][run])):
            for atm in total_fragments[geo][run][frag]:
                frags_charges[geo][run][frag] += atom_charge_dict[atm][11*geo + run][-1]
```

```
i = 10     # OBS: mass-spec not possible for i=0! charge is 0
mass_charge = []
for g in range(11):
    mass_charge.extend(np.ndarray.tolist(np.divide([m for m in frag_weights[g][i]], [c for c in frags_charges[g][i]])))


fig, ax = plt.subplots()

plt.hist(mass_charge, bins=np.arange(min(mass_charge), max(mass_charge) + 0.5, 0.125))
ax.set(xlabel='Mass/Charge [u/e]', ylabel='Counts [#]', title=f'Masspec for ionization {i} at t = {len(time)} fs')
plot_filename=f'H_masspec_{i}.png'
plt.savefig(plot_filename, bbox_inches='tight', format='png', dpi=400)
plt.show()


fig, ax = plt.subplots()
sns.kdeplot(mass_charge, bw=0.125)
ax.set(xlabel='Mass/Charge [u/e]', ylabel='Density', title=f'Masspec for ionization {i} at t = {len(time)} fs')
plot_filename=f'G_masspec_{i}.png'
plt.savefig(plot_filename, bbox_inches='tight', format='png', dpi=400)
plt.show()
```

analyze_trajectories.py

The majority of this code was provided by Oscar to perform some essential functions in the main script, like loading and parsing the simulation data. Some functions are written by me and/or Ebba Koerfer for this project specifically – these are `bond_broken_2`, `mean_distance_dict` and `frags_from_dists`. The last one is based on code I wrote during a previous project.

```
#!/usr/bin/env python
import os, sys
import numpy as np
import shutil
import matplotlib.pyplot as plt
from statistics import mean, stdev
from numpy import linalg as LA
from scipy import interpolate
from itertools import combinations

# To analyze preparsed with "the naming convention", run e.g. ./analyze_preparsed.py ALA 1 10 1 10 C4 "H10 H11 H12" "Alanine C-H methyl bonds"
class atom:
    def __init__(self):
        self.name=""
        self.rvec=np.zeros(3)
        self.dvec = np.zeros(3) # direct
        self.pdos = np.zeros(1)
        self.sumdos = np.zeros(1)
        self.color = 0 # color index is used to find color from list in plotter, otherwise it's messier to change.
        self.phonons = []
        self.speciesName = ""
        self.speciesNumber = 0
        self.specnum=0
        self.speciesZNumber = 0
        self.mass = 0.0
        self.hirshfeldcharge=0.0
        self.mulliken_legend=[]
        self.mulliken_charges=[]

    def distance(self,center=np.asarray([0.0,0.0,0.0])):
        return np.linalg.norm(np.subtract(self.rvec,center))
        # return float(np.sqrt(self.rvec[0]**2+self.rvec[1]**2+self.rvec[2]**2))

    def in_cluster(self,maxrad,center=np.asarray([0.0,0.0,0.0]),minrad=0.0):
        return (self.distance(center) <= float(maxrad) and self.distance(center) >= float(minrad))


class lattice:
     def __init__(self):
         self.bravais=np.zeros((3,3))
         self.reciprocal=np.zeros((3,3))
         self.atoms=[]
         self.lattparam=0.0
         self.indSpecies=[]    # Number of atoms for one individual specie
         self.numSpecies=0     # Number of species
         self.indSpeciesNames=[]
         self.coordtype=""


def deleteContent(fName): #Clearing the previous input file
            with open(fName, "w"):
                pass

def Date_and_Time():
    from time import gmtime, strftime #Current date and time
    t = strftime("%Y-%m-%d %H:%M:%S", gmtime())
    return t
```

```
def parse_text_bond_data(filename):
    bond_integrity=[]
    f=open(filename,'r')
    for i, line in enumerate(f.readlines()):
        if line.split()[-1][-1] is not "]":
            full_line = line.split()
        else:
            full_line = full_line + line.split()
            bond_integrity.append(np.asarray(filter(None,[element.strip('[]') for element in full_line[1:]])).astype(np.float))
    return np.asarray(bond_integrity)

def get_neighborlist(timestep,rmax):
    neighborlist=[]
    rmin = 0.1 # Do not include self
    for i, atm in enumerate(timestep):
        neighborlist.append(find_atoms_within_radius(timestep,atm.rvec,rmax,rmin))
    return neighborlist

def get_neighborlist_2(timestep,rmax):
    neighborlist=[]
    rmin = 0.1 # Do not include self
    for i, atm in enumerate(timestep):
        neighborlist.append(find_atoms_within_radius(timestep,atm.rvec,rmax,rmin))
        if atm.name is "H" and len(neighborlist[i]) > 1:
            wrong_index = abs(max([i-x for x in neighborlist[i]]) - i)
            neighborlist[i].remove(wrong_index)

    return neighborlist


def find_atoms_within_cartesian(cluster,xlim,ylim,zlim):
    indices=[]
    for i, atm in enumerate(cluster):
        within = ((float(xlim[0])<= float(atm.rvec[0]) <= float(xlim[1])) and
                  (float(ylim[0])<= float(atm.rvec[1]) <=float(ylim[1])) and
                  (float(zlim[0])<= float(atm.rvec[2]) <=float(zlim[1])))
        if within:
            indices.append(i)
    return indices

def find_atoms_within_radius(cluster,center,rmax,rmin=0.0):
    indices=[]
    for i, atm in enumerate(cluster):
        if (atm.in_cluster(rmax,center,rmin)):
            indices.append(i)
    return indices

def get_neighborlist(timestep,rmax):
    neighborlist=[]
    rmin = 0.1 # Do not include self
    for i, atm in enumerate(timestep):
        neighborlist.append(find_atoms_within_radius(timestep,atm.rvec,rmax,rmin))
    return neighborlist



#checking if element is int
def Is_Int(s):
    try:
        int(s)
        return True
    except ValueError:
        return False

#Parsing .ANI file
def parse_ANI(filename):
    f = open(filename, 'r')
    contents = f.readlines()
    f.close()
    atoms=[]
    time_serie=[]
    for i in range(len(contents)):
        if (Is_Int(contents[i])):
            atoms_in_timestep=int(contents[i].split()[0])
            for j in range(i+2,i+2+int(atoms_in_timestep)):
                atoms.append(atom())
                atoms[-1].rvec=[float(contents[j].split()[k]) for k in range(1,4)]
                atoms[-1].name=contents[j].split()[0]
            time_serie.append(atoms)
            atoms=[]
    return atoms_in_timestep, time_serie


def distR(D):
    N = np.loadtxt(D, dtype=np.float, delimiter=',')
    Q = [np.linalg.norm(a-b) for a, b in combinations(N, 2)]
    return Q


def dist_timestep(timestep,atom1,atom2):
    return np.linalg.norm(np.subtract(timestep[atom2].rvec,timestep[atom1].rvec))


def bond_broken(dist,mean,T=150):
    B=[]
```

```
    for num in range(0,T):
        try:
            a = np.sqrt((np.sum(dist[num]-mean))**2)-0.5
        except:
                a = a # will this work for simulations that broke before T=150?
        b = 0.03*a
        c = np.exp(b)
        d = 1+c
        e = 1/d
        B.append(e)

    return np.asarray(B)

def bond_broken_2(dist, T, mean, sigma, lamda):
    B=[]
    for num in range(0,T):
        e = (1 + np.exp(lamda*(dist[num]-mean-sigma-0.5)))**(-1)
        B.append(e)

    return np.asarray(B)


def mean_distance_dict(thermalization_list, index_to_atom, neighbors_list):
    """Returns a dictionary with keys in the form '(Atom_A_index, Atom_B_index)' with values in the form of lists, where the elements are
    mean values for the distance between atom A and B over time for each thermalization run in thermalization list (which contains
    information from parse_timestep function). index_to_atom is a dictionary made from make_atom_dictionary_from_timeserie() and
    neighbors_list from get_neighborlist()"""
    mean_distance_lexi = {} # Dictionary with every atom pair_kj, in tuple-form, as keys and their values are mean distances over
                        # all time positions for every thermalization run

    for run_index in range(len(thermalization_list)):
        distance_list = []
        for k in range(len(neighbors_list)): # atom_k, atom_j = atom pair_kj
            distance_lexi = {}
            for j in neighbors_list[k]:      # distance_list has dicts for every atom_k with neighbor atom_j as key and with
                                        # distance_atom pair_kj(t) as values
                distance_lexi[str(j)] = [dist_timestep(thermalization_list[run_index][t_i],k,j) for t_i in
                                         range(len(thermalization_list[run_index]))]
            distance_list.append(distance_lexi)
            for j in neighbors_list[k]:
                if mean_distance_lexi.get(str((index_to_atom[str(k)],index_to_atom[str(j)]))) != None:
                    mean_distance_lexi[str((index_to_atom[str(k)],index_to_atom[str(j)]))].extend([mean(distance_list[k][str(j)])])
                else:
                    mean_distance_lexi[str((index_to_atom[str(k)],index_to_atom[str(j)]))] = [mean(distance_list[k][str(j)])]
    return mean_distance_lexi, distance_list


def frags_from_dists(mean_distances_dict, atom_to_index, ion_dict, lamda, BI_cutoff):
    l=lamda #Lambda value
    broken_bonds_dict={}
    for bond in list(mean_distances_dict.keys()):
        broken_bonds_dict[bond]=[None]*11
        broken_bonds_dict[bond]=[[None]*11 for x in broken_bonds_dict[bond]]
        for geo in range(0,11):
            for ion in range(0,11):
                BI = bond_broken_2(ion_dict[bond][geo][ion],len(ion_dict[bond][geo][ion]),
                                   mean(mean_distances_dict[bond]), stdev(mean_distances_dict[bond]),l)
                if BI[-1] <= BI_cutoff:
                    broken_bonds_dict[bond][geo][ion]="broken"
                else:
                    broken_bonds_dict[bond][geo][ion]="intact"

    total_fragments=[None]*11
    total_fragments=[[None]*11 for x in total_fragments]

    for geo in range(0,11):
        for ion in range(0,11):
            polyatomic=[]
            monoatomic=[]
            for bond in broken_bonds_dict.keys():
                atoms= [x for x in atom_to_index.keys() if bond.split("'")[1]==x or bond.split("'")[3]==x]
                if broken_bonds_dict[bond][geo][ion]=="intact":
                    found=False
                    merged=False
                    for j in range (len(polyatomic)):
                        if (atoms[0] in polyatomic[j]) and (atoms[1] not in polyatomic[j]):
                            for k in range(len(polyatomic)):
                                if atoms[1] in polyatomic[k] and atoms[0] not in polyatomic[k]:
                                    polyatomic[j].extend(polyatomic[k])
                                    polyatomic[k]=[]
                                    merged=True
                                    break
                            if not merged:
                                polyatomic[j].append(atoms[1])
                            found=True
                        elif (atoms[1] in polyatomic[j]) and (atoms[0] not in polyatomic[j]):
                            for k in range(len(polyatomic)):
                                if atoms[0] in polyatomic[k] and atoms[1] not in polyatomic[k]:
                                    polyatomic[j].extend(polyatomic[k])
                                    polyatomic[k]=[]
                                    merged=True
                                    break
                            if not merged:
                                polyatomic[j].append(atoms[0])
                            found=True
                        elif (atoms[0] in polyatomic[j]) and (atoms[1] in polyatomic[j]):
```

```
                        found=True
                    else:
                        pass
                if found==False: #This is the case where the atoms do not occur anywhere in the current version of "polyatomic"
                    polyatomic.append(atoms)
            elif broken_bonds_dict[bond][geo][ion]=="broken":
                atom0_in_somefrag=False
                atom1_in_somefrag=False
                for other_bond in broken_bonds_dict.keys():
                    if broken_bonds_dict[other_bond][geo][ion]=="intact":
                        if (atoms[0]==other_bond.split("'")[1] or atoms[0]==other_bond.split("'")[3]):
                            atom0_in_somefrag=True
                        if (atoms[1]==other_bond.split("'")[1] or atoms[1]==other_bond.split("'")[3]):
                            atom1_in_somefrag=True
                if not atom0_in_somefrag and atoms[0] not in monoatomic:
                    monoatomic.append(atoms[0])
                if not atom1_in_somefrag and atoms[1] not in monoatomic:
                    monoatomic.append(atoms[1])
                else:
                    pass
            else:
                print("broken_bonds_dict["+bond+"]["+geo+"]["+ion+"] was not assigned a value")

        for i in range (len(monoatomic)):
            monoatomic[i]=[monoatomic[i]]
        empty_indices=[]
        for j in range(len(polyatomic)):
            if not polyatomic[j]:
                empty_indices.append(j)
        for empty_index in empty_indices:
            del polyatomic[empty_index]
        fragments=[]
        fragments.extend(polyatomic)
        fragments.extend(monoatomic)
        total_fragments[geo][ion]=fragments
    return total_fragments

def write_xyz_anim(filename,timesteps,skipstep=1):
    f = open(filename,'w')
    for i, step in enumerate(timesteps):
        if (np.mod(i,skipstep)<0.5):
            f.write(str(len(step))+"\n")
            f.write('Timestep: '+str(i*skipstep)+"\n")
            for atm in step:
                f.write(str(atm.name)+" "+str(atm.rvec[0])+" "+str(atm.rvec[1])+" "+str(atm.rvec[2])+"\n")


def parse_hirsh_from_file(ion,lastion,acid):
    all_mean_hirsh=[]
    all_std_hirsh=[]
    for ionstage in range(ion,lastion):
        hirsh=[]
        print("acid, ionstage: ", str(acid), str(ionstage))
        for geostage in range(geometry,lastgeometry):
            Sim = './startgeo{0}_ionization{1}'.format(geostage,ionstage)
            os.chdir(Sim)
            try:
                hirsh.append(parse_hirsh("./stdout"))
            except:
                print("Failed to parse Hirshfeld for: {0}/startgeo{1}_ionization{2}".format(acid, geostage, ionstage))
            os.chdir("..")
        #print np.asarray(hirsh).mean(0).shape
        mean_data_name='{0}_hirshfeld_charge_{1}_hirshrun.dat'.format(acid,ionstage)
        np.savetxt(mean_data_name,np.asarray(hirsh).mean(0))
        mean_data_name='{0}_stdev_hirshfeld_charge_{1}_hirshrun.dat'.format(acid,ionstage)
        np.savetxt(mean_data_name,np.asarray(hirsh).std(0))
        all_mean_hirsh.append(np.asarray(hirsh).mean(0))
        all_std_hirsh.append(np.asarray(hirsh).std(0))
    return all_mean_hirsh, all_std_hirsh

def parse_hirsh(filename):
    f = open(filename, 'r')
    contents = f.readlines()
    f.close()
    timesteps=[]
    charges=[]
    numatm=0
    for i in range(len(contents)):
        if ("NumberOfAtoms" in contents[i]):
            numatm=contents[i].split()[1]
        if ("Atom #    Qatom  Species" in contents[i]):
            for j in range(i+1,i+1+int(numatm)):
                charges.append(contents[j].split()[1])
            timesteps.append(np.asarray(charges,dtype=float))
            charges=[]
    return timesteps

def parse_eigenvalues(filename):
    f = open(filename, 'r')
    contents = f.readlines()
    f.close()
    timeserie_eig=[]
    timeserie_occ=[]

    for i, line in enumerate(contents):
        if ("Timestep" in line):
```

```
            current_step=int(line.split()[1])
            print(current_step)
            num_eigens=int(line.split()[3])
            print(num_eigens)
            eigenvalues=[]
            occupations=[]
            for j in range(i+1,i+num_eigens+1):
                eigenvalues.append(np.asarray(contents[j].split()[0:2], dtype=float))
                occupations.append(np.asarray(contents[j].split()[3:5], dtype=float))
            # Transpose to get spin-channels as timeserie[itime][ispin][:]
            timeserie_eig.append(np.transpose(np.asarray(eigenvalues)))
            timeserie_occ.append(np.transpose(np.asarray(occupations)))

    return np.asarray(timeserie_eig), np.asarray(timeserie_occ)

def read_preparsed_hirsh(acid,ion):
    mean_data_name='{0}_hirshfeld_charge_{1}_hirshrun.dat'.format(acid,ion)
    mean_hirsh=np.loadtxt(mean_data_name, dtype=np.float)
    mean_data_name='{0}_stdev_hirshfeld_charge_{1}_hirshrun.dat'.format(acid,ion)
    std_hirsh=np.loadtxt(mean_data_name, dtype=np.float)
    return mean_hirsh, std_hirsh

def make_atom_dictionary(filename):
    natoms, md_verlet = parse_ANI(filename)
    atomdict={}
    name_list=[atm.name for atm in md_verlet[0]]
    for i, atm in enumerate(md_verlet[0]):
        new_atom_number=name_list[0:i].count(atm.name)+1
        key=str(i)
        value=atm.name+str(new_atom_number)
        atomdict[key]=value
    inverted_dict = dict(map(reversed, atomdict.items()))
    return atomdict, inverted_dict

def make_atom_dictionary_from_timeserie(timeserie):
    atomdict={}
    name_list=[atm.name for atm in timeserie[0]]
    print(name_list), print(type(name_list[0]))
    for i, atm in enumerate(timeserie[0]):
        print(str(i), atm.name)
        new_atom_number=name_list[0:i].count(atm.name)+1
        key=str(i)
        value=atm.name+str(new_atom_number)
        atomdict[key]=value
    inverted_dict = dict(map(reversed, atomdict.items()))
    return atomdict, inverted_dict


def parse_xyz(filename):
    xyz=[]
    f = open(filename, 'r')
    contents = f.readlines()
    f.close()
    for line in contents:
        xyz.append(np.asarray(line.split()[1:4], dtype=float))
    return np.transpose(np.asarray(xyz))

def parse_timestep(filename, outfile=None):
    f = open(filename, 'r')
    contents = f.readlines()
    print("filename: "+str(filename))
    print("length of file: "+str(len(contents)))
    f.close()
    numatm=0
    basissize='SZP'
    time_pos=[]
    time_mulliken=[]
    timesteps=[]
    specieslegend={}
    numberlegend={}
    mulls=[]
    orblegend=[]
    for i in range(len(contents)):
        if ("NumberOfAtoms" in contents[i]):
            numatm=int(contents[i].split()[1])
            print("Number of Atoms: "+str(numatm))
            break

    for i in range(len(contents)):
        if ("PAO.BasisSize" in contents[i]):
            basissize=str(contents[i].split()[1])
            print("Basis Size: "+str(basissize))
            break


    for i in range(len(contents)):
        if ("SpinPolarized" in contents[i]):
            if ("true") in contents[i]:
               spins=2
            else:
               spins=1
            break
    print("Spin components: "+str(spins))

    for i in range(len(contents)):
        if ("AtomicSpecies" in contents[i]):
```

```
            #print "Found AtomicCoord..."
            for j in range(i+1,len(contents)):
                print(str(j-i)+"  "+str(contents[j].split()))
                numberlegend[str(j-i)]=str(contents[j].split()[3])
                if ("AtomicCoordinatesAndAtomicSpecies" in contents[j+1]):
                #    print "Found!"
                    break
            else:
                continue
            break

    for i in range(len(contents)):
        if ("ChemicalSpeciesLabel" in contents[i]):
            for j in range(i+1,len(contents)):
                specieslegend[str(contents[j].split()[0])]=str(contents[j].split()[2])
                if ("ChemicalSpeciesLabel" in contents[j+1]):
                    break
            else:
                 continue
            break
    # print numberlegend
    # print specieslegend


    for i in range(len(contents)):
        if ("Atomic coordinates (Ang)" in contents[i]):
            atoms=[]
            for j in range(i+1,i+numatm+1):
                #print(contents[j])
                #print(contents[j].split()[3])
                #print(numberlegend)
                atoms.append(atom())
                atoms[-1].rvec=[float(contents[j].split()[k]) for k in range(0,3)]
                atoms[-1].name=specieslegend[numberlegend[str(contents[j].split()[4])]]
                #atoms[-1].name=specieslegend[str(contents[j].split()[3])]
                #print(atoms[-1].name)
            time_pos.append(atoms)

# Approximately two lines per atom times number of spins + overhead of a few lines
    approx_mulliken_size=numatm*(3+spins*2)

    for i in range(len(contents)):
        if ("mulliken: Atomic and Orbital Populations:" in contents[i]):
            mulls, orblegend= parse_mulliken(contents[i:i+approx_mulliken_size],numatm,basissize,spins,outfile)
            time_mulliken.append(mulls)

    return time_pos, time_mulliken, orblegend, specieslegend, numberlegend
```

References

[1] Dagmar Ringe Gregory A. Petsko. *Protein structure and function*. Oxford University Press, 2009.

[2] Nicusor Timneanu Henry N. Chapman Carl Caleman. "Diffraction before destruction". In: *Philosophical transactions of the Royal Society B* 369.1647 (2014). DOI: https://doi.org/10.1098/rstb.2013.0313.

[3] Herbert A. Hauptman. "History of X-ray Crystallography". In: *Structural Chemistry* 1 (1990). DOI: https://doi-org.ezproxy.its.uu.se/10.1007/BF00674136.

[4] Alan L. Mackay. "Generalized Crystallography". In: *Science of Crystal Structures: Highlights in Crystallography*. Ed. by Istvan Hargittai and Balazs Hargittai. Springer, 2015. Chap. 4, pp. 37–42.

[5] John C.H. Spence. "Outrunning damage: Electrons vs X-rays – timescales and mechanisms". In: *Structural dynamics* 4 (2017). DOI: https://doi.org/10.1063/1.4984606.

[6] Oscar Grånäs et al. "Femtosecond bond breaking and charge dynamics in ultracharged amino acids". In: *The Journal of Chemical Physics* 151 (2019). DOI: https://doi.org/10.1063/1.5116814.

[7] Ibrahim Eliah Dawod. "Structural integrity of highly ionized peptides". MA thesis. Uppsala universitet, 2019.

[8] NEUROtiker. *Structure of L-cysteine*. URL: https://commons.wikimedia.org/wiki/File:L-Cystein_-_L-Cysteine.svg.

[9] João Mousquès Renke Dullaart. *Keratin – Structure, properties and applications*. Nova science publishers, 2012.

[10] Jü. *(R,R)-Cystine*. URL: https://commons.wikimedia.org/wiki/File:(R,R)-Cystine_BLUE_Structural_Formula.png.

[11] Steven A. Hardinger. *Disulfide bond*. URL: http://www.chem.ucla.edu/~harding/IGOC/D/disulfide_bridge.html.

[12] Ben Mills. *Ball-and-stick model of cystine*. Atomic labels were added by me, the author. URL: https://commons.wikimedia.org/wiki/File:Cystine-3D-balls.png.