Molecular Dynamics and Ab Initio Simulations : Developing a Predictive Model for Molecular Energy Configurations

Table of contents:

This project involves utilizing results from ab initio molecular dynamics simulations to develop a model capable of predicting the energy associated with specific molecular configurations. It leverages two datasets: the first encompasses simulations on the dynamics of the Zundel ion (H2O-H-H2O), while the second focuses on a Mo2S4 aggregate. Each dataset contains about 10,000 atomic configurations and their corresponding potential energies, providing a comprehensive basis for accurate energy prediction in molecular systems.

First view on data

MO2S4 dataset

First, let's try to render a configuration on a 3D plot. We will first look into the MO2S4 dataset by plotting the first and last configurations.

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

MO2S4_file_path = 'data/MO2S4/TRAJEC_short.xyz'
```

```python
# Re-reading the file to confirm its structure
with open(MO2S4_file_path, 'r') as file:
    MO2S4_file_content = file.readlines()

# Parsing the first configuration from the file
# Skipping the first two lines (number of atoms and comment) and taking the next six lin

def plot_configuration_MO2S4(first_line, last_line):

    configuration = MO2S4_file_content[first_line:last_line]

    # Extracting atom types and their positions
    atom_types = []
    positions = []

    for line in configuration:
        parts = line.split()
        atom_types.append(parts[0])  # Atom type (Mo or S)
        positions.append([float(parts[1]), float(parts[2]), float(parts[3])])  # x, y, z

    positions = np.array(positions)

    # Creating a 3D scatter plot using 'o' markers to represent atoms
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')

    # Plotting each atom with 'o' marker
    for i, atom in enumerate(atom_types):
        x, y, z = positions[i]
        color = 'gray' if atom == 'Mo' else 'yellow'  # Gray for Mo, Yellow for S
        ax.scatter(x, y, z, c=color, marker='o', s=100)  # Adjust size as needed

    # Adding labels and title
    ax.set_xlabel('X axis')
    ax.set_ylabel('Y axis')
    ax.set_zlabel('Z axis')
    ax.set_title('3D Visualization of Atomic Configuration with Scatter Plot')

    # Add legend
    ax.scatter([], [], [], c='gray', marker='o', s=100, label='Mo')
    ax.scatter([], [], [], c='yellow', marker='o', s=100, label='S')
    ax.legend()

    # Set the view to XY plane
    ax.view_init(elev=90, azim=0)

    # Add a title

    plt.title('Configuration n°' + str(last_line//8))

    plt.show()

plot_configuration_MO2S4(2,8)
plot_configuration_MO2S4(88002,88008)
```

We can check these configurations with an image from Jmol:

First configuration :

image

Last configuration :

image

We can see the plotting are correct (even tho there is a slight difference in the position of the atoms, but it is due to the fact that the Jmol image is not in the exact same orientation as the plotting).

We notice that the configurations are quite different.

Properties of `TRAJEC_short.xyz` :

- 11,001 configurations (88,008 lines/8 lines per configuration).
- No units for positions, but it can be assumed that they are in Angstroms ($\AA = 10^-10m$).

It is quickly noticed that there are 11,001 lines in the `energies.out` file, and the same in the `potential-energy` file. It seems that the values in `potential-energy.txt` correspond to those in `energies.out`. We can then verify this:

```
In [ ]: MO2S4_energies_out = pd.read_csv("data/MO2S4/energies.out", delim_whitespace=True)
        MO2S4_energies_out.head()
```

```
In [ ]: MO2S4_potential_energy = pd.read_csv('data/MO2S4/potential-energy.txt', delim_whitespace
        MO2S4_potential_energy.columns = ['potential_energy']
        MO2S4_potential_energy.head()
```

```
In [ ]: # On teste si les deux colonnes sont égales
        print(MO2S4_energies_out['potential_energy'].equals(MO2S4_potential_energy['potential_en
```

The two columns quickly lead us to assume that the `energies.out` file represents the energies of the 11,001 configurations in the `TRAJEC_short.xyz` file. We can then simply use the energies.out file to train our model.

Zundel dataset

This dataset is composed of only two files, `new_energies_sparse.out` and `new_positions_sparse.xyz`. We can directly see that these two files are, resp., the energy table we saw on `energies.out` and the `TRAJEC_short.xyz`. Let's import them and look the same way as the MO2S4.

```
In [ ]: zundel_file_path = 'data/zundel_ions/new_positions_sparse.xyz'

        # Re-reading the file to confirm its structure
        with open(zundel_file_path, 'r') as file:
```

```
    zundel_file_content = file.readlines()

# Parsing the first configuration from the file
# Skipping the first two lines (number of atoms and comment) and taking the next six lin

def plot_configuration_zundel(first_line, last_line):

    configuration = zundel_file_content[first_line:last_line]

    # Extracting atom types and their positions
    atom_types = []
    positions = []

    for line in configuration:
        parts = line.split()
        atom_types.append(parts[0])  # Atom type (O or H)
        positions.append([float(parts[1]), float(parts[2]), float(parts[3])])  # x, y, z

    positions = np.array(positions)

    # Creating a 3D scatter plot using 'o' markers to represent atoms
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')

    # Plotting each atom with 'o' marker
    for i, atom in enumerate(atom_types):
        x, y, z = positions[i]
        color = 'red' if atom == 'O' else 'gray'  # Gray for H, Red for O
        ax.scatter(x, y, z, c=color, marker='o', s=100)  # Adjust size as needed

    # Adding labels and title
    ax.set_xlabel('X axis')
    ax.set_ylabel('Y axis')
    ax.set_zlabel('Z axis')
    ax.set_title('3D Visualization of Atomic Configuration with Scatter Plot')

    # Add legend
    ax.scatter([], [], [], c='gray', marker='o', s=100, label='H')
    ax.scatter([], [], [], c='red', marker='o', s=100, label='O')
    ax.legend()

    # Add a title

    plt.title('Configuration n°' + str(last_line//8))

    plt.show()

plot_configuration_zundel(2,9)
plot_configuration_zundel(89993,90000)
```

Again, we can check these configurations with an image from Jmol:

First configuration :

image

Last configuration :

image

We see that ...

Now let's check the table of energies:

In [ ]:
```
# uniquement prendre la 5ème colonne
zundel_potential_energy = pd.read_csv('data/zundel_ions/new_energies_sparse.out', delim_
zundel_potential_energy.head()
zundel_potential_energy.describe()
```

Data Preprocessing

The goal here is to describe the data and to prepare it for the model. Indeed, we will try to directly use the data as it is, without any preprocessing and use a naive model, and then we will try to preprocess it to see if it improves the model.

Naive and linear regression model

We will first try to use the data as it is, without any preprocessing. By this, I mean we will use the coordinates of the atoms as features and the potential energy as the target. Then, we will use the mean of the potential energy as a naive model to predict the potential energy of a configuration and check the score of this model.

In [ ]:
```
# data : MO2S4_file_content (6 lignes par configuration, séparés par 2 lignes, commence
# Les 6 lignes sont : type d'atome, x, y, z
# on va d'abord récupérer dans l'ordre les positions des atomes dans chaque configuratio

def get_liste_configurations_MO2S4():
    liste_configurations = []

    for i in range(2, len(MO2S4_file_content), 8):
        configuration = []
        for j in range(6):
            configuration.append(MO2S4_file_content[i+j].split()[1:])
        liste_configurations.append(configuration)

    liste_configurations = np.array(liste_configurations, dtype=float)

    return liste_configurations

liste_configurations = get_liste_configurations_MO2S4()

print(f" Nombres de configurations : {len(liste_configurations)}"
      f"\n Dimensions de la première configuration : {liste_configurations[0].shape}"
      f"\n Première configuration : \n {liste_configurations[0]}"
      f"\n Moyenne des positions des configurations : \n {np.mean(liste_configurations,
      f"\n Ecart-type des positions des configurations : \n {np.std(liste_configurations
      f"\n Minimum des positions des configurations : \n {np.min(liste_configurations, a
      f"\n Maximum des positions des configurations : \n {np.max(liste_configurations, a
```

In [ ]:
```
# On peut maintenant récupérer les énergies potentielles dans le fichier potential-energ

def get_energies_MO2S4():
    energies = []

    for i in range(len(MO2S4_potential_energy)):
        energies.append(MO2S4_potential_energy['potential_energy'][i])

    energies = np.array(energies, dtype=float)

    return energies

energies = get_energies_MO2S4()

print(f" Nombres d'énergies : {len(energies)}"
      f"\n Première énergie : {energies[0]}"
      f"\n Dernière énergie : {energies[-1]}"
      f"\n Moyenne des énergies : {np.mean(energies)}"
      f"\n Ecart-type des énergies : {np.std(energies)}"
      f"\n Minimum des énergies : {np.min(energies)}, indice : {np.argmin(energies)}"
      f"\n Maximum des énergies : {np.max(energies)}, indice : {np.argmax(energies)}")
```

Naive model

Now, we can create a naive model that predicts the mean of the potential energies using the atom positions as features and the potential energies as the target. The scoring method used will be the Mean Squared Error (MSE).

In [ ]:
```
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

# On peut maintenant créer un modèle naïf qui va prédire la moyenne des énergies potenti

X = liste_configurations
y = energies

y_pred = np.full(len(energies), np.mean(energies))

score = mean_squared_error(energies, y_pred)

print(f" MSE du modèle naïf : {score}")
print(f" RMSE du modèle naif : {np.sqrt(score)}")
```

We obtain an MSE of 1048, which is a high score for energies on the order of 10^2. Therefore, we can conclude that the naive model is not suitable for this problem.

We also easily notice that the RMSE is equal to the standard deviation of the energies, which makes sense since the naive model predicts the mean of the energies, and the standard deviation is the measure of energy dispersion around the mean.

Linear regression model

We can now try a slightly more complex model, using the atom positions as features and the potential energies as the target. We will use a linear regression model and split the data into a training set and a test set. Then, we will calculate the MSE and RMSE of this model.

```
In [ ]: from sklearn.model_selection import train_test_split

        # On sépare les données en un jeu d'entraînement et un jeu de test

        print(X.shape)
        flattened_X = X.reshape(11001, 18)
        print(flattened_X.shape)
        print(y.shape)
        X_train, X_test, y_train, y_test = train_test_split(flattened_X, y, test_size=0.2, rando

        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        print(f" Dimensions du jeu d'entraînement : {X_train.shape}"
              f"\n Dimensions du jeu de test : {X_test.shape}"
              f"\n Dimensions du jeu d'entraînement : {y_train.shape}"
              f"\n Dimensions du jeu de test : {y_test.shape}")
```

```
In [ ]: from sklearn.linear_model import LinearRegression

        # On crée le modèle de régression linéaire

        linear_regression = LinearRegression()

        # On entraîne le modèle

        linear_regression.fit(X_train, y_train)

        # On prédit les énergies potentielles du jeu de test

        y_pred = linear_regression.predict(X_test)

        # On calcule le MSE et le RMSE

        score = mean_squared_error(y_test, y_pred)

        print(f" MSE du modèle de régression linéaire : {score}")
        print(f" RMSE du modèle de régression linéaire : {np.sqrt(score)}")
```

We notice that we have reduced the MSE, but not by much. Therefore, we can deduce that the linear regression model is not suitable for this problem. The question then arises whether the issue lies with the model or the data. We will attempt to improve the data to see if the model performs better.

It's also worth noting that in this case, we do not differentiate between atoms by their type; we only consider the positions of the atoms. However, since the atom indices remain the same in the `TRAJEC_short.xyz` file, we can assume that their types are determined accordingly.

Again with Zundel Ions

We will repeat the same process for the Zundel ions.

In [ ]:
```python
# data : zundel_file_content (7 lignes par configuration, séparés par 2 lignes, commence

def get_liste_configurations_zundel():
    liste_configurations = []

    for i in range(2, len(zundel_file_content), 9):
        configuration = []
        for j in range(7):
            configuration.append(zundel_file_content[i+j].split()[1:])
        liste_configurations.append(configuration)

    liste_configurations = np.array(liste_configurations, dtype=float)

    return liste_configurations

liste_configurations = get_liste_configurations_zundel()

print(liste_configurations.shape)

print(f" Nombres de configurations : {len(liste_configurations)}"
      f"\n Dimensions de la première configuration : {liste_configurations[0].shape}"
      f"\n Première configuration : \n {liste_configurations[0]}"
      f"\n Moyenne des positions des configurations : \n {np.mean(liste_configurations,
      f"\n Ecart-type des positions des configurations : \n {np.std(liste_configurations
      f"\n Minimum des positions des configurations : \n {np.min(liste_configurations, a
      f"\n Maximum des positions des configurations : \n {np.max(liste_configurations, a
```

In [ ]:
```python
# On peut maintenant récupérer les énergies

def get_energies_zundel():
    energies = []

    for i in range(len(zundel_potential_energy)):
        energies.append(zundel_potential_energy['potential_energy'][i])

    energies = np.array(energies, dtype=float)

    return energies

energies = get_energies_zundel()

print(f" Nombres d'énergies : {len(energies)}"
      f"\n Première énergie : {energies[0]}"
      f"\n Dernière énergie : {energies[-1]}"
      f"\n Moyenne des énergies : {np.mean(energies)}"
      f"\n Ecart-type des énergies : {np.std(energies)}"
      f"\n Minimum des énergies : {np.min(energies)}, indice : {np.argmin(energies)}"
      f"\n Maximum des énergies : {np.max(energies)}, indice : {np.argmax(energies)}")
```

In [ ]:
```python
# On peut maintenant créer un modèle naïf qui va prédire la moyenne des énergies potenti

X = liste_configurations
y = energies

y_pred = np.full(len(energies), np.mean(energies))

score = mean_squared_error(energies, y_pred)

print(f" MSE du modèle naïf : {score}")
print(f" RMSE du modèle naïf : {np.sqrt(score)}")
```

In [ ]:
```python
from sklearn.model_selection import train_test_split

# On sépare les données en un jeu d'entraînement et un jeu de test

print(X.shape)
flattened_X = X.reshape(10000, 21)
print(flattened_X.shape)
print(y.shape)

X_train, X_test, y_train, y_test = train_test_split(flattened_X, y, test_size=0.2, rando
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print(f" Dimensions du jeu d'entraînement : {X_train.shape}"
      f"\n Dimensions du jeu de test : {X_test.shape}"
      f"\n Dimensions du jeu d'entraînement : {y_train.shape}"
      f"\n Dimensions du jeu de test : {y_test.shape}")
```

In [ ]:
```python
from sklearn.linear_model import LinearRegression

# On crée le modèle de régression linéaire

linear_regression = LinearRegression()

# On entraîne le modèle

linear_regression.fit(X_train, y_train)

# On prédit les énergies potentielles du jeu de test

y_pred = linear_regression.predict(X_test)

# On calcule le MSE et le RMSE

score = mean_squared_error(y_test, y_pred)

print(f" MSE du modèle de régression linéaire : {score}")
print(f" RMSE du modèle de régression linéaire : {np.sqrt(score)}")
```

Here, we can observe that our linear model performs even worse than our naive model: $RMSE_{linear} = 9.15 \cdot 10^{-4}$ and $RMSE_{naive} = 9.09 \cdot 10^{-4}$.

The goal is to modify the data in order to be able to analyze it afterward.

Describing the data with other representations

BLabla

We will use the Dscribe package to describe the data with other representations.

In [ ]:
```python
from dscribe.descriptors import CoulombMatrix
from ase.io import read

def get_atoms_from_xyz(xyz_file_path):
    # Read the atomic coordinates from the XYZ file
    atoms = read(xyz_file_path, format='xyz', index=':')
    return atoms

MO2S4_atoms = get_atoms_from_xyz('data/MO2S4/TRAJEC_short.xyz')
```

Coulomb matrix

The Coulomb matrix encodes the atomic species and interatomic distances of a finite system in a pair-wise, two-body matrix inspired by the form of the Coulomb potential. The elements of this matrix are given by :

$$M_{ij} = \begin{cases} \frac{1}{2} Z_i^{2.4} & \text{if } i = j \\ \frac{Z_i Z_j}{|R_i - R_j|} & \text{if } i \neq j \end{cases}$$

where Z_i is the atomic number of atom i and $|R_i - R_j|$ is the Euclidian distance between atoms i and j.

In [ ]:
```
def get_coulomb_matrix(atoms):
    # Create a Coulomb matrix
    n_atom = len(atoms)
    cm = CoulombMatrix(n_atoms_max=n_atom, permutation="sorted_l2")
    coulomb_matrix = cm.create(atoms)
    coulomb_matrix_reshaped = coulomb_matrix.reshape((n_atom,n_atom))
    # Print the Coulomb matrix on a heatmap with values inside the cells
    plt.imshow(coulomb_matrix_reshaped, cmap="hot", interpolation="nearest")
    plt.colorbar()

    # Add a title
    plt.title(f"Coulomb matrix for {atoms.get_chemical_formula()}")
    plt.show()
    return coulomb_matrix

coulomb_matrix = get_coulomb_matrix(MO2S4_atoms[0])
```

We indeed succeed to form a matrix with the atomic species and interatomic distances of a finite system. I calculated the first two values of the matrix and they are correct.

Ewald sum matrix

Another type of representation is the Ewald sum matrix. It is a matrix representation of the Ewald sum of a periodic system. It is tho a computationally expensive representation.

In [ ]:
```
```