MoleKing: A Python Module for Theoretical Chemistry.

User Guide, Version 1.6.0

Universidade Federal de Goiás

LEEDMOL - Laboratory of Electronic Structure and Molecular Dynamics.

Contents

1 MoleKing: General Overview **1**

1.1 Disclaimer . . . 1

1.2 Classes Overview . . . 1

2 Installation **3**

2.1 Requirements . . . 3

2.2 Installation - Linux, Windows and MacOS . . . 3

2.3 Installation - From Source . . . 3

3 Usage **4**

3.1 PeriodicTable . . . 4

3.1.1 getAtomicNumber . . . 4

3.1.2 getAtomicMass . . . 5

3.1.3 getSymbol . . . 5

3.1.4 getCovalentRadii . . . 6

3.1.5 getColor . . . 6

3.1.6 getConstant . . . 7

3.1.7 getConversion . . . 7

3.2 Atom . . . 9

3.2.1 Creating an Atom object . . . 9

3.2.2 getAtomicMass, getAtomicSymbol, getAtomicNumber, getAtomicRadio 10

3.2.3 setX, setY, setZ, setNewPos . . . 10

3.2.4 setAtomicCharge . . . 11

3.2.5 getX, getY, getZ, getPos, getAtomicCharge . . . 12

3.2.6 translation . . . 12

3.2.7 Operators . . . 13

3.3 Molecule . . . 15

3.3.1 Creating a Molecule Object . . . 15

3.3.2 removeAtom . . . 16

3.3.3 getAtom . . . 17

3.3.4 setCharge, setMultiplicity and setVDWRation . . . 18

3.3.5 getCharge, getMultiplicity and getVDWRation . . . 19

3.3.6 Copy and Clear . . . 20
3.3.7 getMolecule . . . 20
3.3.8 getMassCenter . . . 21
3.3.9 getMM . . . 21
3.3.10 getIRCBonds . . . 22
3.3.11 getIRCAngles . . . 23
3.3.12 getIRCDihedral . . . 24
3.3.13 removeElement . . . 25
3.3.14 spinMolecule . . . 26
3.3.15 translation . . . 27
3.3.16 moveMassCenter . . . 28
3.3.17 moveTail . . . 28
3.3.18 bondLenght . . . 29
3.3.19 valenceAngle . . . 30
3.3.20 torsion . . . 30
3.3.21 toXYZ . . . 31
3.3.22 toGJF . . . 32
3.3.23 RMSD . . . 33
3.3.24 addChargePoints . . . 34
3.3.25 normChargePoints . . . 35
3.3.26 getChargePoints . . . 36
3.3.27 Operators . . . 37
3.4 SupraMolecule . . . 39
3.4.1 Creating a Supramolecule object . . . 39
3.4.2 addMolecule . . . 39
3.4.3 removeMolecule . . . 40
3.4.4 addAtomToMolecule . . . 41
3.4.5 removeAtom . . . 42
3.4.6 removeElement . . . 44
3.4.7 getSize . . . 45
3.4.8 getMolecule . . . 46
3.4.9 getCharge, setMultiplicity, getMultiplicity . . . 47

3.4.10 getMassCenter . . . 48
3.4.11 getSupraMM . . . 49
3.4.12 getIRCBonds . . . 49
3.4.13 getIRCBonds . . . 50
3.4.14 getIRCDihedrals . . . 51
3.4.15 spinSupraMolecule . . . 52
3.4.16 translation . . . 53
3.4.17 moveTail . . . 54
3.4.18 Operators . . . 55
3.5 ChargePoint . . . 57
3.5.1 Creating a ChargePoint object . . . 57
3.5.2 setX, setY, setZ, setNewPos . . . 57
3.5.3 getX, getY, getZ, getPos . . . 58
3.5.4 translation . . . 58
3.5.5 Operators . . . 59
3.6 G16LOGfile . . . 60
3.6.1 Loading a Gaussian 16 Log File . . . 60
3.6.2 getDate, getBasis, getMethod . . . 60
3.6.3 getMolecule . . . 61
3.6.4 getEnergy . . . 61
3.6.5 getDipole . . . 62
3.6.6 getOrbitals, getHOMO and getLUMO . . . 63
3.6.7 getVibFrequencies . . . 64
3.6.8 getZPE, getZPVE . . . 64
3.6.9 getH, getS, getG . . . 65
3.6.10 get_qEle . . . 66
3.6.11 get_qVib . . . 66
3.6.12 get_qRot . . . 67
3.6.13 get_qTrans . . . 67
3.6.14 get_qTot . . . 68
3.6.15 get_sigmaR . . . 68
3.6.16 getTransitions . . . 69

3.6.17 getNLOFrequency 70
3.6.18 getNLO 70
3.6.19 getAlpha 71
3.6.20 getBeta 72
3.6.21 getGamma 74
3.6.22 getLinkSize 75

1 MoleKing: General Overview

MoleKing is a Python package written in C++ with pybind11 Linkage created and maintained by LEEDMOL Research Group. It contains several useful classes for those who program python scripts aimed at computational chemistry. This package's main goal is to introduce chemistry concepts, such as Molecules, Atoms, and Geometries, to python, making programming more intuitive and understandable. Additionally, MoleKing is capable of reading and writing inputs and outputs files for Gaussian 16 package.

1.1 Disclaimer

MoleKing is still under development, and some classes may not be fully functional or may not have been tested. The user is encouraged to report any bugs or issues found. Any feedback is welcome and will be used to improve the package. Please, contact the authors for any questions or suggestions.

Pedro H. F. Matias: phfmatias@discente.ufg.br

Mateus R. Barbosa: mateus_barbosa@ufg.br

Rafael F. Veríssimo: rafaelfv27@discente.ufg.br

1.2 Classes Overview

Diving into MoleKing's classes, you'll find the following:

i) **PeriodicTable** - Contains all the information in a periodic table.

ii) **Atom** - Creates an atom object.

iii) **ChargePoint** - Creates a charge point variable type.

iv) **Molecule** - Creates a molecule object type.

v) **SupraMolecule** - Creates a set of molecules variable type.

vi) **Point** - Creates a point variable type.

vii) **SphericalCoords** - Allows the interchange between Cartesian and spherical coordinates.

viii) **Vector3D** - Creates a vector (x$\hat{i}$ + y$\hat{j}$ + z$\hat{k}$) variable type.

ix) **Matrix** - Creates a Matrix variable type.

x) **PovRay** - Creates a PovRay file from a molecule.

xi) **G16LOGfile** - Used to parse and extract information from Gaussian 16 log files.

xii) **Psi4OUTfile** - Used to parse and extract information from Psi4 output files.

2 Installation

2.1 Requirements

- C++11 or higher;
- CMake 3.15 or higher;
- libEigen;
- pybind11;
- PIP10 or higher;

2.2 Installation - Linux, Windows and MacOS

MoleKing is available on PyPI, so you can install it using pip:

```
pip3 install MoleKing
```

2.3 Installation - From Source

To install MoleKing from source, you need to clone the repository and run the following commands:

```
git clone https://github.com/Mateus-RB/MoleKing -recursive
pip3 install ./MoleKing
```

3 Usage

In this chapter, we will explore the various classes of MoleKing and learn how to use them effectively. MoleKing is a powerful tool that allows you to perform various tasks related to molecular calculations in chemistry. Whether you are a student, researcher, or professional in the field, understanding how to utilize the different classes of MoleKing will greatly enhance your productivity and accuracy.

We will begin by providing an overview of the main classes in MoleKing and their respective functionalities. Then, we will delve into each class individually, discussing their key features, methods, and best practices for usage. By the end of this chapter, you will have a comprehensive understanding of how to leverage MoleKing to perform complex mole calculations with ease. Let's get started!

3.1 PeriodicTable

The **PeriodicTable** class is a fundamental class of MoleKing that contains information about all elements. This class allows the user to access various properties of elements, such as: atomicNumber, AtomicMass,Symbol, covalentRadii and color.

3.1.1 getAtomicNumber

The **getAtomicNumber** method returns the atomic number of an element based on its symbol. This method takes the symbol of the element as a parameter and returns the corresponding atomic number.

Listing 1: **getAtomicNumber()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the atomic number of an element by symbol
atomic_number = pt.getAtomicNumber('H')
atomic_number2 = pt.getAtomicNumber('W')

print('AtomicNumber H :', atomic_number, '\nAtomicNumber 74
    :',atomic_number2)
```

Output:

```
AtomicNumber H : 1
AtomicNumber 74 : 74
```

3.1.2 getAtomicMass

The **getAtomicMass** method returns the atomic mass of an element based on its symbol. This method takes the symbol of the element as a parameter and returns the corresponding atomic mass.

Listing 2: **getAtomicMass()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the atomic mass of an element by symbol
atomic_mass = pt.getAtomicMass('H')
atomic_mass2 = pt.getAtomicMass('W')

print('AtomicMass H :', atomic_mass, '\nAtomicMass 74
    :',atomiatomic_mass2)
```

Output

```
AtomicMass H:  1.00794
AtomicMass 74: 183.84
```

3.1.3 getSymbol

The **getSymbol** method returns the symbol of an element based on its atomic number. This method takes the atomic number of the element as a parameter and returns the corresponding symbol.

Listing 3: **getSymbol()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the atomic symbol of an element by atomic number
atomic_symbol = pt.getSymbol(1)
atomic_symbol2 = pt.getSymbol(74)

print('AtomicSymbol 1 :', atomic_symbol, '\AtomicSymbol 74
    :',atomic_symbol2)
```

Output

```
    AtomicSymbol 1:  H
    AtomicSymbol 74: W
```

3.1.4 getCovalentRadii

The **getCovalentRadii** method returns the covalent radii of an element based on its symbol. This method takes the symbol of the element as a parameter and returns the corresponding covalent radii.

Listing 4: **getCovalentRadii()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the covalent radii of an element by symbol
covalenteRadii = pt.getCovalentRadii('H')
covalenteRadii2 = pt.getCovalentRadii('W')

print('CovalenteRadii H :', covalenteRadii, '\nCovalenteRadii W
    :',covalenteRadii2)
```

Output

```
CovalenteRadii H : 0.32
CovalenteRadii W : 1.37
```

3.1.5 getColor

The **getColor** method returns the color of an element based on its atomic number. This method takes the atomic number of the element as a parameter and returns the corresponding color.

Listing 5: **getColor()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the color of an element by symbol
colorAtom = pt.getColor('H')
colorAtom2 = pt.getColor('W')

print('colorAtom 1 (H):', colorAtom, '\ncolorAtom 74 (W):',colorAtom2)
```

Output

```
colorAtom 1 (H): FFFFFF
colorAtom 74 (W): 2194D6
```

3.1.6 getConstant

The **getConstant** method returns physical constants used in calculations. This method takes the name of the constant as a parameter and returns the corresponding value. The available constants are: Bohr radius ('a0_bohr') in Angstroms, electron charge ('qe_c') in Coulombs, electron charge in ESU ('qe_esu'), Planck's constant ('h') in Joule-secs, Pi ('PI') dimensionless, reduced Planck's constant ('hbar') in Joule-secs, Avogadro's number ('NA') dimensionless, Hartree energy ('hartree') in Joules, speed of light ('C') in cm/sec, Boltzmann constant ('k_B') in Joule/Kelvin, electron mass ('me') in kg, gas constant in cal/(mol*K) ('R_cal'), gas constant in J/(mol*K) ('R_J'), standard pressure ('P_0') in Pa.

Listing 6: **getConstant()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()

# Get the value of a physical constant
bohr_radius = pt.getConstant('a0_bohr')
electron_charge = pt.getConstant('qe_c')

print('Bohr radius (a0_bohr):', bohr_radius, 'Angstroms')
print('Electron charge (qe_c):', electron_charge, 'Coulombs')
```

Output

```
Bohr radius (a0_bohr): 0.52917721092 Angstroms
Electron charge (qe_c): 1.602176487e-19 Coulombs
```

3.1.7 getConversion

The **getConversion** method returns conversion factors used in calculations. This method takes the name of the conversion factor as a parameter and returns the corresponding value. The available conversion factors are: Hartree to eV ('Hartree_to_ev'), Hartree to KJ/mol ('Hartree_to_KJ'), Hartree to Kcal/mol ('Hartree_to_Kcal'), eV to Joule ('ev_to_Joule'), Kcal to KJ ('Kcal_to_KJ'), atm to Pa ('atm_to_Pa'), bar to Pa ('bar_to_Pa'), amu to kg ('amu_to_kg'), amu to kg/m$\hat{2}$ ('amu_to_kg_m2').

Listing 7: **getConversion()**

```
from MoleKing import PeriodicTable

# Create an instance of the PeriodicTable class
pt = PeriodicTable()
```

```
# Get the value of a conversion factor
hartree_to_ev = pt.getConversion('Hartree_to_ev')
hartree_to_KJ = pt.getConversion('Hartree_to_KJ')

print('Hartree to eV (Hartree_to_ev):', hartree_to_ev, 'eV')
print('Hartree to KJ/mol (Hartree_to_KJ):', hartree_to_KJ, 'KJ/mol')
```

Output

```
Hartree to eV (Hartree_to_ev): 27.2113845 eV
Hartree to KJ/mol (Hartree_to_KJ): 2625.49962 KJ/mol
```

3.2 Atom

The **Atom** class creates an Atom object. This class is fundamental to create the molecule object. The atom class contains a series of properties that can be accessed and modified. The Atom class methods include: getAtomicMass, getAtomicSymbol, getAtomicNumber, getAtomicRadio, setX, setY, setZ, setNewPos, setAtomicCharge, getX, getY, getZ, getPos, getAtomicCharge, translation and iterators.

3.2.1 Creating an Atom object

There are two ways to create an atom object. The first one, is using as parameter the atomic Number, position at X, Y and Z and atomic charge.

Listing 8: **Atom Object (I)**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)

# Print the atom object

print('Atom1: ',atom)
```

Output

```
Atom1:  Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000)
    Charge: -1.000000
```

And, the second way is using as parameter the atomic Symbol, position at X, Y and Z and atomic charge.

Listing 9: **Atom Object (II)**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom('F', 0, 0, 0, -1)

# Print the atom object

print('Atom1: ',atom)
```

Output

```
Atom1:  Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000)
    Charge: -1.000000
```

3.2.2 getAtomicMass, getAtomicSymbol, getAtomicNumber, getAtomicRadio

The methods getAtomicMass, getAtomicSymbol, getAtomicNumber and getAtomicRadio return the atomic mass, atomic symbol, atomic number and atomic radio of the atom object, respectively. It's closed to the periodic table.

Listing 10: **getAtomicMass(), getAtomicSymbol(), getAtomicNumber() and getAtomicRadio()**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)

# Print the atom object

atomicMass = atom.getAtomicMass()
atomicSymbol = atom.getAtomicSymbol()
atomicNumber = atom.getAtomicNumber()
atomicRadii = atom.getAtomicRadio()

print('Atomic Mass: ', atomicMass, '\nAtomic Symbol: ', atomicSymbol,
    '\nAtomic Number: ', atomicNumber, '\nAtomic Radii: ', atomicRadii)
```

Output

```
Atomic Mass:  18.998
Atomic Symbol:  F
Atomic Number:  9
Atomic Radii:  0.64
```

3.2.3 setX, setY, setZ, setNewPos

You can easily change the position of the atom object using the methods setX, setY, setZ and setNewPos. The setX, setY and setZ methods change the position of the atom object in the X, Y and Z coordinates, respectively. The setNewPos method changes the position of the atom object in the X, Y and Z coordinates at the same time.

Listing 11: **setX(), setY(), setZ() and setNewPos()**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)

# Printing the atom

print(atom)
```

```
# Modifying the atom's position

atom.setX(1)
atom.setY(2)
atom.setZ(3)

print(atom)

# Modifying the atom's position using setNewPos method.

atom.setPos(4, 5, 6)

print(atom)
```

Output

```
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (1.000000, 2.000000, 3.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (4.000000, 5.000000, 6.000000) Charge:
    -1.000000
```

3.2.4 setAtomicCharge

The setAtomicCharge method changes the atomic charge of the atom object.

Listing 12: **setAtomicCharge()**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)

# Printing the atom

print(atom)

# Modifying the atomic charge

atom.setAtomicCharge(0)

print(atom)
```

Output

```
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    0.000000
```

3.2.5 getX, getY, getZ, getPos, getAtomicCharge

The methods getX, getY and getZ return the position of the atom object in the X, Y and Z coordinates, respectively. The getPos method returns the position of the atom object in the X, Y and Z coordinates at the same time. The getAtomicCharge method returns the atomic charge of the atom object.

Listing 13: **getX(), getY(), getZ(), getPos() and getAtomicCharge()**

```
from MoleKing import Atom

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)

# Printing the atom

print(atom)

# Getting the atom's coordinates, and the atom's charge.

x = atom.getX()
y = atom.getY()
z = atom.getZ()
pos = atom.getPos()
charge = atom.getAtomicCharge()

print('X: ', x, '\nY: ', y, '\nZ: ', z, '\nPos: ', pos, '\nCharge: ',
    charge)
```

Output

```
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
X:  0.0
Y:  0.0
Z:  0.0
Pos:  [0.0, 0.0, 0.0]
Charge:  -1.0
```

3.2.6 translation

The translation method moves the atom object in the X, Y and Z coordinates. The translation method receives as parameters the Vector3D (see section X) object that represents the translation in the X, Y and Z coordinates.

Listing 14: **translation()**

```
from MoleKing import Atom, Vector3D

# Create an instance of the Atom class.
```

```
atom = Atom(9, 0, 0, 0, -1)

# Printing the atom

print(atom)

# Translating the atom

myVector = Vector3D([5.5,7.2,5.3])
atom.translation(myVector)

print(atom)
```

Output

```
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (5.500000, 7.200000, 5.300000) Charge:
    -1.000000
```

Obviously, you can use the setNewPos method to do the same thing. But this method is important to Molecule class.

3.2.7 Operators

You can compare two Atom objects using the operators == and !=.

Listing 15: **Iterators**

```
from MoleKing import Atom, Vector3D

# Create an instance of the Atom class.
atom = Atom(9, 0, 0, 0, -1)
atom2 = Atom(9, 1, 1, 1, -1)
atom3 = Atom(9, 0, 0, 0, -1)

print(atom)
print(atom2)
print(atom3)

# Using iterators to compare the atoms.

print(atom == atom2)
print(atom == atom3)
print(atom2 == atom3)
```

Output

```
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (1.000000, 1.000000, 1.000000) Charge:
    -1.000000
Element F (Z = 9) Cartesian pos: (0.000000, 0.000000, 0.000000) Charge:
    -1.000000
```

```
False
True
False
```

3.3 Molecule

The Molecule class is a container for Atom objects. It is the main class in MoleKing and is used to represent a molecule. The Molecule class provides various methods for manipulating and analyzing molecules, such as calculating the molecular weight, center of mass, and bond length. In this section, we will explore the key features of the Molecule class and learn how to use them effectively.

3.3.1 Creating a Molecule Object

To create a Molecule object, you need to create a empty Molecule object and add Atom objects to it. The Molecule class has a method called addAtom that allows you to add Atom objects to the molecule.

Listing 16: **Creating a Molecule Object (I)**

```
from MoleKing import Molecule, Atom

# Create a molecule

H2O = Molecule()

# Add atoms to the molecule

Oxygen = Atom('O', 0, 0, 0)
Hydrogen1 = Atom('H', 0.529, 0.529, 0)
Hydrogen2 = Atom('H', -0.529, -0.529, 0)

H2O.addAtom(Oxygen)
H2O.addAtom(Hydrogen1)
H2O.addAtom(Hydrogen2)

# Print the molecule

print(H2O)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

Or, you can create a Molecule object without instantiating the Atom objects first, by using the addAtom method with the atomic symbol, position at X, Y and Z and atomic charge as parameters, see the chapter 3.2 for more details.

Listing 17: **Creating a Molecule Object (II)**

```
from MoleKing import Molecule, Atom
```

```
# Create a molecule

H2O = Molecule()

# Add atoms to the molecule

H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule

print(H2O)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

3.3.2 removeAtom

You can remove an Atom from your Molecule object using the removeAtom method. This method takes the index of the Atom object to be removed as a parameter or the Atom object itself.

Listing 18: **removeAtom() (I)**

```
from MoleKing import Molecule

# Create a molecule

H2O = Molecule()

# Add atoms to the molecule

H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule

print(H2O)

H2O.removeAtom(0)

print(H2O)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Molecule H_{2}, with charge 0 and multiplicity 1
```

Listing 19: **removeAtom() (II)**

```
from MoleKing import Molecule, Atom

# Create a molecule

H2O = Molecule()

# Add atoms to the molecule

Oxygen = Atom('O', 0, 0, 0)
Hydrogen1 = Atom('H', 0.529, 0.529, 0)
Hydrogen2 = Atom('H', -0.529, -0.529, 0)

H2O.addAtom(Oxygen)
H2O.addAtom(Hydrogen1)
H2O.addAtom(Hydrogen2)

# Print the molecule

print(H2O)

H2O.removeAtom(Hydrogen2)

print(H2O)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Molecule O_{1}H_{1}, with charge 0 and multiplicity 1
```

3.3.3 getAtom

The getAtom method returns the Atom object at the specified index in the Molecule object.

Listing 20: **getAtom()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

x = H2O.getAtom(2)

print(x)
print(type(x))
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
['1', '0.529000', '0.529000', '0.000000']
<class 'list'>
```

Similarly, you can use the __getitem__ method to access the Atom object at the specified index in the Molecule object.

Listing 21: **__getItem__()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

x = H2O.getAtom(2)

print(x)
print(type(x))
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Element H (Z = 1) Cartesian pos: (-0.529000, -0.529000, 0.000000)
    Charge: 0.000000
<class 'MoleKing.Atom'>
```

3.3.4 setCharge, setMultiplicity and setVDWRation

The setCharge method changes the charge of the Molecule object, the setMultiplicity method changes the multiplicity of the Molecule object and the setVDWRation method changes the VDW ratio that will be considered in the calculation of what's considered a bond in the molecule (default is 1.3).

Listing 22: **setCharge(), setMultiplicity() and setVDWRation()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()
```

```
# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

H2O.setCharge(-1)
H2O.setMultiplicity(2)
H2O.setVDWRatio(1.3)

print(H2O)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Molecule O_{1}H_{2}, with charge -1 and multiplicity 2
```

3.3.5 getCharge, getMultiplicity and getVDWRation

The getCharge method returns the charge of the Molecule object, the getMultiplicity method returns the multiplicity of the Molecule object and the getVDWRation method returns the VDW ratio that will be considered in the calculation of what's considered a bond in the molecule.

Listing 23: **title**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

charge = H2O.getCharge()
multiplicity = H2O.getMultiplicity(2)
vdwR = H2O.getVDWRatio(1.3)

print('Charge:', charge, '\nMultiplicity:', multiplicity, '\nVDW
    Ratio:', vdwR)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Charge: 0.0
Multiplicity: 1
VDW Ratio: 1.3
```

3.3.6 Copy and Clear

The copy method returns a copy of the Molecule object, while the clear method removes all Atom objects from the Molecule object.

Listing 24: **copy() and clear()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

# Copying the molecule

H2O_copy = H2O.copy()

print(H2O_copy)

# Clear the molecule

H2O_copy = H2O_copy.clear()

print(H2O_copy)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
None
```

3.3.7 getMolecule

The getMolecule method returns a list of Atom objects in the Molecule object, it receives a boolean value that if True will return the Atom objects as a list of lists.

Listing 25: **getMolecule()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
```

```
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

x = H2O.getMolecule(True)

print(x)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
[['O', '0.000000', '0.000000', '0.000000'], ['H', '0.529000',
    '0.529000', '0.000000'], ['H', '-0.529000', '-0.529000', '0.000000']]
```

3.3.8 getMassCenter

The getMassCenter method returns the center of mass of the Molecule object.

Listing 26: **getMassCenter()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

# Get the mass center of the molecule

x = H2O.getMassCenter()

print(x)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Coords in Cartesian Space (x, y, z): (0.000000, 0.000000, 0.000000)
```

3.3.9 getMM

The getMM method returns the molecular mass of the Molecule object.

Listing 27: **getMM()**

```
from MoleKing import Molecule

# Create a molecule
H2O = Molecule()

# Add atoms to the molecule
H2O.addAtom('O', 0, 0, 0)
H2O.addAtom('H', 0.529, 0.529, 0)
H2O.addAtom('H', -0.529, -0.529, 0)

# Print the molecule
print(H2O)

# Get the mass center of the molecule

x = H2O.getMM()

print(x)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
18.0148
```

3.3.10 getIRCBonds

The getIRCBonds method calculates all the bonds in the Molecule object and return a list of lists with the indexes of the atoms that are bonded.

Listing 28: **getIRCBonds()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Print the molecule

print(Ethane)

# IRC Bond

x = Ethane.getIRCBonds()
```

```
print(x)
```

Output

```
Molecule C_{2}H_{6}, with charge 0 and multiplicity 1
[[0, 1], [0, 2], [0, 3], [0, 4], [4, 5], [4, 6], [4, 7]]
```

It's noteworthy that the getIRCBonds method is a zero-based index, so the first atom at your visualization software will be the atom 0 for the MoleKing. So, based on the output above, we have that atom 0[1] is bonded with the atoms 1,2,3,4 [2,3,4,5] and the atom 4[5] is bonded with the atoms 0,5,6,7 [0,5,6,7]. And, the Figure 1 shows the Ethane with atom indexes. So, we have carbon C1 bonded with hydrogen H2, H3, H4 and C2, and carbon C2 bonded with the hydrogen H5, H6, H7 and H8, as MoleKing output to us.

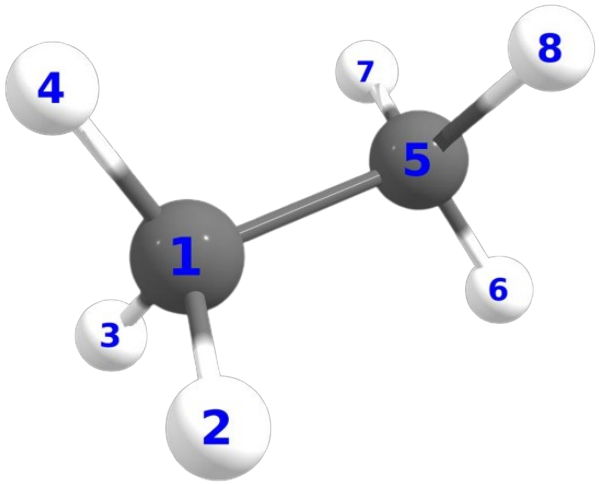

Figure 1: Ethane molecule and atom indexes

3.3.11 getIRCAngles

The getIRCAngles method calculates all the angles in the Molecule object and will return a list of lists with the indexes of the atoms that are bonded. It's closely related to the getIRCBonds method, so the output will be the same as the getIRCBonds method, but with the angles.

Listing 29: **getIRCAngles()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()
```

```
# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Print the molecule

print(Ethane)

# IRC Angles

x = Ethane.getIRCAngles()

print(x)
```

Output

```
[[1, 0, 2], [1, 0, 3], [1, 0, 4], [2, 0, 3], [2, 0, 4], [3, 0, 4], [0,
    4, 5], [0, 4, 6], [0, 4, 7], [5, 4, 6], [5, 4, 7], [6, 4, 7]]
```

So, looking at the Figure 1 and the MoleKing output we can see all angles presents in ethane.

Table 1: Angles in the Ethane molecule

$H_2 - C_1 - H_3$	$C_1 - C_5 - H_6$
$H_2 - C_1 - H_4$	$C_1 - C_5 - H_7$
$H_2 - C_1 - C_5$	$C_1 - C_5 - H_8$
$H_3 - C_1 - H_4$	$H_6 - C_5 - H_7$
$H_3 - C_1 - C_5$	$H_6 - C_5 - H_8$
$H_4 - C_1 - C_5$	$H_7 - C_5 - H_8$

3.3.12 getIRCDihedral

The getIRCDihedral method calculates all the dihedrals in the Molecule object and will return a list of lists with the indexes of the atoms that are bonded. It's closely related to the getIRCBonds and getIRCAngles methods, so the output will be the same as the getIRCBonds and getIRCAngles methods, but with the dihedrals.

Listing 30: **getIRCDihedral()**

```
from MoleKing import Molecule

# Create a molecule
```

```
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Print the molecule

print(Ethane)

# IRC dihedrals

x = Ethane.getIRCDihedral()

print(x)
```

Output

```
[[1, 0, 4, 5], [1, 0, 4, 6], [1, 0, 4, 7], [2, 0, 4, 5], [2, 0, 4, 6],
    [2, 0, 4, 7], [3, 0, 4, 5], [3, 0, 4, 6], [3, 0, 4, 7]]
```

So, looking at the Figure 1 and the MoleKing output we can see all dihedrals presents in ethane.

Table 2: Dihedrals in the Ethane molecule

$H_2 - C_1 - C_5 - H_6$	$H_3 - C_1 - C_5 - H_8$
$H_2 - C_1 - C_5 - H_7$	$H_4 - C_1 - C_5 - H_6$
$H_2 - C_1 - C_5 - H_8$	$H_4 - C_1 - C_5 - H_7$
$H_3 - C_1 - C_5 - H_6$	$H_4 - C_1 - C_5 - H_8$
$H_3 - C_1 - C_5 - H_7$	

3.3.13 removeElement

The removeElement method removes all Atom objects with the specified atomic symbol from the Molecule object.

Listing 31: **removeElement()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms
```

```
Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Print the molecule

print(Ethane)

# Remove the carbon atoms

Ethane.removeElement('C')

print(Ethane)
```

Output

```
Molecule C_{2}H_{6}, with charge 0 and multiplicity 1
Molecule H_{6}, with charge 0 and multiplicity 1
```

3.3.14 spinMolecule

The spinMolecule method rotates the Molecule object by the specified angle around the specified axis.

Listing 32: **spinMolecule()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Spining the molecule

Ethane.spinMolecule(59,'z')
Ethane.toXYZ('Ethane_spined.xyz')
```

Output

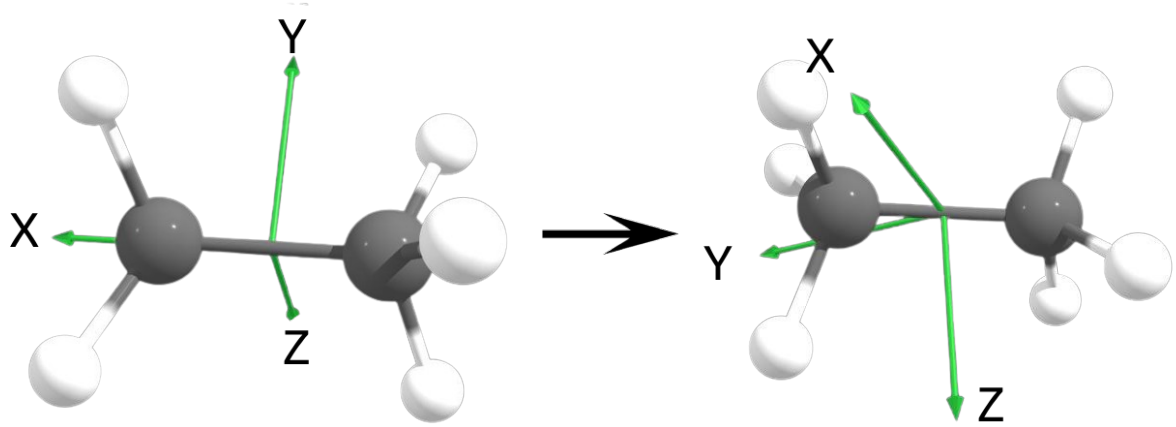

Figure 2: Ethane molecule spined by 59 degrees in the z axis.

3.3.15 translation

The translation method translates the Molecule object by the specified vector.

Listing 33: **translation()**

```
from MoleKing import Molecule, Vector3D

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Translate the molecule

Ethane.translation(Vector3D([0,2.2,0]))

for atom in Ethane:
    print(atom.getAtomicSymbol(), atom.getPos())
```

Output (Translated XYZ Coordinates)

```
C [-0.7480714, 1.9646532800000003, -0.34750096]
H [-0.39141698, 0.9558432700000001, -0.34750096]
H [-0.39139856, 2.46905147, -1.22115247]
H [-1.8180714, 1.9646664600000001, -0.34750096]
C [-0.23472918, 2.69060955, 0.909904]
H [0.8352708, 2.6904269800000002, 0.91000147]
H [-0.59122424, 3.6994758800000005, 0.90980642]
H [-0.59156144, 2.18632374, 1.78355529]
```

3.3.16 moveMassCenter

The moveMassCenter method moves the Molecule object mass center to a sspecified position.

Listing 34: **moveMassCenter()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Move the mass center of the molecule

Ethane.moveMassCenter(3.0, 3.0, 3.0)

for atom in Ethane:
    print(atom.getAtomicSymbol(), atom.getPos())
```

Output

```
C [2.74332889268153, 2.6370218881281966, 2.3712975310613116]
H [3.0999833126815304, 1.6282118781281965, 2.3712975310613116]
H [3.1000017326815303, 3.1414200781281965, 1.4976460210613116]
H [1.6733288926815302, 2.6370350681281964, 2.3712975310613116]
C [3.25667111268153, 3.3629781581281963, 3.6287024910613117]
H [4.32667109268153, 3.3627955881281966, 3.6287999610613118]
H [2.9001760526815303, 4.371844488128197, 3.628604911061312]
H [2.89983885268153, 2.8586923481281965, 4.502353781061312]
```

3.3.17 moveTail

The moveTail method moves the Molecule object so that the tail atom is at a specified position, without spinning the molecule. The code below is moving the fourth atom (C5) to the origin.

Listing 35: **moveTail()**

```
from MoleKing import Molecule
```

```
# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.91000147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Move an atom to specific coordinate.

Ethane.moveTail(4, 0.0, 0.0, 0.0)

for atom in Ethane:
    print(atom.getAtomicSymbol(), atom.getPos())
```

Output

```
C [-0.5133422200000001, -0.72595627, -1.2574049600000001]
H [-0.15668780000000002, -1.73476628, -1.2574049600000001]
H [-0.15666938, -0.22155808, -2.13105647]
H [-1.58334222, -0.72594309, -1.2574049600000001]
C [0.0, 0.0, 0.0]  # Tail Atom
H [1.06999998, -0.00018256999999999302, 9.746999999993289e-05]
H [-0.35649506, 1.00886633, -9.758000000004152e-05]
H [-0.35683226, -0.5042858100000001, 0.87365129]
```

3.3.18 bondLenght

The bondLength method calculates the bond length between two atoms in the Molecule object.

Listing 36: **bondLenght()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)
```

```
# Calculate the bond length between C1 and H2

bonds = Ethane.getIRCBonds()

print('Bond Lenght of C1-H2: {}'.format(Ethane.bondLength(0,1)))
```

Output

```
Bond Lenght of C1-H2: 1.0700000054120264
```

3.3.19 valenceAngle

The valenceAngle method calculates the valence angle between three atoms in the Molecule object.

Listing 37: **valenceAngle()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Calculate the valence angle between C1, H2 and H3

bonds = Ethane.getIRCAngles()

print('Angle of C1-H2-H3: {}'.format(Ethane.valenceAngle(0,1,2)))
```

Output

```
Angle of C1-H2-H3: 35.264398655465115
```

3.3.20 torsion

The torsion method calculates the torsion angle between four atoms in the Molecule object.

Listing 38: **torsion()**

```
    from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Move the mass center of the molecule

bonds = Ethane.getIRCDihedrals()

print('Torsion of C1-H2-H3-H4: {}'.format(Ethane.torsion(0,1,2,3)))
```

Output

```
    Torsion of C1-H2-H3-H4: -35.26438744831664
```

3.3.21 toXYZ

The toXYZ method writes the Molecule object to an XYZ file.

Listing 39: **toXYZ()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Creating a XYZ

Ethane.toXYZ(fileName='Ethane.xyz')
```

Output (XYZ File)

```
8
XYZ file generated by MoleKing!
C      -0.748071   -0.235347   -0.347501
H      -0.391417   -1.244157   -0.347501
H      -0.391399   0.269051    -1.221152
H      -1.818071   -0.235334   -0.347501
C      -0.234729   0.490610    0.909904
H      0.835271    0.490427    0.910015
H      -0.591224   1.499476    0.909806
H      -0.591561   -0.013676   1.783555
```

3.3.22 toGJF

The toGJF method writes the Molecule object to a Gaussian input file. This method receives the following parameters:

fileName = The name of the file that will be created.

method = The method that will be used in the Gaussian calculation.

basis = The basis set that will be used in the Gaussian calculation.

addKeywords = A list of additional keywords that will be added to the Gaussian input file.

midKeywords = A list of keywords when the molecule have charge points.

endKeywords = A list of keywords that will be added at the end of the Gaussian input file. Such as frequencies for NLO calculations.

charge = The charge of the molecule.

multiplicity = The multiplicity of the molecule.

zmatrix = A boolean value that if True will write the Gaussian input file in Z-matrix format.

EField = A list with the electtric field values. To use this option, zmatrix should be True.

bondFixer = A list of lists with the indexes of the atoms that will have their bond length fixed. To use this option, zmatrix should be True. With this option, the bond will be aligned with the z axis by default!

modHF = A int value to customize the %HF exchange in DFT functionals.

mem = The amount of memory to be used in the Gaussian calculation.

procs = The number of processors to be used in the Gaussian calculation.

Listing 40: **toGJF()**

```
from MoleKing import Molecule

# Create a molecule
Ethane = Molecule()
```

```
# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Creating a GJF

Ethane.toGJF(fileName='Ethane.gjf', method='CAM-B3LYP',
   basis='def2-tzvp', addKeywords='opt polar=gamma',
   endKeywords='1024nm', charge=0, multiplicity = 1, zmatrix=False)
```

Output

```
%chk=Ethane.chk
#p CAM-B3LYP/def2-tzvp opt polar=gamma

Gaussian job generated by MoleKing!

0 1
C     -0.748071   -0.235347   -0.347501
H     -0.391417   -1.244157   -0.347501
H     -0.391399   0.269051    -1.221152
H     -1.818071   -0.235334   -0.347501
C     -0.234729   0.490610    0.909904
H     0.835271    0.490427    0.910015
H     -0.591224   1.499476    0.909806
H     -0.591561   -0.013676   1.783555

 1024nm
```

3.3.23 RMSD

The **RMSD** class calculates the Root Mean Square Deviation (RMSD) between two molecules. The RMSD is a measure of the average distance between the atoms of two molecules. The RMSD is a useful metric for comparing the similarity between two molecules, the output is in angstrom. The RMSD is calculated as follows:

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2} \tag{1}$$

Listing 41: **RMSD()**

```
from MoleKing import Molecule
```

```
# Create a molecule
Ethane = Molecule()

# Add atoms

Ethane.addAtom('C',-0.74807140,-0.23534672,-0.34750096)
Ethane.addAtom('H',-0.39141698,-1.24415673,-0.34750096)
Ethane.addAtom('H',-0.39139856, 0.26905147,-1.22115247)
Ethane.addAtom('H',-1.81807140,-0.23533354,-0.34750096)
Ethane.addAtom('C',-0.23472918, 0.49060955, 0.90990400)
Ethane.addAtom('H', 0.83527080, 0.49042698, 0.9100147)
Ethane.addAtom('H',-0.59122424, 1.49947588, 0.90980642)
Ethane.addAtom('H',-0.59156144,-0.01367626, 1.78355529)

# Second molecule

Ethane2 = Molecule()

# Add atoms to Ethane2
Ethane2.addAtom('C', -1.28384854, 0.53177193, -1.31448286)
Ethane2.addAtom('H', -0.59159746, 0.77777553, -2.09240983)
Ethane2.addAtom('H', -2.16363105, 1.13208868, -1.41695657)
Ethane2.addAtom('H', -1.54765762, -0.50274146, -1.38580478)
Ethane2.addAtom('C', -0.63425528, 0.80262715, 0.05528608)
Ethane2.addAtom('H', -0.37025980, 1.83709771, 0.12653987)
Ethane2.addAtom('H', -1.32657537, 0.55680588, 0.83320903)
Ethane2.addAtom('H', 0.24542068, 0.20216667, 0.15783813)


# Calculating the RMSD

RMSD = Ethane.RMSD(Ethane2)

print('RMSD = ', RMSD)
```

Output

```
RMSD =  1.725673157839484
```

3.3.24 addChargePoints

The addChargePoints method adds charge points to the Molecule object. You cand do it in two ways,passing X,Y,Z coordinates and the charge values or passing a ChargePoint object, see subsection **3.5**.

Listing 42: **addChargePoints() - (I)**

```
from MoleKing import Molecule

# Create a molecule
Water = Molecule()

# Add atoms
```

```
# Add atoms to Water
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)

# Adding chargePoints

Water.addChargePoints(-2.68764706,1.76141277,0.01027907,-0.895322)
Water.addChargePoints(-3.46724854,1.20236374,0.04613585,0.447661)
Water.addChargePoints(-2.95502854,2.68316798,-0.01150351,0.447661)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1 and with 3 charge
    points
```

Listing 43: **addChargePoints() - (II)**

```
from MoleKing import Molecule, ChargePoint

# Create a molecule
Water = Molecule()

# Add atoms

# Add atoms to Water
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)

# Adding chargePoints

cp1 = ChargePoint(-2.68764706,1.76141277,0.01027907,-0.895322)
cp2 = ChargePoint(-3.46724854,1.20236374,0.04613585,0.447661)
cp3 = ChargePoint(-2.95502854,2.68316798,-0.01150351,0.447661)

Water.addChargePoints(cp1)
Water.addChargePoints(cp2)
Water.addChargePoints(cp3)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1 and with 3 charge
    points
```

3.3.25 normChargePoints

The normChargePoints method normalizes charge points in the Molecule object. The normalization is done by dividing the charge of each charge point by a int value.

Listing 44: **normChargePoints()**

```
from MoleKing import Molecule, ChargePoint

# Create a molecule
Water = Molecule()

# Add atoms

# Add atoms to Water
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)

# Adding chargePoints

Water.addChargePoints(-2.68764706,1.76141277,0.01027907,-0.895322)
Water.addChargePoints(-3.46724854,1.20236374,0.04613585,0.447661)
Water.addChargePoints(-2.95502854,2.68316798,-0.01150351,0.447661)

# Normalize the chargePoints

Water.normChargePoints(10)

print(Water.getChargePoints())
```

Output

```
[['-2.687647', '1.761413', '0.010279', '-0.089532'], ['-3.467249',
    '1.202364', '0.046136', '0.044766'], ['-2.955029', '2.683168',
    '-0.011504', '0.044766']]
```

3.3.26 getChargePoints

The getChargePoints method returns a list with the charge points in the Molecule object.

Listing 45: **getChargePoints()**

```
from MoleKing import Molecule, ChargePoint

# Create a molecule
Water = Molecule()

# Add atoms

# Add atoms to Water
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)

# Adding chargePoints

Water.addChargePoints(-2.68764706,1.76141277,0.01027907,-0.895322)
Water.addChargePoints(-3.46724854,1.20236374,0.04613585,0.447661)
Water.addChargePoints(-2.95502854,2.68316798,-0.01150351,0.447661)

# Get the chargePoints
```

```
print(Water.getChargePoints())
```

Output

```
[['-2.687647', '1.761413', '0.010279', '-0.089532'], ['-3.467249',
    '1.202364', '0.046136', '0.044766'], ['-2.955029', '2.683168',
    '-0.011504', '0.044766']]
```

3.3.27 Operators

The Molecule object is an iterable object, so you can iterate over the atoms in the molecule using a for loop, compare two molecules and etc.

Listing 46: **Operators in Molecule**

```
from MoleKing import Molecule, Atom

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# Add atoms to Water

A1 = Atom("O", -0.68023254, 0.42596771, -0.00557622)
A2 = Atom("H", 0.27976746, 0.42596771, -0.00557622)
A3 = Atom("H", -1.00068713, 1.33090354, -0.00557622)

Water.addAtom(A1)
Water.addAtom(A2)
Water.addAtom(A3)
Water2.addAtom(A1)
Water2.addAtom(A2)
Water2.addAtom(A3)

# Iterators

#Loop over atoms

for atom in Water:
    print(atom)

# Compare molecules

print('Water == Water2?')

Water2.addAtom(A3)

print('Water == Water2?')
print('Water != Water2?')
print('Water > Water2?')
print('Water < Water2?')
```

Output

```
Element O (Z = 8) Cartesian pos: (-0.680233, 0.425968, -0.005576)
   Charge: 0.000000
Element H (Z = 1) Cartesian pos: (0.279767, 0.425968, -0.005576)
   Charge: 0.000000
Element H (Z = 1) Cartesian pos: (-1.000687, 1.330904, -0.005576)
   Charge: 0.000000
Water == Water2? True
Water == Water2? False
Water != Water2? True
Water > Water2? False
Water < Water2? True
```

3.4 SupraMolecule

The Supramolecule class represents a collection of molecules objects. It is a container class that allows you to store and manipulate multiple molecules at once. This class is useful when you need to work with a group of molecules that are related in some way, such as in crystallography or molecular dynamics simulations.

3.4.1 Creating a Supramolecule object

To create a supramolecule object, you need to create a empty supramolecule then add Molecules to it.

Listing 47: **Creating a SupraMolecule Object**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule
Water, Water2 = Molecule(), Molecule()

# addingAtoms to Molecules

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Print SupraMolecule

print(SupraMolecule)
```

Output

```
Supramolecule:
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

3.4.2 addMolecule

Listing 48: **Creating a SupraMolecule Object**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule
Water, Water2 = Molecule(), Molecule()

# addingAtoms to Molecules

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Print SupraMolecule

print(SupraMolecule)
```

Output

```
Supramolecule:
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

3.4.3 removeMolecule

The removeMolecule method allows you to remove a molecule from the supramolecule by its index.

Listing 49: **removeMolecule()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)
```

```
# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Removing Molecules from SupraMolecule

SupraMolecule.removeMolecule(0)

print(SupraMolecule)
```

Output

```
Supramolecule:
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

3.4.4 addAtomToMolecule

The addAtomToMolecule method allows you to add an atom to a specific molecule in the supramolecule. There's two ways to use this method, the first one is by passing the molecule index and the atom object, and the second one is by passing the molecule index and the x,y and z coordinates of the atom.

Listing 50: **addAtomToMolecule (I)**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# Adding Atoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)
```

```
# Adding a extra hydrogen atom to Molecule 1 in SupraMolecule

SupraMolecule.addAtomToMolecule(0,"H", 0.27976746, 0.42596771,
   -0.00557622)
```

Output

```
Supramolecule:
    Molecule O_{1}H_{3}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

Listing 51: **addAtomToMolecule (II)**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# Adding Atoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Adding a extra hydrogen atom to Molecule 2 in SupraMolecule

A1 = Atom('H',-3.46724854,1.20236374,0.04613585)
SupraMolecule.addAtomToMolecule(1,A1)

print(SupraMolecule)
```

Output

```
Supramolecule:
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{3}, with charge 0 and multiplicity 1
```

3.4.5 removeAtom

The removeAtom method allows you to remove an atom from a specific molecule in the supramolecule. There's two ways to use this method, the first one is by passing the molecule

index and the atom index, and the second one is by passing the molecule index and the atom object.

Listing 52: **removeAtom() (I)**

```
from MoleKing import SupraMolecule, Molecule, Atom

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Removing the first atom for the first molecule

SupraMolecule.removeAtom(0, 0)

print(SupraMolecule)
```

Output

```
Supramolecule: 2 molecules with a total charge of 0.
    Molecule H_{2}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

Listing 53: **removeAtom() (II)**

```
from MoleKing import SupraMolecule, Molecule, Atom

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Atom2 = Atom('H',-3.46724854,1.20236374,0.04613585)
```

```
Water2.addAtom(Atom2)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Removing the second atom for the second molecule

SupraMolecule.removeAtom(Atom2)

print(SupraMolecule)
```

Output

```
Supramolecule: 2 molecules with a total charge of 0.
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{1}, with charge 0 and multiplicity 1
```

3.4.6 removeElement

The removeElement method allows you to remove all atoms of a specific element from a specific molecule in the supramolecule, or for the entire supramolecule.

Listing 54: **removeElement() (I)**

```
from MoleKing import SupraMolecule, Molecule, Atom

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)
```

```
# Removing all Hydrogens from Molecule 1

SupraMolecule.removeElement(0, 'H')

print(SupraMolecule)
```

Output

```
Supramolecule: 2 molecules with a total charge of 0.
    Molecule O_{1}, with charge 0 and multiplicity 1
    Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

Listing 55: **removeElement() (II)**

```
from MoleKing import SupraMolecule, Molecule, Atom

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Removing all Hydrogens from the entire SupraMolecule

SupraMolecule.removeElement('H')

print(SupraMolecule)
```

Output

```
Supramolecule: 2 molecules with a total charge of 0.
    Molecule O_{1}, with charge 0 and multiplicity 1
    Molecule O_{1}, with charge 0 and multiplicity 1
```

3.4.7 getSize

The getSize method allows you to retrieve the number of molecules in the supramolecule.

Listing 56: **getSize()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting the size of SupraMolecule

print('Size of SupraMolecule = ',SupraMolecule.getSize())
```

Output

```
Size of SupraMolecule =  2
```

3.4.8 getMolecule

The getMolecule method allows you to retrieve a molecule from the supramolecule by its index.

Listing 57: **getMolecule()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)
```

```
# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting a Molecule from SupraMolecule

Water = SupraMolecule.getMolecule(0)

print(Water)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
```

3.4.9 getCharge, setMultiplicity, getMultiplicity

The getCharge method allows you to retrieve the total charge of the supramolecule. The setMultiplicity method allows you to set the multiplicity of the supramolecule. The getMultiplicity method allows you to retrieve the multiplicity of the supramolecule.

Listing 58: **getCharge(), setMultiplicity(), getMultiplicity()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)


Water.setCharge(-1)
Water2.setCharge(3)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)
```

```
# Setting the Multiplicity and Getting the Charge and Multiplicity

SupraMolecule.setMultiplicity(1)

print('Total Charge:', SupraMolecule.getCharge())
print('Multiplicity:', SupraMolecule.getMultiplicity())
```

Output

```
Total Charge: 2.0
Multiplicity: 1
```

3.4.10 getMassCenter

The getMassCenter method allows you to retrieve the mass center of the supramolecule.

Listing 59: **getMassCenter()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting the Center of Mass

print('COM = ', SupraMolecule.getMassCenter())
```

Output

```
COM =  Coords in Cartesian Space (x, y, z): (-1.695338, 1.129152,
    0.002745)
```

3.4.11 getSupraMM

The getSupraMM method allows you to retrieve the total mass of the supramolecule.

Listing 60: **getSupraMM()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting the MOlecular Mass of the SupraMolecule

print('supraMM = ', SupraMolecule.getSupraMM())
```

Output

```
supraMM =  36.0296
```

3.4.12 getIRCBonds

The getIRCBonds method allows you to retrieve all the bonds for each molecule in the supramolecule.

Listing 61: **getIRCBonds()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule
```

```
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)


# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting the Bonds of the SupraMolecule

bonds = SupraMolecule.getIRCBonds()

print('Bonds: {} \n'.format(bonds))

for i in range(len(bonds)):
    for j in bonds[i]:
        molecule = SupraMolecule.getMolecule(i)
        atom1 = molecule[j[0]]
        atom2 = molecule[j[1]]
        print('Bonds in Molecule {}: {}_{}-{}_{}'.format(i+1,
            atom1.getAtomicSymbol(), j[0]+1, atom2.getAtomicSymbol(),
            j[1]+1))
```

Output

```
Bonds: [[[0, 1], [0, 2]], [[0, 1], [0, 2]]]

Bonds in Molecule 1: O_1-H_2
Bonds in Molecule 1: O_1-H_3
Bonds in Molecule 2: O_1-H_2
Bonds in Molecule 2: O_1-H_3
```

3.4.13 getIRCBonds

The getIRCBonds method allows you to retrieve all the bonds for each molecule in the supramolecule.

Listing 62: **getIRCBonds()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule
```

```
Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Getting the angles of the SupraMolecule

angles = SupraMolecule.getIRCAngles()

print('Angles: {} \n'.format(angles))


for i in range(len(angles)):
    for j in angles[i]:
        molecule = SupraMolecule.getMolecule(i)
        atom1 = molecule[j[0]]
        atom2 = molecule[j[1]]
        print('Angles in Molecule {}: {}_{}-{}_{}-{}_{}'.format(i+1,
            atom1.getAtomicSymbol(), j[0]+1, atom2.getAtomicSymbol(),
            j[1]+1, atom1.getAtomicSymbol(), j[2]+1))
```

Output

```
Angles: [[[1, 0, 2]], [[1, 0, 2]]]

Angles in Molecule 1: H_2-O_1-H_3
Angles in Molecule 2: H_2-O_1-H_3
```

3.4.14 getIRCDihedrals

The getIRCDihedrals method allows you to retrieve all the dihedrals for each molecule in the supramolecule.

Listing 63: **getIRCDihedrals()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Ethane, Ethane2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule
```

```
# addingAtoms to SupraMolecule

Ethane.addAtom('C', -3.07089564, 0.75444199, 0.06986466)
Ethane.addAtom('H', -2.71424121, -0.25436801, 0.06986466)
Ethane.addAtom('H', -2.71422279, 1.25884018, -0.80378684)
Ethane.addAtom('H', -4.14089564, 0.75445517, 0.06986466)
Ethane.addAtom('C', -2.55755342, 1.48039826, 1.32726963)
Ethane.addAtom('H', -1.48755344, 1.48021569, 1.32736710)
Ethane.addAtom('H', -2.91404847, 2.48926459, 1.32717205)
Ethane.addAtom('H', -2.91438567, 0.97611246, 2.20092092)
Ethane2.addAtom('C', -0.75061471, -2.64684470, -0.50535552)
Ethane2.addAtom('H', -0.39396029, -3.65565470, -0.50535552)
Ethane2.addAtom('H', -0.39394187, -2.14244651, -1.37900703)
Ethane2.addAtom('H', -1.82061471, -2.64683151, -0.50535552)
Ethane2.addAtom('C', -0.23727249, -1.92088842, 0.75204944)
Ethane2.addAtom('H', 0.83272749, -1.92107099, 0.75214691)
Ethane2.addAtom('H', -0.59376755, -0.91202209, 0.75195186)
Ethane2.addAtom('H', -0.59410475, -2.42517423, 1.62570073)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Ethane)
SupraMolecule.addMolecule(Ethane2)

# Getting the angles of the SupraMolecule

dihedrals = SupraMolecule.getIRCDihedrals()

print('Dihedrals: {} \n'.format(dihedrals))
```

Output

```
Dihedrals: [[[1, 0, 4, 5], [1, 0, 4, 6], [1, 0, 4, 7], [2, 0, 4, 5],
    [2, 0, 4, 6], [2, 0, 4, 7], [3, 0, 4, 5], [3, 0, 4, 6], [3, 0, 4,
    7]], [[1, 0, 4, 5], [1, 0, 4, 6], [1, 0, 4, 7], [2, 0, 4, 5], [2, 0,
    4, 6], [2, 0, 4, 7], [3, 0, 4, 5], [3, 0, 4, 6], [3, 0, 4, 7]]]
```

3.4.15 spinSupraMolecule

The spinSupraMolecule method allows you to rotate the supramolecule by a specific angle.

Listing 64: **spinSupraMolecule()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()
```

```
# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Spining SupraMolecule

SupraMolecule.spinSupraMolecule(59,'z')
```

Output

To a more visual representation of this method, please look at the subsection **3.3.14**, the method works the same way for the SupraMolecule class.

3.4.16 translation

The translation method allows you to translate the supramolecule by a specific distance in the x, y and z axis.

Listing 65: **translation()**

```
from MoleKing import SupraMolecule, Molecule, Vector3D

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule
```

```
SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Translate SupraMolecule

SupraMolecule.translation(Vector3D([0,2.2,0]))
```

Output

To a more visual representation of this method, please look at the subsection **3.3.15**, the method works the same way for the SupraMolecule class.

3.4.17 moveTail

The moveTail method moves the a molecule in the object so that the tail atom is at the specified position, without spinning the molecule. The below code is moving the second atom for the first molecule to the position (0,0,0). All the molecules in the supramolecule are moved to follow the same rule.

Listing 66: **moveTail()**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2 = Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)

# Create a SupraMolecule

SupraMolecule = SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)

# Move the second atom in molecule 1 to origin

SupraMolecule.moveTail(0, 1, 0.0, 0.0, 0.0)

# Printing the first molecule positions

for mol in SupraMolecule:
    for atom in mol:
        print(atom.getAtomicSymbol(), atom.getPos())
```

Output

```
O [-0.96, 0.0, 0.0]
H [0.0, 0.0, 0.0]
H [-1.28045459, 0.90493583, 0.0]
O [-2.96741452, 1.33544506, 0.01585529]
H [-3.747016, 0.77639603, 0.05171207]
H [-3.234796, 2.25720027, -0.00592729]
```

3.4.18 Operators

The Molecule object is an iterable object, so you can iterate over the atoms in the molecule using a for loop, compare two supramolecules and etc.

Listing 67: **Operators**

```
from MoleKing import SupraMolecule, Molecule

# Create a molecule

Water, Water2, Ethane = Molecule(), Molecule(), Molecule()

# addingAtoms to SupraMolecule

Water.addAtom("O", -0.68023254, 0.42596771, -0.00557622)
Water.addAtom("H", 0.27976746, 0.42596771, -0.00557622)
Water.addAtom("H", -1.00068713, 1.33090354, -0.00557622)
Water2.addAtom('O',-2.68764706,1.76141277,0.01027907)
Water2.addAtom('H',-3.46724854,1.20236374,0.04613585)
Water2.addAtom('H',-2.95502854,2.68316798,-0.01150351)
Ethane.addAtom('C', -3.07089564, 0.75444199, 0.06986466)
Ethane.addAtom('H', -2.71424121, -0.25436801, 0.06986466)
Ethane.addAtom('H', -2.71422279, 1.25884018, -0.80378684)
Ethane.addAtom('H', -4.14089564, 0.75445517, 0.06986466)
Ethane.addAtom('C', -2.55755342, 1.48039826, 1.32726963)
Ethane.addAtom('H', -1.48755344, 1.48021569, 1.32736710)
Ethane.addAtom('H', -2.91404847, 2.48926459, 1.32717205)
Ethane.addAtom('H', -2.91438567, 0.97611246, 2.20092092)

# Create a SupraMolecule

SupraMolecule, SupraMolecule2, SupraMolecule3 = SupraMolecule(),
    SupraMolecule(), SupraMolecule()

# Add Molecules to SupraMolecule

SupraMolecule.addMolecule(Water)
SupraMolecule.addMolecule(Water2)
SupraMolecule2.addMolecule(Water)
SupraMolecule2.addMolecule(Water2)
SupraMolecule3.addMolecule(Water)
SupraMolecule3.addMolecule(Ethane)

#Loop over molecules

for mol in SupraMolecule:
```

```
    print(mol)

# Compare SupraMolecules

print(SupraMolecule == SupraMolecule2)

SupraMolecule2.addMolecule(Water2)

print('SupraMolecule == SupraMolecule?', SupraMolecule ==
    SupraMolecule2)
print('SupraMolecule2 < SupraMolecule?', SupraMolecule2 < SupraMolecule)
print('SupraMolecule2 > SupraMolecule?', SupraMolecule2 > SupraMolecule)
print('SupraMolecule != SupraMolecule2?', SupraMolecule !=
    SupraMolecule2)
print('SupraMolecule == SupraMolecule3?', SupraMolecule ==
    SupraMolecule3)
print('SupraMolecule < SupraMolecule3?', SupraMolecule < SupraMolecule3)
print('SupraMolecule > SupraMolecule3?', SupraMolecule > SupraMolecule3)
print('SupraMolecule3 != SupraMolecule?', SupraMolecule3 !=
    SupraMolecule)
```

Output

```
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
Molecule O_{1}H_{2}, with charge 0 and multiplicity 1
SupraMolecule == SupraMolecule2? True
SupraMolecule2 < SupraMolecule? False
SupraMolecule2 > SupraMolecule? False
SupraMolecule != SupraMolecule2? True
SupraMolecule == SupraMolecule3? True
SupraMolecule < SupraMolecule3? False
SupraMolecule > SupraMolecule3? False
SupraMolecule3 != SupraMolecule? False
SupraMolecule3 == SupraMolecule? True
```

3.5 ChargePoint

The **ChargePoint** class creates a ChargePoint object. This class is fundamental to create the molecule object. The ChargePoint class contains a series of properties that can be accessed and modified. The ChargePoint class methods include: setX, setY, setZ, setNewPos, setCharge, getX, getY, getZ, getPos, getCharge, translation and iterators.

3.5.1 Creating a ChargePoint object

To create a ChargePoint object, you have to pass the x, y, z coordinates and the charge of the point. The following code snippet shows how to create a ChargePoint object:

Listing 68: **Creating a ChargePoint object**

```
from MoleKing import ChargePoint

# Create a ChargePoint
CP = ChargePoint(0.0, 0.0, 0.0, -1.0)

print(CP)
```

Output

```
Charge -1.000000 Cartesian pos: (0.000000, 0.000000, 0.000000)
```

3.5.2 setX, setY, setZ, setNewPos

You can easily modify the x, y, z coordinates of a ChargePoint object using the setX, setY, setZ, and setNewPos methods. The setX, setY, and setZ methods take a single argument, which is the new value of the x, y, or z coordinate, respectively. The setNewPos method takes three arguments, which are the new x, y, and z coordinates, respectively. The following code snippet demonstrates how to use these methods:

Listing 69: **setX(), setY(), setZ(), setNewPos()**

```
from MoleKing import ChargePoint

# Create a ChargePoint
CP = ChargePoint(0.0, 0.0, 0.0, -1)

# Changing X,Y,Z

CP.setX(1.0)
CP.setY(1.0)
CP.setZ(1.0)
```

```
print(CP)

# Changing all valies at once with seNewPos

CP.setNewPos(2.0, 2.0, 2.0)

print(CP)
```

Output

```
Charge -1.000000 Cartesian pos: (1.000000, 1.000000, 1.000000)
Charge -1.000000 Cartesian pos: (2.000000, 2.000000, 2.000000)
```

3.5.3 getX, getY, getZ, getPos

You can access the x, y, z coordinates of a ChargePoint object using the getX, getY, and getZ methods. The getPos method returns a tuple containing the x, y, and z coordinates. The following code snippet demonstrates how to use these methods:

Listing 70: **getX(), getY(), getZ(), getPos()**

```
from MoleKing import ChargePoint

# Create a ChargePoint
CP = ChargePoint(0.0, 0.0, 0.0, -1)

# Getting X,Y,Z

print(CP.getX())
print(CP.getY())
print(CP.getZ())

# Getting all valies at once with seNewPos

print(CP.getPos())
```

Output

```
0.0
0.0
0.0
[0.0, 0.0, 0.0]
```

3.5.4 translation

The translation method allows you to translate a ChargePoint object by a given vector. The translation method takes a single argument, which is a Vector3d object representing the translation vector. The following code snippet demonstrates how to use the translation method:

Listing 71: **translation()**

```
from MoleKing import ChargePoint, Vector3D

# Create a ChargePoint
CP = ChargePoint(0.0, 0.0, 0.0, -1)

# Translate the ChargePoint


CP.translation(Vector3D([5.5,7.2,5.3]))

# Getting all valies at once with seNewPos

print(CP.getPos())
```

Output

```
[5.5, 7.2, 5.3]
```

3.5.5 Operators

The ChargePoint class also supports the following operators:

Listing 72: **Operators**

```
from MoleKing import ChargePoint
# Create a ChargePoint
CP = ChargePoint(0.0, 0.0, 0.0, -1)
CP2 = ChargePoint(0.0, 0.0, 0.0, -1)
CP3 = ChargePoint(0.0, 0.0, 0.0, 1)

# Using operators

print('CP == CP2?', CP == CP2)
print('CP != CP2?', CP != CP2)
print('CP > CP3?', CP > CP3)
print('CP < CP3?', CP < CP3)
```

Output

```
CP == CP2? True
CP != CP2? False
CP > CP3? False
CP < CP3? True
```

3.6 G16LOGfile

The **G16LOGfile** class is designed to parse and extract information from Gaussian 16 log files. Gaussian 16 is a widely used computational chemistry software package that performs quantum mechanical calculations on molecular systems. The log files generated by Gaussian 16 contain detailed information about the calculations performed, including the input parameters, molecular geometry, electronic structure, and vibrational frequencies. The arguments of G16LOGfile are: polarAsw if you want to extract NLO properties, tdAsw if you want to extract TD-DFT properties, thermoAsw if you want to extract thermodynamic properties, cpAsw if any charge point is included in the log file, and link to specify the link calculation to be extracted.

3.6.1 Loading a Gaussian 16 Log File

To load a Gaussian 16 log file into MoleKing, you can use the **G16LOGfile** class constructor as follows:

Listing 73: **Loading a Gaussian 16 Log File**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object

log = G16LOGfile('MK_test1.log')
```

Observation: The G16LOGfile class by default extracts the log from the last calculation in the log file. If you want to extract a specific link calculation, you can use

Listing 74: **Extracting a Specific Link Calculation**

```
log = G16LOGfile('MK_test1.log', link=N)
```

where **N** is the link number you want to extract. Default is **-1**.

3.6.2 getDate, getBasis, getMethod

The **G16LOGfile** class provides methods to extract key information from the log file, such as the date of the calculation, the basis set used, and the computational method employed. These methods are **getDate()**, **getBasis()**, and **getMethod()** respectively.

Listing 75: **getDate(), getBasis(), getMethod()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object

log = G16LOGfile('MK_test1.log')

# Getting information from the log file

print('Date: ', log.getDate())
print('Method: ', log.getMethod())
print('Basis Set: ', log.getBasis())
```

Output

```
Date:  2024
Method:  B3LYP
Basis Set:  6-31+G(d) (6D, 7F)
```

3.6.3 getMolecule

The **getMolecule()** method of the **G16LOGfile** class is used to extract the molecular structure from the log file. This method returns a **Molecule** object that contains the atomic coordinates and other relevant information about the molecule, all functionality is inherited from the Molecule (**Chapter 3.3**) class.

Listing 76: **getMolecule()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_test1.log')

# Getting the molecule object from the log file
molecule = log.getMolecule()

print(molecule)
```

Output

```
Molecule C_{2}H_{6}O_{1}, with charge 0 and multiplicity 1
```

3.6.4 getEnergy

The **getEnergy()** method retrieves the total energy of the system from the log file. This energy is typically reported in Hartrees (Eh).

Listing 77: **getEnergy()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_test1.log')

% # Getting the molecule's energy from the log file
energy = log.getEnergy()

print(energy)
```

Output

```
-155.045622249
```

3.6.5 getDipole

The **getDipole()** method retrieves the dipole moment of the molecule from the log file.The method can return the dipole moment in three components (x, y, z) or as a total value. By default, it returns the total dipole moment.

Listing 78: **getDipole()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_test1.log')

# Getting the dipoles from the log file

dipoles = log.getDipole()

print("Dipoles:", dipoles)


x = log.getDipole(axis='x')
y = log.getDipole(axis='y')
z = log.getDipole(axis='z')
tot = log.getDipole(axis='tot')

print("\nDipole x:", x)
print("Dipole y:", y)
print("Dipole z:", z)
print("Dipole tot:", tot)
```

Output

```
Dipoles: 1.8039

Dipole x: -0.1293
Dipole y: 1.7992
Dipole z: 0.0001
Dipole tot: 1.8039
```

3.6.6 getOrbitals, getHOMO and getLUMO

The **getOrbitals()** method retrieves the molecular orbitals from the log file. Specifically, creates an dictionary with occupied and unoccupied orbitals, where the keys are the list of orbitals energies.

The **getHOMO()** and **getLUMO()** methods specifically return the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO) energies, respectively.

Listing 79: **getHOMO(), getLUMO()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_test1.log')

# Getting the orbitals from the log file
orbitals = log.getOrbitals()

homo = log.getHOMO()
lumo = log.getLUMO()

print('Orbitals:', orbitals)
print('HOMO:', homo)
print('LUMO:', lumo)

print('---')

print(float(orbitals['Occupied'][-1]) == homo)
print(float(orbitals['Unoccupied'][0]) == lumo)

print('---')

# You also can get homo-N like:

homo_1 = log.getHOMO(-1)
print('HOMO-1:', homo_1)

# And, lumo+N as well:

lumo_1 = log.getLUMO(+1)
print('LUMO+1:', lumo_1)
```

Output

```
Orbitals: {'Occupied': ['-19.15718', '-10.23836', '-10.17958',
    '-1.02281', '-0.75283', '-0.61571', '-0.52222', '-0.46892',
    '-0.40559', '-0.38730', '-0.36940', '-0.33757', '-0.27816'],
    'Unoccupied': ['0.00023', '0.02614', '0.03779', '0.04800',
    '0.07426', '0.07999', '0.08533', '0.13299', '0.15742', '0.18058',
    '0.18313', '0.19259', '0.21084', '0.23109', '0.25023', '0.26419',
    '0.30001', '0.31295', '0.31768', '0.36373', '0.62556', '0.64103',
    '0.65169', '0.70417', '0.72437', '0.74782', '0.91228', '0.94528',
    '0.97550', '1.01220', '1.05976', '1.09405', '1.11677', '1.20246',
    '1.20873', '1.28433', '1.45251', '1.47389', '1.52570', '1.67889',
```

```
    '1.73367', '1.80630', '1.92824', '1.96333', '2.03823', '2.17763',
    '2.24275', '2.33522', '2.39177', '2.40356', '2.52308', '2.63690',
    '2.88109', '4.04214', '4.21316', '4.43564']}
HOMO: -0.27816
LUMO: 0.00023
---
True
True
---
HOMO-1: -0.33757
LUMO+1: 0.02614
```

3.6.7 getVibFrequencies

The **getVibFrequencies()** method retrieves the vibrational frequencies of the molecule from the log file. It returns a list of vibrational frequencies in wavenumbers (cm^{-1}). ThermoAsw must be set to True when creating the G16LOGfile object to use this method.

Listing 80: **getVibFrequencies()**

```python
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the vibrational frequencies

vibFrequencies = log.getVibFrequencies()

print("Vibrational Frequencies (cm^-1):", vibFrequencies)
```

Output

```
    Vibrational Frequencies (cm^-1):
    [-650.1271, 47.1151, 55.8582, 91.514, 116.1731, 138.6816, 165.968,
       198.4175, 208.2555, 232.0198, 241.5873, 261.5832, 299.4168,
       334.0419, 341.0172, 346.3377, 390.4843, 431.3451, 475.8158,
       504.3182, 539.6479, 582.1662, 593.4334, 637.9197, 646.2021,
       663.6157, 729.1147, 752.4012, 830.2735, 918.3505, 932.7132,
       1015.6307, 1094.3852, 1131.0221, 1190.1923, 1275.0766,
       1328.4787, 1375.122, 1441.8576, 1520.1579, 1589.8923, 1637.2384,
       1682.5668, 1813.9011, 3576.585, 3685.2502, 3689.2468, 3709.9677]
```

3.6.8 getZPE, getZPVE

The **getZPE()** method retrieves the zero-point energy (ZPE) of the molecule from the log file. The ZPE is the lowest possible energy that a quantum mechanical physical system may have, and it is a crucial component in thermodynamic calculations. About the ZPVE, the

getZPVE() method retrieves the zero-point vibrational energy (ZPVE) of the molecule from the log file. The ZPVE is the vibrational contribution to the zero-point energy and is calculated based on the vibrational frequencies of the molecule.

Listing 81:

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Zero Point Energy and Zero Point Vibrational Energy

ZPE = log.getZPE()
ZPVE = log.getZPVE()

print("ZPE:", ZPE)
print("ZPVE:", ZPVE)
```

Output

```
ZPE: -1946.420138
ZPVE: 64.99171
```

3.6.9 getH, getS, getG

The **getH()**, **getS()**, and **getG()** methods retrieve the enthalpy (H), entropy (S), and Gibbs free energy (G) of the molecule from the log file, respectively. These thermodynamic properties are essential for understanding the behavior of molecules in various chemical processes. The units are the same as in Gaussian 16 output.

Listing 82:

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Enthalpy, Entropy, and Gibbs Free Energy

H = log.getH()
S = log.getS()
G = log.getG()

print(f'Enthalpy (H): ', H)
print(f'Entropy (S): ', S)
print(f'Gibbs Free Energy (G): ', G)
```

Output

```
Enthalpy (H):  -1946.405323
Entropy (S):  119.997
Gibbs Free Energy (G):  -1946.462337
```

3.6.10 get_qEle

The **get_qEle()** method retrieves electronic partition function (qEle) from the log file, defined as:

$$q_e le = \omega_0 \tag{2}$$

where, ω_0 is the ground state degeneracy, or simply the multiplicity of the system.

Listing 83: **get_qEle()**

```python
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Electronic Partition Function
qEle = log.get_qEle()

print(f'Electronic Partition Function (qEle): ', qEle)
```

Output

```
Electronic Partition Function (qEle):  2.0
```

3.6.11 get_qVib

The **get_qVib()** method calculates vibrational partition function (qVib), defined as:

$$q_{vib} = \frac{1}{1 - e^{-\theta_{vib}/T}} \tag{3}$$

where, θ_{vib} is the characteristic vibrational temperature, and T is the temperature in Kelvin.

Listing 84: **get_qVib()**

```python
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Vibrational Partition Function
qVib = log.get_qVib()
```

```
print(f'Vibrational Partition Function (qVib): ', qVib)
```

Output

```
Vibrational Partition Function (qVib): 25190.276062320867
```

3.6.12 get_qRot

The **get_qRot()** method calculates rotational partition function (qRot), defined as, if the molecule is linear:

$$q_{rot} = \frac{T}{\sigma_r \theta_{rot}} \tag{4}$$

where, σ_r is the rotational symmetry number, θ_{rot} is $h^2/8\pi^2 I k_B$, with I being the moment of inertia. And for non-linear molecules:

$$q_{rot} = \sqrt{\frac{\pi}{\sigma_r}} \left(\frac{T^{3/2}}{(\theta_A \theta_B \theta_C)^{1/2}} \right) \tag{5}$$

where, θ_A, θ_B and θ_C are the characteristic rotational temperatures for each principal axis of rotation.

Listing 85: **get_qRot()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Rotational Partition Function
qRot = log.get_qRot()

print(f'Rotational Partition Function (qRot): ', qRot)
```

Output

```
Rotational Partition Function (qRot):  3156417.206378763
```

3.6.13 get_qTrans

The **get_qTrans()** method calculates translational partition function (qTrans) from, defined as:

$$q_{trans} = \left(\frac{2\pi m k_B T}{h^2}\right)^{3/2} \frac{K_b T}{P} \tag{6}$$

Listing 86: **get_qTrans()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Translational Partition Function
qTrans = log.get_qTrans()

print(f'Translational Partition Function (qTrans): ', qTrans)
```

Output

```
Translational Partition Function (qTrans):  165487549.49042565
```

3.6.14 get_qTot

The **get_qTot()** method calculates total partition function (qTot) and is defined as:

$$q_{tot} = q_{ele} \cdot q_{vib} \cdot q_{rot} \cdot q_{trans} \tag{7}$$

Listing 87: **get_qTot()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Total Partition Function
qTotal = log.get_qTot()

print(f'Total Partition Function (qTotal): ', qTotal)
```

Output

```
Total Partition Function (qTotal):  2.631616797820357e+19
```

3.6.15 get_sigmaR

The **get_sigmaR()** method retrieves the rotational symmetry number (σ_r) of the molecule from the log file. The rotational symmetry number is a factor that accounts for the indistinguishability of identical orientations of a molecule due to its symmetry.

Listing 88: **get_sigmaR()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the Rotational Symmetry Number
sigmaR = log.get_sigmaR()
print(f'Rotational Symmetry Number (sigmaR): ', sigmaR)
```

Output

```
Rotational Symmetry Number (sigmaR):  1.0
```

3.6.16 getTransitions

The **getTransitions()** method retrieves the electronic transitions of the molecule from the log file. It's noteworthy that this method is only available if the log file contains information about electronic transitions, which is typically the case for TD-DFT calculations, and you have to set tdAsw=True when creating the G16LOGfile object.

Listing 89: **getTransitions()**

```
from MoleKing import G16LOGfile
from pandas import DataFrame

# Creating a G16LOGfile object
log = G16LOGfile('MK_TD.log', tdAsw=True)

# Getting the transitions from the log file

transitions = log.getTransitions()

print(transitions)

# Easily convert the transitions to a pandas DataFrame for better
    visualization

df = DataFrame(transitions)
```

Output

```
 2: {'Energy': 7.3821, 'Oscillation_Strength': 0.0092, 'Wavelength':
    167.95},
 3: {'Energy': 7.6797, 'Oscillation_Strength': 0.0281, 'Wavelength':
    161.44},
 4: {'Energy': 7.9066, 'Oscillation_Strength': 0.0339, 'Wavelength':
    156.81},
 5: {'Energy': 8.1203, 'Oscillation_Strength': 0.0012, 'Wavelength':
    152.68},
 6: {'Energy': 8.6663, 'Oscillation_Strength': 0.0001, 'Wavelength':
    143.06},
```

```
 7: {'Energy': 8.8991, 'Oscillation_Strength': 0.0001, 'Wavelength':
    139.32},
 8: {'Energy': 9.0347, 'Oscillation_Strength': 0.0293, 'Wavelength':
    137.23},
 9: {'Energy': 9.0829, 'Oscillation_Strength': 0.0041, 'Wavelength':
    136.5},
 10: {'Energy': 9.1299, 'Oscillation_Strength': 0.0037, 'Wavelength':
    135.8},
 11: {'Energy': 9.2907, 'Oscillation_Strength': 0.1242, 'Wavelength':
    133.45},
 12: {'Energy': 9.513, 'Oscillation_Strength': 0.0431, 'Wavelength':
    130.33}}
```

3.6.17 getNLOFrequency

The **getNLOFrequency()** method retrieves the frequency of NLO properties from the log file. It's noteworthy that this method is only available if the log file contains information about NLO properties, which is typically the case for ONL calculations, and you have to set polarAsw=True when creating the G16LOGfile object.

Listing 90: **getNLOFrequency()**

```
from MoleKing import G16LOGfile
from pandas import DataFrame

# Creating a G16LOGfile object
log = G16LOGfile('MK_ONL.log', polarAsw=True)

# Getting the frequency of NLO properties

frequencies = log.getNLOFrequency()

print(frequencies)
```

Output

```
[0.0, 1024.0]
```

3.6.18 getNLO

The **getNLO()** method retrieves the nonlinear optical (NLO) properties of the molecule from the log file. Specifically, it will create a list with every NLO property available in the log file.

Listing 91: **getNLO()**

```
from MoleKing import G16LOGfile
from pandas import DataFrame

# Creating a G16LOGfile object
log = G16LOGfile('MK_ONL.log', polarAsw=True)

# Getting the NLO properties as text list

nlo_properties = log.getNLO()

print(nlo_properties)
```

Output

```
[' Electric dipole moment (input orientation):\n (Debye = 10**-18
   statcoulomb cm , SI units = C m)\n                    (au)
   (Debye)          (10**-30 SI)\n   Tot         0.709706D+00
   0.180389D+01       0.601714D+01\n   x          -0.508833D-01
   -0.129332D+00      -0.431406D+00\n   y           0.707880D+00
   0.179925D+01       0.600166D+01\n   z           0.485943D-04
   0.123514D-03       0.411999D-03\n\n Dipole polarizability, Alpha
   (input orientation).\n (esu units = cm**3 , SI units = C**2 m**2
   J**-1)\n Alpha(0;0):\n                (au)             (10**-24 esu)
        (10**-40 SI)\n   iso         0.297748D+02       0.441217D+01
   0.490920D+01\n   aniso       0.445391D+01       0.660001D+00
   0.734350D+00\n   xx          0.325089D+02       0.481733D+01
   0.536000D+01\n   yx          0.108865D-01       0.161321D-02
   0.179494D-02\n   yy          0.294106D+02       0.435819D+01
   0.484915D+01\n   zx         -0.113263D-03      -0.167839D-04
   -0.186746D-04\n   zy         -0.625463D-04      -0.926841D-05
   -0.103125D-04\n   zz          0.274049D+02       0.406099D+01 .... ]
```

3.6.19 getAlpha

The **getAlpha()** method retrieves the polarizability (γ) from the log file and returns a dictionary containing the tensor components. Three unit systems are available: **au**, **SI**, and **esu**, which can be specified using the **unit** keyword (default: **esu**). You can also define the tensor orientation via the **orientation** keyword, with options **input** or **dipole** (default: **dipole**). Additionally, the frequency values for γ can be controlled with the **frequency** keyword, which must be a list starting at **0.0** followed by user-defined values as specified in the input file (default: **[0.0]**). Note: if **esu** units are selected, the raw output will be an unscaled value (e.g., **2.0** instead of 2.0×10^{-24}). It is the user's responsibility to apply the correct scaling factor.

Listing 92: **getAlpha()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_ONL.log', polarAsw=True)
```

```
# Getting the polarizability of the molecule

alpha = log.getAlpha()
print('Polarizability:', alpha)

alpha_yy = alpha['yy']
print(f"Alpha yy: {alpha_yy}")

alpha_yy_si = log.getAlpha(unit='SI')['yy']
print(f"Alpha yy in SI units: {alpha_yy_si}")

alpha_yy_dipole_si = log.getAlpha(unit='SI', orientation='input')['yy']
print(f"Alpha yy in SI units (input orientation): {alpha_yy_dipole_si}")

# Getting the polarizability in different frequencies

alpha_0 = log.getAlpha(frequency=0)
print(f"Alpha at frequency 0: {alpha_0}")
alpha_1024 = log.getAlpha(frequency=1024)
print(f"Alpha at frequency 1024: {alpha_1024}")
```

Output

```
Polarizability: {'aniso': 0.660001, 'iso': 4.41217, 'xx': 4.81132,
    'yx': 0.0539501, 'yy': 4.06487, 'zx': -0.0311584, 'zy': -0.00221109,
    'zz': 4.36032}
Alpha yy: 4.06487
Alpha yy in SI units: 4.52277
Alpha yy in SI units (dipole orientation): 4.84915
Alpha at frequency 0: {'aniso': 0.660001, 'iso': 4.41217, 'xx':
    4.81132, 'yx': 0.0539501, 'yy': 4.06487, 'zx': -0.0311584, 'zy':
    -0.00221109, 'zz': 4.36032}
Alpha at frequency 1024: {'aniso': 0.66405, 'iso': 4.44508, 'xx':
    4.84608, 'yx': 0.0542981, 'yy': 4.09481, 'zx': -0.0316665, 'zy':
    -0.00224696, 'zz': 4.39435}
```

3.6.20 getBeta

The **getBeta()** method retrieves the first hyperpolarizability (γ) from the log file and returns a dictionary containing the tensor components. Three unit systems are available: **au**, **SI**, and **esu**, which can be specified using the **unit** keyword (default: **esu**). You can also define the tensor orientation via the **orientation** keyword, with options **input** or **dipole** (default: **dipole**). Additionally, the frequency values for γ can be controlled with the **frequency** keyword, which must be a list starting at **0.0** followed by user-defined values as specified in the input file (default: **[0.0]**). Note: if **esu** units are selected, the raw output will be an unscaled value (e.g., **2.0** instead of 2.0×10^{-30}). It is the user's responsibility to apply the correct scaling factor.

When running a frequency-dependent calculation, two tensor forms are available: $\beta(-\omega;\omega,0)$ and $\beta(-2\omega;\omega,\omega)$. By default, the first form is used when a frequency is specified. To compute

the second form, corresponding to the generalized second harmonic generation (BSHG), set the keyword **BSHG** to **True**.

Listing 93: **getBeta()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_ONL.log', polarAsw=True)

# Getting the first hyperpolarizability of the molecule

beta = log.getBeta()
print('Beta:', beta)

beta_xxx = beta['xxx']
print(f"Beta xxx: {beta_xxx}")

beta_xxx_si = log.getBeta(unit='SI')['xxx']
print(f"Beta xxx in SI units: {beta_xxx_si}")

beta_xxx_dipole_si = log.getBeta(unit='SI', orientation='Input')['xxx']
print("Beta xxx in SI units (input orientation):", beta_xxx_dipole_si)

beta_0 = log.getBeta(frequency=0)
print(f"Beta at frequency 0: {beta_0}")
beta_1024 = log.getBeta(frequency=1024)
print(f"Beta at frequency 1024: {beta_1024}") #this is Beta(-w;w,0)

# Getting dynamic Beta(-2w;w,w)

print('\n -----')

beta_2www = log.getBeta(frequency=1024, BSHG=True)
print(beta_2www)
```

Output

```
Beta xxx: 0.225847
Beta xxx in SI units: 0.083821
Beta xxx in SI units (input orientation): 0.0685653
Beta at frequency 0: {'_|_(z)': 0.0702808, 'x': 1.49073, 'xxx':
    0.225847, 'xxy': 0.00182195, 'xxz': 0.196172, 'y': 0.107238, 'yxy':
    0.0999813, 'yxz': 0.0115403, 'yyy': 0.0216215, 'yyz': 0.0366772,
    'z': 1.05421, 'zxz': 0.17108, 'zyz': 0.0123026, 'zzz': 0.118555,
    '||': 0.365794, '||(z)': 0.210842}
Beta at frequency 1024: {'_|_(z)': 0.0734347, 'x': 1.53781, 'xxx':
    0.232508, 'xxy': 0.00180764, 'xxz': 0.199309, 'y': 0.110626, 'yxx':
    0.00184035, 'yxy': 0.103399, 'yxz': 0.0114503, 'yyx': 0.102946,
    'yyy': 0.0223287, 'yyz': 0.0410696, 'z': 1.08547, 'zxx': 0.198916,
    'zxy': 0.0116255, 'zxz': 0.177068, 'zyx': 0.0116254, 'zyy':
    0.0382521, 'zyz': 0.0127331, 'zzx': 0.17641, 'zzy': 0.0126858,
    'zzz': 0.123584, '||': 0.377113, '||(z)': 0.217094}

 -----

{'_|_(z)': 0.0734893, 'x': 1.64006, 'xxx': 0.246883, 'xyx': 0.00180556,
    'xyy': 0.111304, 'xzx': 0.205515, 'xzy': 0.0114029, 'xzz': 0.19082,
    'y': 0.117984, 'yxx': 0.00191071, 'yyx': 0.109847, 'yyy': 0.0238626,
```

```
    'yzx': 0.0114025, 'yzy': 0.0479546, 'yzz': 0.0137219, 'z': 1.15425,
    'zxx': 0.20429, 'zyx': 0.0119756, 'zyy': 0.0387973, 'zzx': 0.188797,
    'zzy': 0.0135763, 'zzz': 0.134741, '||': 0.401796, '||(z)': 0.23085}
```

3.6.21 getGamma

The **getGamma()** method retrieves the second hyperpolarizability (γ) from the log file and returns a dictionary containing the tensor components. Three unit systems are available: **au**, **SI**, and **esu**, which can be specified using the **unit** keyword (default: **esu**). You can also define the tensor orientation via the **orientation** keyword, with options **input** or **dipole** (default: **dipole**). Additionally, the frequency values for γ can be controlled with the **frequency** keyword, which must be a list starting at **0.0** followed by user-defined values as specified in the input file (default: **[0.0]**). Note: if **esu** units are selected, the raw output will be an unscaled value (e.g., **2.0** instead of 2.0×10^{-36}). It is the user's responsibility to apply the correct scaling factor.

When running a frequency-dependent calculation, two tensor forms are available: $\gamma(-\omega;\omega,0,0)$ and $\gamma(-2\omega;\omega,\omega,0)$. By default, the first form is used when a frequency is specified. To compute the second form, corresponding to the generalized second harmonic generation (GSHG), set the keyword **GSHG** to **True**.

Listing 94: **getGamma()**

```python
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_ONL.log', polarAsw=True)

# Getting the second hyperpolarizability of the molecule

gamma = log.getGamma()
print('Gamma:', gamma)

gamma_xxxx = gamma['xxxx']
print(f"Gamma xxxx: {gamma_xxxx}")

gamma_xxxx_si = log.getGamma(unit='SI')['xxxx']
print(f"Gamma xxxx in SI units: {gamma_xxxx_si}")

gamma_xxxx_dipole_si = log.getGamma(unit='SI',
    orientation='Input')['xxxx']
print("Gamma xxxx in SI units (input orientation):",
    gamma_xxxx_dipole_si)

gamma_0 = log.getGamma(frequency=0)
print(f"Gamma at frequency 0: {gamma_0}")
gamma_1024 = log.getGamma(frequency=1024)
print(f"Gamma at frequency 1024: {gamma_1024}") #this is Gamma(-w;w,0)
```

```
# Getting dynamic Gamma(-2w;w,w)

print('\n -----')

gamma_2www = log.getGamma(frequency=1024, GSHG=True)
print(gamma_2www)
```

Output

```
Gamma xxxx: 2.99534
Gamma xxxx in SI units: 3.7082
Gamma xxxx in SI units (input orientation): 3.14301
Gamma at frequency 0: {'_|_': 0.72518, 'xxxx': 2.99534, 'xxyx':
    0.104339, 'xxyy': 0.520744, 'xxzx': -0.457873, 'xxzy': -0.0385817,
    'xxzz': 1.02341, 'yxyy': 0.0265938, 'yxzy': 0.0367705, 'yxzz':
    0.0199885, 'yyyy': 1.18302, 'yyzy': 0.00846118, 'yyzz': 0.746448,
    'zxzz': -0.355587, 'zyzz': -0.0255402, 'zzzz': 2.11812, '||':
    2.17554}
Gamma at frequency 1024: {'_|_': 0.754765, 'xxxx': 3.11699, 'xxyx':
    0.107881, 'xxyy': 0.54467, 'xxzx': -0.478299, 'xxzy': -0.0406166,
    'xxzz': 1.06112, 'yxxx': 0.107271, 'yxyx': 0.545175, 'yxyy':
    0.0285823, 'yxzx': -0.040885, 'yxzy': 0.0405385, 'yxzz': 0.0190583,
    'yyxx': 0.553112, 'yyyx': 0.0291932, 'yyyy': 1.22816, 'yyzx':
    0.0442818, 'yyzy': 0.00957753, 'yyzz': 0.797021, 'zxxx': -0.476261,
    'zxyx': -0.0407366, 'zxyy': 0.0431694, 'zxzx': 1.06499, 'zxzy':
    0.0201882, 'zxzz': -0.369086, 'zyxx': -0.0406264, 'zyyx': 0.041633,
    'zyyy': 0.0096546, 'zyzx': 0.020188, 'zyzy': 0.785254, 'zyzz':
    -0.0265101, 'zzxx': 1.06888, 'zzyx': 0.0200544, 'zzyy': 0.790996,
    'zzzx': -0.369774, 'zzzy': -0.0265599, 'zzzz': 2.21049, '||':
    2.27096}

 -----

{'_|_': 0.817672, 'xxxx': 3.38358, 'xxxy': 0.115431, 'xxxz': -0.523226,
    'xyxx': 0.114684, 'xyxy': 0.598719, 'xyxz': -0.0455649, 'xyyx':
    0.590734, 'xyyy': 0.0325053, 'xyyz': 0.0451705, 'xzxx': -0.520963,
    'xzxy': -0.0453825, 'xzxz': 1.14749, 'xzyx': -0.0462165, 'xzyy':
    0.0549643, 'xzyz': 0.0177862, 'xzzx': 1.15218, 'xzzy': 0.0206471,
    'xzzz': -0.397409, 'yxxx': 0.112617, 'yxxy': 0.599868, 'yxxz':
    -0.0465578, 'yyxx': 0.618188, 'yyxy': 0.034573, 'yyxz': 0.0590154,
    'yyyx': 0.0353201, 'yyyy': 1.32687, 'yyyz': 0.012378, 'yzxx':
    -0.045784, 'yzxy': 0.0489566, 'yzxz': 0.0177856, 'yzyx': 0.0605772,
    'yzyy': 0.0124704, 'yzyz': 0.900964, 'yzzx': 0.0206463, 'yzzy':
    0.866049, 'yzzz': -0.0285449, 'zxxx': -0.51392, 'zxxy': -0.0459029,
    'zxxz': 1.16058, 'zyxx': -0.0452546, 'zyxy': 0.0591306, 'zyxz':
    0.0217409, 'zyyx': 0.0501097, 'zyyy': 0.0127461, 'zyyz': 0.859303,
    'zzxx': 1.16489, 'zzxy': 0.0200596, 'zzxz': -0.399881, 'zzyx':
    0.0200594, 'zzyy': 0.886899, 'zzyz': -0.0287236, 'zzzx': -0.400881,
    'zzzy': -0.0287958, 'zzzz': 2.41437, '||': 2.4825}
```

3.6.22 getLinkSize

The **getLinkSize()** method retrieves the size of the linked files associated with the log file. This is particularly useful for large calculations that are split across multiple files, and if you're using MoleKing as backend for other applications.

Listing 95: **getLinkSize()**

```
from MoleKing import G16LOGfile

# Creating a G16LOGfile object
log = G16LOGfile('MK_Thermo.log', thermoAsw=True)

# Getting the size of linked files
link_size = log.getLinkSize()
print(f'Link Size: ', link_size)
```

Output

```
Link Size:  2
```