UNIVERSITAT
ROVIRA i VIRGILI

Marc Nieto Rodríguez

DEVELOPMENT OF A PYTHON TOOLKIT FOR MOLECULE MANIPULATION

Bachelor's Thesis

Directed by Dr. Coen de Graaf

Chemistry Degree

Physical and Inorganic Chemistry Department

Tarragona

June 2022

- **Acknowledgments:**

  I would like to thank Dr. Coen de Graaf, director, and supervisor of this project, for giving me the opportunity to learn programming skills that have led to the development of this work. I am beyond thankful for the patience showed and his time dedicated towards me and this thesis. I would also like to thank Yannick Roselló and Aitor Sanchez, PhD students of the Quantum Chemistry research group, that at some point have helped me to resolve some problems encountered.

- **Summary:**

English

The presented work focuses on the development of a Python toolkit for molecule manipulation with the aim to be applied in the near future in Theoretical Chemistry research groups.

The toolkit was started with the elaboration of two different, well-founded methods, designed to rotate the coordinates of a certain molecule to get a match with those of a target one, that could be identical or non-identical to the studied compound. The rotations were applied using quaternions, a particular mathematical entity which is introduced in the foundations of this thesis.

Furthermore, a module being able to rotate *p*- and *d*-orbital functions was also coded following a recurrence procedure based on deriving three-dimensional rotation matrices into five-dimensional ones.

Catalan

El treball presentat s'enfoca en el desenvolupament d'un conjunt d'eines Python per a la manipulació de molècules amb l'objectiu de ser aplicat en un futur proper en grups de recerca de química teòrica.

El conjunt d'eines es va iniciar amb l'elaboració de dos mètodes diferents, i ben fonamentats, dissenyats per fer girar les coordenades d'una determinada molècula i aconseguir una superposició amb les d'una altra, anomenada molècula diana. Aquesta pot ser idèntica o semblant al compost estudiat. Les rotacions es van aplicar mitjançant quaternions, una entitat matemàtica particular que s'introdueix en els fonaments d'aquesta tesi.

A més, també es va codificar un mòdul capaç de girar les funcions dels orbitals p i d seguint un procediment basat en obtenir matrius penta-dimensionals derivant matrius de rotació tridimensionals.

- **Index:**

1 Objectives:

The aim of this work was to initiate the development of a toolkit for molecule manipulation using Python as programming language. Starting with relatively simple operations such as translations and rotations, the module is meant to be extended in the future with more elaborate functionalities such as atom substitutions, group additions, automatic generation of input files for electronic structure calculations. This extended manipulation toolkit has the potential to become a powerful tool in the daily research practice of Theoretical Chemistry groups.

2 Technical aspects:

This thesis has been supervised by Professor Coen de Graaf of the Quantum Chemistry Group[1] of the Physical and Inorganic Chemistry department of the Universitat Rovira i Virgili's (URV).

The investigation of this group focusses on the usage of computational tools to tackle chemical problems of different research fields which include organometallic complexes, polyoxometalates, fullerenes, study of photochemical reactions and the modelling of homogenous and heterogenous catalytic processes, among others.

Dr. Coen de Graaf became an ICREA Research Professor of the Quantum Chemistry group after his post-doctoral studies back in 2005. Over the last few years, his research has mainly been motivated by the development of new computational schemes based on non-orthogonal configuration interaction to explore alternatives to existing theoretical methods.

3 Introduction:

3.1 Python:

Python is a high-level programming language that can be employed for building software in a large scope of application fields. The founding-father, Guido Van Rossum, published the first ever version (0.9.0) back in 1991 after having worked on it since the late 1980s[2]. After thirty years of innovation and several version releases, it is considered the most popular programming language ahead of Java and C according to the TIOBE index[3].

The enterprise that holds the property rights is Python Software Foundation[4], a non-profit corporation that, among other labors, administrates an approved open-source license which makes Python free to access for everyone. This is one of the reasons behind the raise in popularity Python has had over the years.

Some of the other attributes that make Python preferable over other programing languages are: its compatibility with various platforms; the wide application range it has; the simplicity of its language, which allows any user to start coding almost immediately; and last but not least, the large number of freely accessible libraries

and frameworks that boost Python productivity, as there is lot of coding that can be implemented automatically[5].

Regarding the scientific applications, this programming language is useful for an extensive domain of computational chemistry problems involving data analysis and visualization. Van Staveren's article[6] discusses specific scientific applications for this programming language.

It also plays an important role in machine- and deep-learning as, by containing a vast list of chemistry-related libraries, it eases the development of algorithms to build models based on sample data[7]. Overall, it is a great tool to automatize large step calculation processes.

3.1.1 Python functions:

One of the central entities of Python is the function, a piece of the code being defined by related statements that carries out a specific task whenever it is called[8]. They are crucial in programing since they help to produce a modular and organized code by avoiding the repetition of a recurrent task throughout the program.

Functions are elaborated with the *def* keyword that stands for "define". The first line consists of the given name and the arguments (parameters) that the function takes to work with. Afterwards, there is the function body, where the task to be carried out is determined. It is recommended to add a description at the beginning of the body, which can be consulted with the statement *function_name.__doc__*. Finally, after the *return* keyword, there is the output value that the function is asked to deliver.

Figure 12 gives an example of a well-defined function. It calculates the kinetic energy depending on a given mass and velocity, which are the arguments. In the case they are not specified, the default value for each of them is 1.

```
def kinetic_energy(mass = 1.0,velocity = 1.0):
    '''
    This function calculates the kinetic energy
    of a given mass or velocity

    '''
    result = 0.5 * mass * velocity **2

    return result
```

Figure 1. *Kinetic Energy function.*

3.1.2 Good programming habits:

When programming, is important to implement some habits to improve the quality of the code and make it clearer for the other people using it.

A well-structured code can be achieved by means of using proper names for modules, classes, functions and variables; making compact functions ensuring that

they only carry out one task; having an adequate comment for each individual piece of the code to help the user understand every step; and introducing documentation for all the different functions so it is accessible in case some clarification is needed[9].

These habits have been considered when programming the code introduced in this report.

3.2 Module applications:

The module for molecule manipulation presented in this work aspires to be useful for computational chemists dealing with molecular modeling, manipulation, and optimization.

It offers the possibility to modify the orientation of a certain molecule by applying a rotation to match it with a target one. Furthermore, it creates a new output file with the transformed coordinates.

It is not only applicable to pairs of identical molecules, but it also works for non-identical pairs, for example, differently substituted molecules, isomers, or distorted compounds.

The following sections describe some cases where this code could ease the groundwork for any computational researcher.

3.2.1 Electronic interactions in crystalline structures:

Crystalline structures are constructed by the ordered arrangement of a set of atoms, ions or molecules packed in a repetitive way and following some regular pattern. Plenty of chemical compounds present these constitutions.

Over the recent years, crystal structure prediction for different kind of molecules has been a target goal of many studies. The discovery of new materials presenting specific physical properties or pharmaceutical drugs design dealing with organic compounds are some of the examples that justify why is important to study and get to know the chemistry behind the arrangement of these solid layers[10].

Imagine wanting to study, for a given crystalline structure, the electronic effects produced by a radical found in a substituted repeating unit with respect to the neighboring ones. If there is interest on studying these interactions over the whole crystalline lattice, it requires a considerable amount of time to generate all the input files for the possible positions that the studied molecule could adopt. Figure 2 shows an example of a crystal having one substituted unit.

With the presented module, those different files can be generated automatically as, by having the coordinates of a target unit, the studied molecule can be

transformed with the desired orientation, and this long and laborious process is optimized.

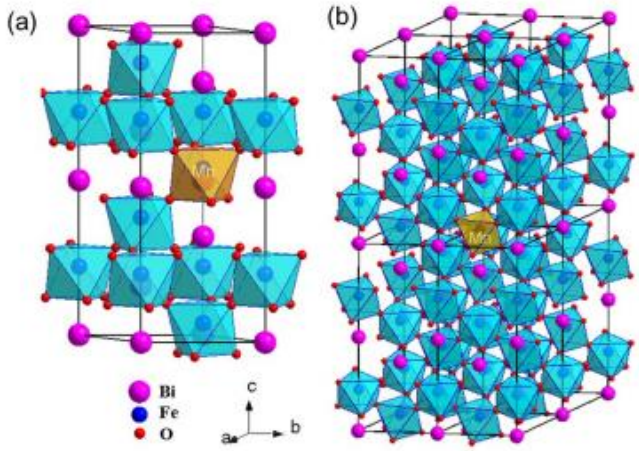

Figure 2. Schematic atomic crystal structures illustration of BFO unit cell with indication of Mn substitution lattice site[11].

3.2.2 Intermolecular energy transfer. Singlet fission:

Beside the rotation of molecules, this module can also be helpful in computational applications that require a considerable amount of wavefunctions describing different electronic state of molecules in different orientations. This requires the rotation of the orbitals that define the electronic state, which is slightly more involved than the relatively simple rotation of the Cartesian coordinates that define the positions of the atoms in the molecule

By being able to apply rotations on wavefunctions, the user would only need to generate the wave functions of the different electronic states in one orientation and then rotate both the molecule and the wave function to produce the other units involved in the process under study.

One particular example that fulfills the explained conditions would be singlet fission, which is a photophysical reaction where a singlet excited state splits into two spin-triplet states that are coupled together as another singlet. This phenomenon has gained relevance lately as it represents a new world for the improvement of the efficiency of photovoltaic solar cells based on organic materials[12]. Figure 3 shows a schematic representation of the singlet fission process.

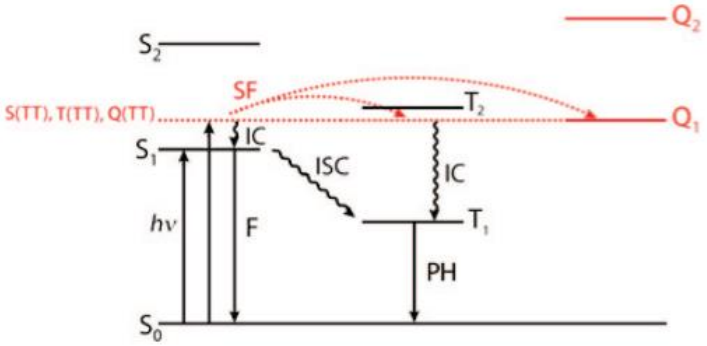

Figure 3. *Schematic diagram of a singlet fission process (SF). IC refers to internal conversion, ISC to intersystem crossing, F to fluorescence and PH to phosphorescence*[13].

Singlet fission materials (such as covalent dimers, polyenes and carotenoids, or conjugated polymers among others) boost electron movement by generating twice as many charge carriers per absorbed photon than conventional materials, or in other words, they can double the induced photocurrent[12].

Computational chemists and theoreticians working on this scope are trying to unravel the mechanism that controls the efficiency of this process, mostly focusing on the coupling between the excited singlet and the singlet coupled double triplet state, and the diffusion of these two triplets. Insight can be obtained by calculating these properties varying the relative orientations of the singlet fission chromophores, which could eventually lead to the discovery of novel types and variations of these materials (also referred as sensitizers) with improved properties.

The generation of the electronic states of these systems in different orientations is usually more time-consuming than the calculation of the couplings itself. This is the reason why the inclusion of wave function rotation in the molecular toolkit can be useful on this specific subject.

4 Theoretical background:

4.1 Rotation matrices:

Rotation is defined as the movement of a certain body when turning around a determined axis[14]. In linear algebra, this type of motion can be applied to determined coordinates in the Euclidean space by employing rotation matrices.

These mathematical objects describe the rotation of a certain object in the two ($\mathbb{R}^2$) or three-dimensional ($\mathbb{R}^3$) space while a coordinate system of axes is being fixed.

4.1.1 Two-dimensional rotation matrices:

In two-dimensional space, a column vector rotates when applying the multiplication of its initial coordinates by the rotation matrix which depends on a determined angle θ. Equation one shows the transformation for a 2D vector.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (1)$$

4.1.2 Three-dimensional rotation matrices:

In the third dimension, two different types of rotation matrices are distinguished.

Basic rotations refer to transformations in which the working axis is defined by one of the three Cartesian axes (*x*, *y*, or *z*). For each one, there is an angle which describes the orientation of the rotated vector with respect to the fixed coordinate system. Those are the so-called Euler angles that, although there are several conventions for them, are generally established as α for the *x*-axis, β for the *y*-axis and γ for the *z*-axis[15].

Figure 4 illustrates a fixed coordinate system with the Euler angles corresponding to each axis.

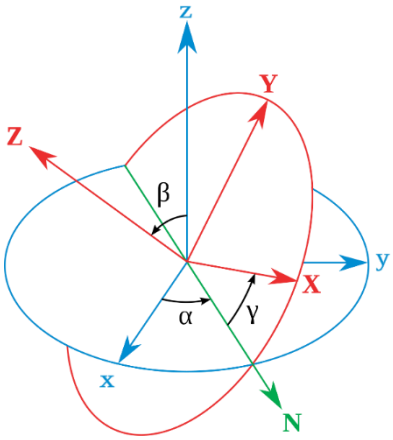

Figure 4. *The transformed vectors (red) are oriented with respect to the fixed coordinate system (blue) by means of an Euler rotation angle (α, β and γ)*[16].

Hence, three different basic rotation matrices ($R_n\ (\varphi)$) can be obtained. Each one of them depending on a Cartesian axis and its corresponding Euler angle, as demonstrated in equations 2-4 [17]. These entities can be applied to transform coordinates in the same way as in the two-dimensional space, by matrix multiplication.

$$R_x\ (\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\ \alpha & sin\ \alpha \\ 0 & -sin\ \alpha & cos\ \alpha \end{bmatrix} \tag{2}$$

$$R_y\ (\beta) = \begin{bmatrix} cos\ \beta & 0 & -sin\ \beta \\ 0 & 1 & 0 \\ sin\ \beta & 0 & cos\ \beta \end{bmatrix} \tag{3}$$

$$R_z\ (\gamma) = \begin{bmatrix} cos\ \gamma & sin\ \gamma & 0 \\ -sin\ \gamma & cos\ \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Furthermore, we have the general forms for three-dimensional rotation matrices. The first one is obtained after the combination of the basic matrices already presented.

It works by splitting the whole rotation motion into three separated sequences that a certain vector experiences over each Cartesian axis. It is only applicable to column vectors in the specified order and knowing the orientation with respect to all the three components of the coordinate system (*α*, *β* and *γ*).

$$R\ = R_x\ (\alpha) \cdot\ R_y\ (\beta) \cdot\ R_z\ (\gamma)$$

$$= \begin{bmatrix} cos\ \alpha\ cos\ \beta & cos\ \alpha\ sin\ \beta\ sin\ \gamma - sin\ \alpha\ cos\ \gamma & cos\ \alpha\ sin\ \beta\ cos\ \gamma + sin\ \alpha\ sin\ \gamma \\ sin\ \alpha\ cos\ \beta & sin\ \alpha\ sin\ \beta\ sin\ \gamma + cos\ \alpha\ cos\ \gamma & sin\ \alpha\ sin\ \beta\ cos\ \gamma - cos\ \alpha\ sin\ \gamma \\ -sin\ \beta & cos\ \beta\ sin\ \gamma & cos\ \beta\ cos\ \gamma \end{bmatrix} \tag{5}$$

The other general description for a three-dimensional rotation is obtained after defining a certain rotation axis and angle. The form of this matrix is shown in equation 6 where n_1, n_2 and n_3 are the *x*, *y* and *z* components of the working axis respectively. The angle θ is the desired orientation of the rotated vector with respect to its initial position.

$$R\ (n, \theta) = \begin{bmatrix} cos\ \theta + n_1^2(1 - cos\ \theta) & n_1 n_2(1 - cos\ \theta) - n_3 sin\ \theta & n_1 n_3(1 - cos\ \theta) + n_2 sin\ \theta \\ n_1 n_2(1 - cos\ \theta) + n_3 sin\ \theta & cos\ \theta + n_2^2(1 - cos\ \theta) & n_2 n_3(1 - cos\ \theta) - n_1 sin\ \theta \\ n_1 n_3(1 - cos\ \theta) - n_2 sin\ \theta & n_2 n_3(1 - cos\ \theta) + n_1 sin\ \theta & cos\ \theta + n_3^2(1 - cos\ \theta) \end{bmatrix} \quad (6)$$

More specific background on this topic can be found in the article *Three-Dimensional rotation matrices*[18].

4.2 Quaternions:

4.2.1 Introduction:

Quaternions are defined as a mathematical expression composed by a scalar part representing a real dimension and a vectorial part comprised by three complex numbers representing an imaginary dimension, see equation 7. They are mainly applied to describe mechanics in 3D such as rotational motions[19].

$$Q = a + bi + cj + dk \quad (7)$$

4.2.2 Early history:

This concept was introduced on October 16th, 1843, by the Irish mathematician William Rowan Hamilton.

Although complex numbers were introduced in the 16th century, they were considered useless at that time by many theoreticians. It was not until Leonhard Euler contributions in the 1700s or Cauchy and Riemann studies in the early 1800s that they were started to be appraised. Hamilton himself was also involved with Complex Analysis before introducing the quaternions as he was striving for the expansion of the 2D vision of complex numbers into the three-dimensional space[20].

After publishing the Couplets Theory which defined two real numbers as a couple, he wanted to extend this concept by expressing a number as the combination of one real and two imaginary parts, the Theory of Triplets[20].

After ten years of dedication, he realized that he needed not two, but three imaginary parts to complete the mathematical structure he was creating.

The legend tells that on the day that gave birth to quaternions he was taking a walk with his wife when he had a mental breakthrough and craved on a stone the formula for the three basic quaternions *i*, *j* and *k*, see equation 8[21].

$$i^2 = j^2 = k^2 = ijk = -1 \quad (8)$$

4.2.3 The quaternion group:

The four-membered tuple that forms a quaternion has some specific algebraic foundation that is important to understand the geographic representation and how they can be applied to describe 3D motion.

Multiplication of basic quaternions (*i*, *j* and *k*; also called basis vectors) is non-commutative, and the product of any two of these components equals plus or minus another one, i.e., $ij = k$ and $ji = -k$[17].

So, under the product of all the possible combinations between basis vectors, a set is formed that represents a three-dimensional system, called the quaternion group, where $\pm i$ is orthogonal to $\pm j$ also orthogonal to $\pm k$. The fourth dimension is introduced when we consider the scalar part represented as *a* in equation 7.

Figure 5 is a 2D representation of the quaternion group. The three axis which define the imaginary dimension are orthogonal to each other. The real dimension is described by the axis going from 1 to -1.

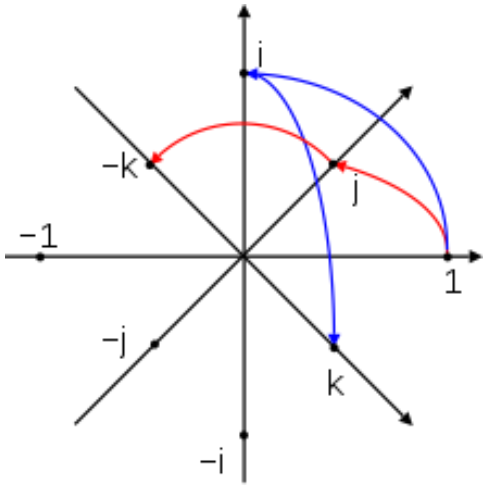

***Figure 5**. Graphical representation of the quaternion group*[22].

4.2.4 Quaternions and 3D transformations:

As said in the introduction of this topic, quaternions are applied in mathematics as transformation entities in the three-dimensional space. When they are used to describe rotations, they are referred as rotation quaternions.

Rotation quaternions describe a given axis an angle by means of an extension of Euler's formula. So, being $n = (n_x, n_y, n_z)$ a rotation axis working around a determined angle θ, the quaternionic representation would be the one shown in the following expression[17]:

$$q = e^{\frac{\theta}{2}(n_x i + n_y j + n_z k)} = cos\frac{\theta}{2} + \left(n_x\, i + n_y\, j + n_z\, k\right) sin\frac{\theta}{2} \qquad (9)$$

An ordinary vector rotates after performing a conjugation operation of itself by a rotation quaternion[20]. Hence, being *p* a vector with the form $p = p_x i + p_y j + p_z k$, gets transformed into *p'* after the operation[23] $p' = qpq^{-1}$.

4.2.5 Advantages of quaternions over rotation matrices and applications:

Quaternions are four-dimensional mathematical entities which are used to represent 3D rotations. As demonstrated, this kind of transformation operations can also be performed with traditional Euler rotation matrices. However, the quaternion properties present advantages with respect to other transforms.

The representation that quaternions offer is numerically more stable and therefore more efficient than those described by rotation matrices. Traditional methods describe rotations as the composition of three rotation sequences about each Cartesian axis. On the other hand, quaternions require only an axis and angle that can be encoded in just four numbers. It is a more compact way to describe the same transformation[24].

Euler Angles matrices also present a significant disadvantage that quaternions avoid, the gimbal lock. This incident which is characteristic of the 3D space consists in the alignment of two rotation axes while the motion is being applied on a third one[25]. For instance, if a rotation of 90 degrees is performed over the *x* axis, the *z* and *y* axes will line up becoming the same and the system will lose one degree of freedom since there would be two axes describing the same motion.

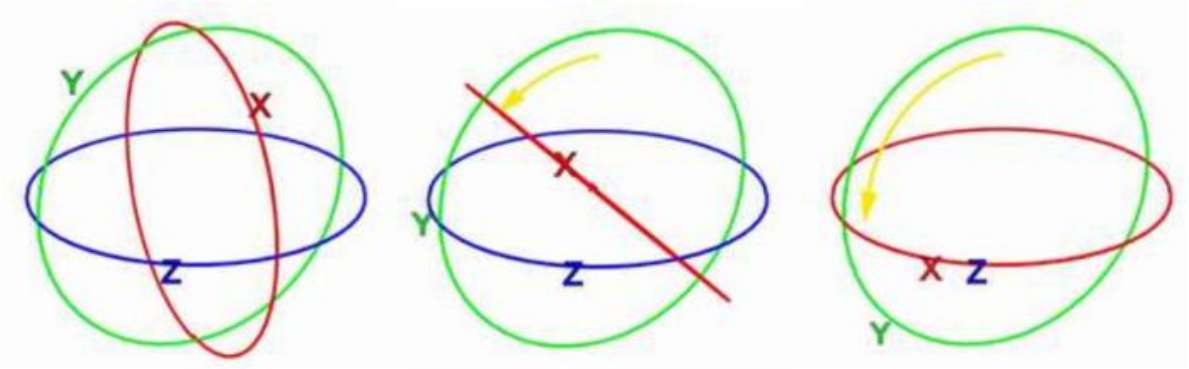

Figure 6. *Gimbal-lock phenomenon. When rotation is applied over the y axis the other two line up*[26].

The mathematical demonstration on how quaternions avoid this phenomenon and more specific background on this topic can be found in the article *Quaternions Algebra, Their Application in Rotations and Beyond Quaternions*[25].

Since they offer a smoother solution to describe rotations, quaternions are employed in different applications such as molecular modelling, gyroscopic motion, navigation, flight simulators and computer game development among many others.

5 Experimental part:

5.1 Matching two molecules:

5.1.1 Program functions:

To develop the presented toolkit for molecule manipulation, several functions had to be defined. This section goes through the ones needed to develop the code for molecule matching.

- *Atom class:*
  In Python, a class defines a type of object. The basic ones that the programming language has by default would be *strings* (not numeric inputs), *integers* (non-decimal numbers), *floats* (integers with a fractional part) and *booleans* (true or false). Besides these ones, more classes can be imported from Python libraries or frameworks.

  Describing a new type of object allows the user to determine its attributes and modify them for all variables of that class at the same time when necessary.

  The atom class that has been defined is useful for cartesian coordinates manipulation as it correlates an atom with its label and the corresponding *x*, *y* and *z*.

  If wanted, more chemical attributes to this object type could be added, like for instance the number of electrons, nuclear charge, electronegativity, etc.

- *Read .xyz file:*
  This function goes through an *.xyz* file and takes all the lines containing coordinates (with their respective element label). It has been defined inspired on the general format for this type of files.

```
def read_xyz_file(filename):
    '''
    This function reads an .xyz file and takes the labels and coordinates
    '''
    coordinates = []
    with open(filename,'r') as xyzFile:

        lines = [line for line in xyzFile.readlines() if line.strip()]
        num_of_atoms = int(lines[0])

        for line in range(2, num_of_atoms + 2):
            coordinates.append(lines[line])

    xyzFile.close

    return coordinates
```

Figure 7. *Read xyz files.*

- *Get molecule*:
  This function uses the atom class to correlate labels with coordinate components. It first calls two other functions (*get xyz* and *get label*) that

split the lines taken from the input file into strings (labels) and integers (coordinates).

Figure 8 shows that the variable *moleculeatom* is in fact an atom class object, and that the output *molecule* is composed by all the objects of this kind that can be created from the argument *lines* indicated with the *for*-loop structure.

```
def get_molecule(lines):
    '''
    This function uses the atom class to organize the labels and
    coordinates of the xyz file.
    '''
    x, y, z = get_xyz(lines)
    label = get_label(lines)
    molecule = []

    for i in range (len(label)):
        moleculeatom = atom(label[i], float(x[i]), float(y[i]), float(z[i]))
        molecule.append(moleculeatom)

    return molecule
```

Figure 8. *Get molecule function.*

- *Translation:*
  This function translates all the coordinates for a given molecule taken as argument.

  It defines a variable *zero coord*, which is an atom from the molecule, and then its coordinates are subtracted to every other one including itself. So, at the end, the coordinates of the atom being defined as the variable will be: *zero_coord* = (0, 0, 0).

  As shown in Figure 9, the argument *molecule* is being modified by another function called *append coordinates*. This function creates a list of *x*, *y*, and *z* coordinates from atom class objects. Hence, by being applied to a list of several atoms, it returns a list of lists.

  The argument *num* will be an input value from the user that gives the index to define the *zero coord* variable.

```
def translation(molecule, num):
    '''
    This function translates the coordinates of a molecule.
    '''
    xyz = append_coordinates(molecule)
    zx, zy, zz = zero_coord = xyz[num]
    trans_xyz = []

    for atom in xyz:
        ax, ay, az = atom
        xyz_prime = [ax - zx, ay - zy, az - zz]
        trans_xyz.append(xyz_prime)

    return trans_xyz
```

Figure 9. *Translation function.*

- *Closest atom:*
  This function finds, within a molecule, the closest atom to the one specified as an argument.

  It does so by creating a list of all the individual distances from the argument atom to all other atoms and returning the list index corresponding to the minimum value.

  In Figure 10 it can be seen that those distances are calculated by calling out the function *get distance*. As the subtraction of the argument atom by itself is 0 (and that would be the minimum value), the function modifies the null distance and appends it as 1000 to the empty list *distances* being filled as the *for*-loop goes.

```
def closest_atom(coords, atom):
    '''
    This function founds, within a list of atom coordinates, the closest atom
    with respect to one being called as argument.
    '''
    distances = []

    for i in range(len(coords)):
        distance_i = get_distance(coords[i], atom)
        if distance_i == 0:
            distance_i = int(1000)
        distances.append(distance_i)

    minimum_distance = min(distances)
    closest_atom = distances.index(minimum_distance)

    return closest_atom
```

Figure 10. *Closest atom function.*

- *Get vector:*
  This function takes two atoms from a list of lists of coordinates and generates a vector connecting them.

```
def get_vector(coords, num, num2):
    '''
    This function takes two atoms from a list of lists of coordinates
    and generates a vector.
    '''
    num_of_atoms = len(coords) - 1

    p0 = coords[num]
    p1 = coords[num2]

    x0, y0, z0 = p0
    x1, y1, z1 = p1

    vector = [x1-x0, y1-y0, z1-z0]

    return vector
```

Figure 11*. Get vector function.*

- *Get cross product*:
  This function finds the normal with respect two vectors by applying the definition of the vectorial product, see equation 12.

$$\vec{c} = \vec{a}\ x\ \vec{b} = \begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} = i\begin{bmatrix} a_y & a_z \\ b_y & b_z \end{bmatrix} - j\begin{bmatrix} a_x & a_z \\ b_x & b_z \end{bmatrix} + k\begin{bmatrix} a_x & a_y \\ b_x & b_y \end{bmatrix} \quad (12)$$

```
def get_cross_product(vector_1, vector_2):
    '''
    This function gives the cross product between two vectors.
    '''
    ux, uy, uz = vector_1
    vx, vy, vz = vector_2
    cross_product = [uy*vz-uz*vy, uz*vx-ux*vz, ux*vy-uy*vx]

    return cross_product
```

Figure 12. *Get cross product function.*

- *Get angle*:
  This function finds the angle with respect two vectors after applying the definition of the vectorial product, see equations 13 and 14.

$$\vec{a} \cdot \vec{b} = |a||b| \cos\theta \quad (13)$$

$$\cos\theta = \frac{a \cdot b}{|a||b|} \quad (14)$$

```
def get_angle(vector_1, vector_2):
    '''
    This function gives the angle resulting from
    the scalar product between two vectors.
    '''
    ux, uy, uz = vector_1
    vx, vy, vz = vector_2

    dot_product = ux*vx + uy*vy + uz*vz
    modulus_1 = math.sqrt(ux**2 + uy**2 + uz**2)
    modulus_2 = math.sqrt(vx**2 + vy**2 + vz**2)
    angle = np.arccos(dot_product/(modulus_1 * modulus_2))

    return angle
```

Figure 13. *Get angle function.*

- *Rotate by Quaternion:*
  This function applies a rotation to the coordinates of an atom. It works out by calling out the "Quaternion" class taken from the library *pyquaternion* which, from a defined axis and angle, applies the Euler extension formula (equation 9) to find a rotation quaternion.

Subsequently, the atom defined in the argument gets transformed after applying a conjugation of itself by the rotation quaternion. As shown in Figure 14, this operation comes implicit with the class.

```
def rotate_by_quaternion(atom, r_axis, r_angle):
    '''
    This function rotates an atom by defining an axis and angle
    and applying the Quaternion class.
    '''
    q = Quaternion(axis = r_axis, angle = r_angle)
    q_coords = q.rotate(atom)

    return q_coords
```

Figure 14. *Quaternion function.*

- *Quadratic regression:*
  This function calculates a quadratic regression of 2nd degree by taking a set of rotation angles and distances.

  It takes as argument a list of three angles and calls the *get distance for angle* function three times (explained later on in this section).

  As shown in Figure 16, the method to calculate the regression is in fact a function taken from *numpy* library. The function gives back the equation of the parabola with the two sets that define it.

```
def get_quadratic_regression(a_1, a_2, r_axis, r_angles):
    '''
    This function:
    1. Generates a set of rotation_angles and distances between two atoms.
    2. Calculates quadratic regression and gives the equation of the parabola.
    '''
    d1 = get_distance_from_angle(a_1, a_2, r_axis, r_angles[0])
    d2 = get_distance_from_angle(a_1, a_2, r_axis, r_angles[1])
    d3 = get_distance_from_angle(a_1, a_2, r_axis, r_angles[2])

    angles_i = [r_angles[0], r_angles[1], r_angles[2]]
    distances_for_angle = [d1, d2, d3]
    equation = np.poly1d(np.polyfit(angles_i, distances_for_angle, 2))


    return equation, distances_for_angle, angles_i
```

Figure 15. *Get quadratic regression function.*

- *Get minimum:*
  This function finds the minimum value of a parabolic equation. Knowing that the general expression for a parabola is:

$$y(x) = ax^2 + bx + c \qquad (15)$$

And that the minimum of a function is the null value of its first derivative (when $y'(x) = 0$). Then, the general formula applied to a parabola would be the one shown in equation 17.

$$y'(x) = \frac{dy}{dx} = 2ax + b$$

(16)

$$if\ y' = 0: \quad x = \frac{-b}{2a} \tag{17}$$

```
def get_minimum(quadraticequation):
    '''
    This function calculates the minimum of a quadratic function.
    '''
    minimum = (-quadraticequation[1])/(2*quadraticequation[2])

    return minimum
```

Figure 16. *Get minimum function.*

- *Distance from angle*:
  This function takes two atoms as argument and calculates the distance after applying a rotation on one of them. So, it relates a certain rotation angle with a distance. As shown in Figure 20, it calls the *quaternion* function to apply the rotation.

```
def get_distance_from_angle(a_1, a_2, r_axis, r_angle):
    '''
    This function calculates the distance between two
    atoms after applying a rotation on one of them.
    '''
    q = quaternion(a_1, r_axis, math.radians(r_angle))
    distance = get_distance(q, a_2)

    return distance
```

Figure 17*. Get distance from angle function.*

5.1.2 Two-plane procedure:

Within all the different capabilities that were thought for this toolkit (stated in the *Objectives* chapter) this work started with the development of a module to match pairs of molecules. As such transformation cannot be performed directly, this code was designed following a specific procedure.

For further understanding, a schematic representation of the result of the code being applied to a pair of cyclohexanes has been added to every step.

1) *Organize the coordinates*:
   First, the code calls the *read xyz file* and *get molecule* functions to take the coordinates and labels of two different input files. At this point, the module has the spatial information of both molecules.

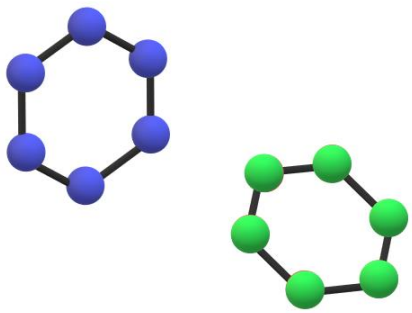

Figure 18. *A pair of identical cyclohexanes with different orientations.*

2) *Translation:*

Once all the coordinates from the chosen *.xyz* files have been taken and organized, the code applies a translation on both molecules by calling out the *translation* function, in such a way that they get connected by a pair of atoms being the *zero coord* variable for each one. The justification of this outcome is shown in equation 11 where the points p and s would represent the coordinates of the superposed atoms.

$$p\,(x_1,\, y_1,\, z_1) \;=\; s\,(x_2,\, y_2,\, z_2) \;=\; (0,0,0) \qquad (18)$$

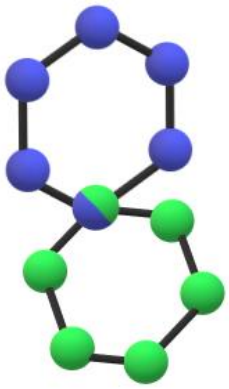

Figure 19. *Two cyclohexanes sharing the same coordinates for one atom.*

3) *Finding a rotation axis:*

The next step consists of defining two equivalent vectors for each molecule by calling the *get vector* function., from the superposed atom to the closest one in each molecule that is found with the function *closest atom*.

Then, the code will find the normal vector between them by calling out the *get cross product function.*

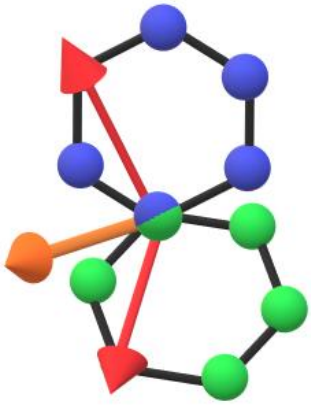

Figure 20. *Normal between two vectors.*

4) *First match:*

A rotation is applied to one cyclohexane. First, the code calls the *get angle function* to find out the angle between the previous vectors. Subsequently, it rotates all the coordinates for one of both molecules by taking the calculated angle as rotation angle, the normal vector as rotation axis and calling out

rotation by quaternion function within a *for*-loop structure that will go through every atom of the molecule. This loop structure can be observed in Figure 21.

As result, both cyclohexanes will be connected by two pairs of superposed atoms.

```
# 5. Apply quaternion rotation with the defined axis and angle:

rot_coords = []

for i in range(len(mol_trans_1)):
    rot_molecule = quaternion(mol_trans_1[i], rotation_axis, rotation_angle)
    rot_coords.append(rot_molecule)
```

Figure 21. *Quaternion function being applied within a loop structure.*

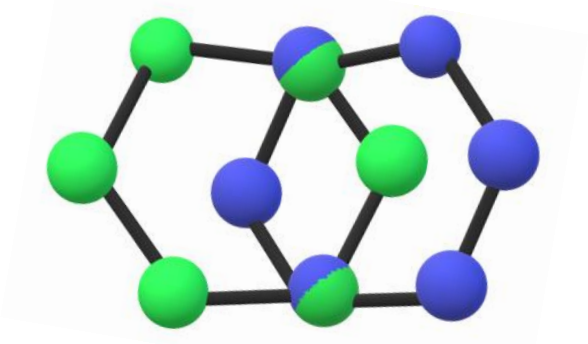

Figure 22. *Cyclohexanes connected by two pair of superposed atoms.*

5) *Find a new rotation angle:*

In order to match the remaining coordinates, a plane for each molecule must be specified. Both planes, spanned by three vectors as Figure 23 shows, will share one of them connecting the pairs of superposed atoms, and the other two, analogous to each other, are defined by connecting the first atoms with a randomly chosen atom.

The angle between both surfaces is also the angle between the normal vectors of each plane, so, by calculating them and employing right after the scalar product the rotation angle is found.

The functions used for this step are once again the *get vector*, *get vectorial product*, and *get angle.*

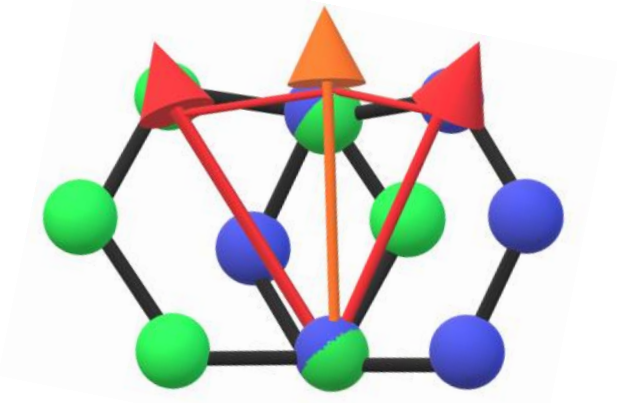

Figure 23. *Two analogous planes defined by three vectors.*

6) *Final rotation:*

After having found the rotation angle and by taking the shared vector for both planes as the rotation axis, a second transformation is applied by calling the *rotate by quaternion* function to achieve a perfect match between both molecules.

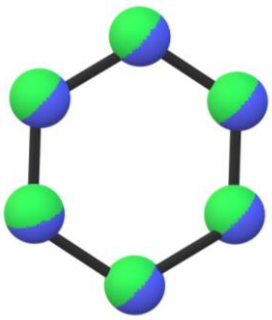

Figure 24. *Two superposed cyclohexanes.*

5.1.3 While-loop procedure:

The two-plane method can be applied for both identical and non-identical pairs of molecules. However, for the second case, this procedure would certainly give good matches but not necessarily the best ones since the two analogous planes that are required, explained in step 5, would not always be mirror images of each other.

Therefore, a more reliable method was designed to be applied by default in the molecule matching module for identical and non-identical pairs.

What distinguishes this new procedure from the other one, is the way they find the angle to perform the final rotation. In this approach, a *while*-loop algorithm is applied. This type of structures work by repeating a specific piece of the code until a condition is met[27].

The loop starts by applying three different rotations to a given atom and calculating the distances between itself and the same atom in the other molecule. By doing this, a set of three rotation angles with three associated distances is created.

Then, the code calls out the *get quadratic regression* function and describes a parabola with the two formed sets. Right after the *get minimum* function is applied to find the angle that would give the lowest distance value from that equation (x when $y' = 0$, see equations 15-17).

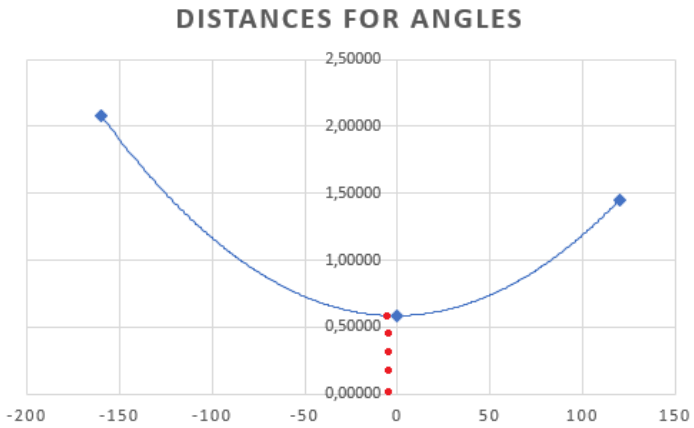

Figure 25. *Example of a quadratic regression and its minimum.*

Finally, the code modifies the set by taking off the largest distance and its associated angle and appending the new values obtained in the previous step. The loop starts again by finding a new parabolic expression with a new minimum angle value and repeats itself *n* times until the minimum distance from the list is lower than a certain threshold value which is specified as a variable. Figure 25 portrays this procedure into a flowchart.

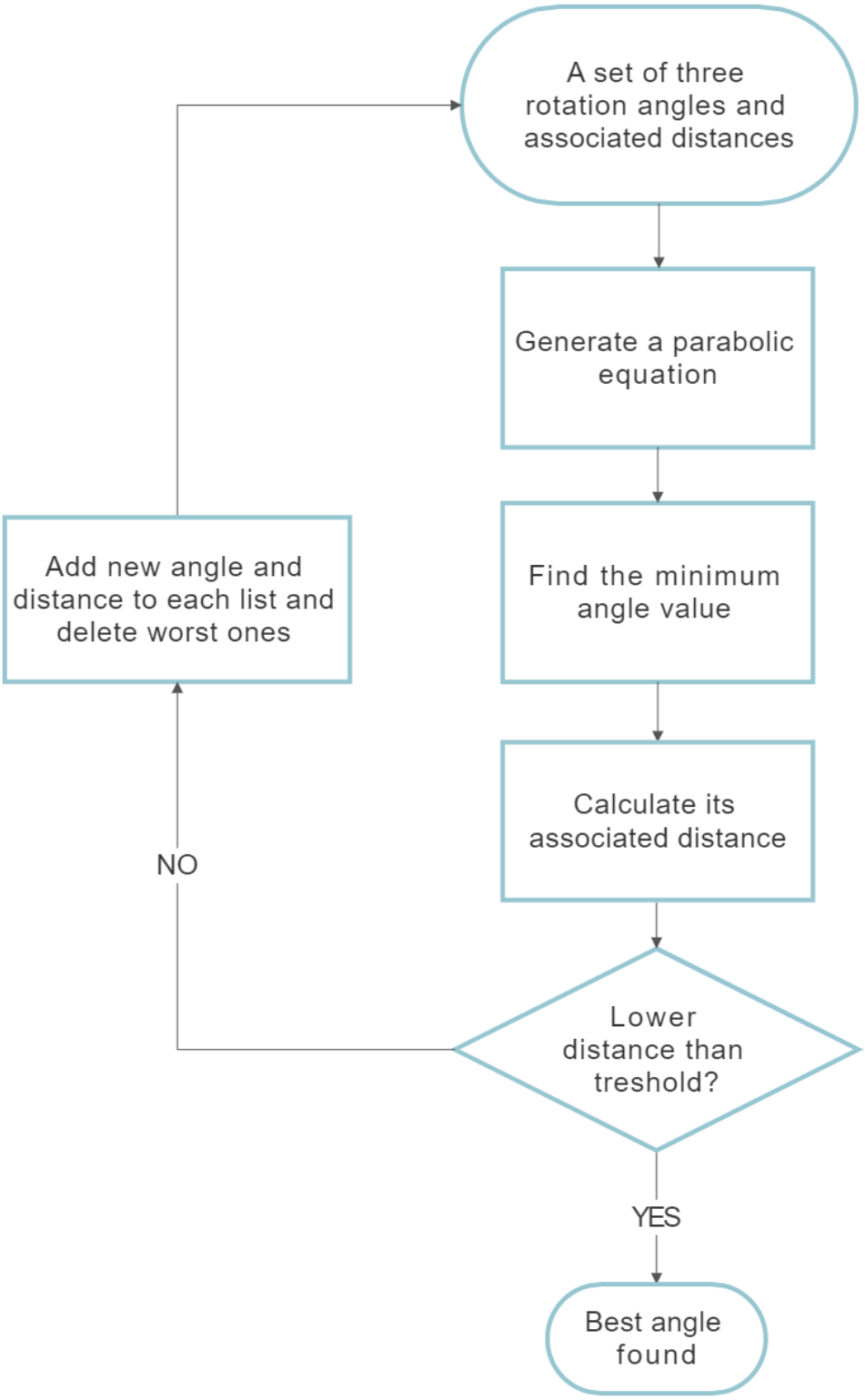

Figure 26. *Flowchart representing the procedure of the while-loop structure.*

5.2 Atomic orbitals:

Moving onto new attributes for this molecular toolkit, a module to apply rotations on atomic orbitals was started. The objective was to demonstrate that for given orbitals defined within a set of three axis *x*, *y* and *z*, their orientation could be transformed applying rotation matrices.

Since the shape for *s*-orbital functions is spherical, rotation will not affect it whatsoever.

5.2.1 Rotation of p-orbitals:

There are three different *p*-orbitals (p_x, p_y and p_z) which share the same lobular shape as it can be observed in Figure 27.

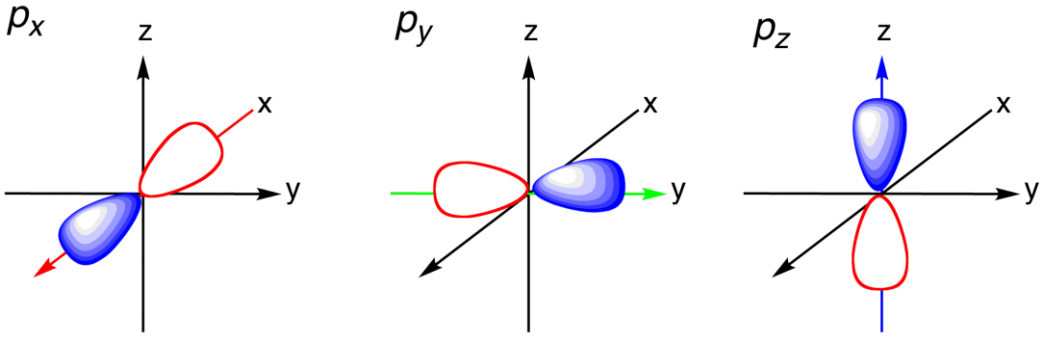

***Figure 27**. Representation of p-orbital functions*[28].

As they all lie along one particular Cartesian axis, they can be described as the unit vector of the one that defines them:

$$p_x = [1, 0, 0] \quad (19)$$

$$p_y = [0, 1, 0] \quad (20)$$

$$p_x = [0, 0, 1] \quad (21)$$

Hence, rotating *p*-orbital functions it is just as easy as defining a certain axis and angle, constructing the three-dimensional rotation matrix following equation 6, and multiplying the given matrix by the vector that describes the orbital.

To apply this procedure into Python, a function was first defined to generate these 3x3 matrices. As shown in Figure 28, it calls out the *normalize* function which gives back the unit vector of the one taken as argument.

The expression to normalize a vector $\vec{v}$:

$$\overrightarrow{u} = \frac{\langle v_x,\ v_y,\ v_z \rangle}{\sqrt{v_x^2 + v_y^2 + v_z^2}} \quad (22)$$

```
def get_3D_rot_matrix(r_axis, r_angle):
    '''
    This function creates a 3x3 matrix from an axis and angle.
    '''
    axis = normalize(r_axis)
    x = axis[0]
    y = axis[1]
    z = axis[2]
    alpha = math.radians(r_angle)
    cos = math.cos(alpha)
    sin = math.sin(alpha)

    R = np.array([[(cos + x**2*(1-cos)), (x*y*(1-cos)-z*sin), (x*z*(1 - cos) + y*sin)],
                  [(y*x*(1-cos)+z*sin), (cos + y**2*(1-cos)), (y*z*(1 - cos) - x*sin)],
                  [(x*z*(1-cos)-y*sin), (y*z*(1-cos)+x*sin), (cos+z**2*(1-cos))]])

    return R
```

Figure 28. *Get a three-dimensional rotation matrix function.*

To show the functionality of the code, two examples are presented:

- *Rotation of 180º around the z axis:*
  If a rotation of 180º around the *z* axis is applied, the p_z orbital function would remain unchanged while the p_x and p_y functions would be converted into - p_x and - p_y

  As shown in Figure 27, the two lobes composing the shape of the orbital have different signs following the positive and negative phases of a wavefunction, so, by applying a rotation of 180º the sense of the lobes gets switched.

  Figure 29 shows the result on applying this rotation on each p-function.

```
# Get a 3D matrix:
matrix_3D = get_3D_rot_matrix([0,0,1], 180)
print('The rotation matrix:', '\n', matrix_3D)

# Apply rotations on p functions:
px = np.array([1,0,0])
py = np.array([0,1,0])
pz = np.array([0,0,1])

print('\n','px rotation: ', '\n', px*matrix_3D )
print('\n', 'py rotation: ', '\n', py*matrix_3D )
print('\n', 'pz rotation: ', '\n', pz*matrix_3D )
```

```
The rotation matrix:
 [[-1  0  0]
 [ 0 -1  0]
 [ 0  0  1]]

 px rotation:
 [[-1  0  0]
 [ 0  0  0]
 [ 0  0  0]]

 py rotation:
 [[ 0  0  0]
 [ 0 -1  0]
 [ 0  0  0]]

 pz rotation:
 [[0 0 0]
 [0 0 0]
 [0 0 1]]
```

Figure 29. *Result on applying the Python module on p-orbital functions.*

- *Rotation of 90º around the* [1,1,1] *axis:*
  If a rotation of 90º is applied around the [1, 1, 1] axis each orbital should get converted into a completely different one, as it can be understood as a 90º rotation of the whole Cartesian system of axes. The result of applying such rotation would be the following:
  - $p_x \rightarrow p_y$
  - $p_y \rightarrow p_z$
  - $p_z \rightarrow p_x$

  Figure 30 shows the result of applying this rotation on the Python code. As shown, each orbital has indeed turned into a completely new one.

```
# Get a 3D matrix:
matrix_3D = get_3D_rot_matrix([1,1,1], 90)
print('The rotation matrix:', '\n', matrix_3D)

# Apply rotations on p functions:
px = np.array([1,0,0])
py = np.array([0,1,0])
pz = np.array([0,0,1])

print('\n','px rotation: ', '\n', px*matrix_3D )
print('\n', 'py rotation: ', '\n', py*matrix_3D )
print('\n', 'pz rotation: ', '\n', pz*matrix_3D )
```

```
The rotation matrix:
 [[0 0 1]
 [1 0 0]
 [0 1 0]]

 px rotation:
 [[0 0 0]
 [1 0 0]
 [0 0 0]]

 py rotation:
 [[0 0 0]
 [0 0 0]
 [0 1 0]]

 pz rotation:
 [[0 0 1]
 [0 0 0]
 [0 0 0]]
```

Figure 30. *Result on applying the Python code on p-orbital functions.*

5.2.2 Rotation of d-orbitals:

There are five different d-orbitals as illustrated in Figure 31. Four of them share the same shape, composed by four lobes of alternating signs with different spatial orientations (d_{xy}, d_{xz}, d_{yz} and $d_{x^2-y^2}$), and the fifth one being a double lobe with a ring having the opposite sign (d_{z^2}).

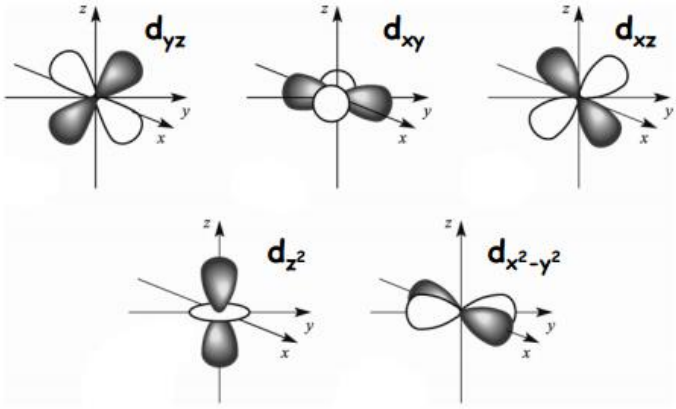

Figure 31. *Representation of the five d-orbitals*[29].

Finding a rotation matrix applicable to *d*-orbitals is, however, more complicated than in the case of *p*-functions. Ivanic et. al.[30] describe a detailed recurrence procedure to obtain rotation matrices between real spherical harmonics from the original 3x3 matrix. So, from a given three-dimensional matrix, a five-dimensional one can be obtained that could derive into a 7D matrix for *f*-functions. Figure 32 shows the result of the recursive process to obtain a five-dimensional rotation matrix.

$(3R_{00}^2 - 1)/2$	$\sqrt{3}R_{01}R_{00}$	$\sqrt{3}R_{0,-1}R_{00}$	$\sqrt{3}(R_{01}^2 - R_{0,-1}^2)/2$	$\sqrt{3}R_{01}R_{0,-1}$
$\sqrt{3}R_{10}R_{00}$	$R_{11}R_{00} + R_{10}R_{01}$	$R_{1,-1}R_{00} + R_{10}R_{0,-1}$	$R_{11}R_{01} - R_{1,-1}R_{0,-1}$	$R_{11}R_{0,-1} + R_{1,-1}R_{01}$
$\sqrt{3}R_{-10}R_{00}$	$R_{-11}R_{00} + R_{-10}R_{01}$	$R_{-1,-1}R_{00} + R_{-10}R_{0,-1}$	$R_{-11}R_{01} - R_{-1,-1}R_{0,-1}$	$R_{-11}R_{0,-1} + R_{-1,-1}R_{01}$
$\sqrt{3}(R_{10}^2 - R_{-10}^2)/2$	$R_{11}R_{10} - R_{-11}R_{-10}$	$R_{1,-1}R_{10} - R_{-1,-1}R_{-10}$	$(R_{11}^2 - R_{1,-1}^2 - R_{-11}^2 + R_{-1,-1}^2)/2$	$R_{11}R_{1,-1} - R_{-11}R_{-1,-1}$
$\sqrt{3}R_{10}R_{-10}$	$R_{11}R_{-10} + R_{10}R_{-11}$	$R_{1,-1}R_{-10} + R_{10}R_{-1,-1}$	$R_{11}R_{-11} - R_{1,-1}R_{-1,-1}$	$R_{11}R_{-1,-1} + R_{1,-1}R_{-11}$

Figure 32. *Five-dimensional rotation matrix obtained after a recurrence procedure. 0 refers to z,; 1 refers to x; and -1 refers to y. Hence, the term R_{00} refers to the R[2][2] element of the 3D rotation matrix, and so on.*[30].

Following Figure 32 expression, a function was created to derive the 5x5 rotation matrix for the rotation of d-functions. To show the functionality of this code, an example is presented.

- *Rotation of 45º around the z axis:*
  Based on Figure 31 representations, the outcome of applying a rotation of 45º around the z axis should be:
  - $d_{z^2} \rightarrow d_{z^2}$
  - $d_{x^2-y^2} \rightarrow d_{xy}$
  - $d_{xy} \rightarrow -(d_{x^2-y^2})$
  - $d_{yz} \rightarrow$ linear combination of d_{yz} and d_{xz}
  - $d_{xz} \rightarrow$ linear combination of d_{yz} and d_{xz}

As shown in Figure 33, these are indeed the results obtained from the Python code.

If the rotation applied does not affect the orbital or transforms it into the opposite (1 or -1), the result would appear in its corresponding place of the 5x5 matrix diagonal. This is the case for the d_{z^2} orbital which occupies the first position. For the ones that give a linear combination, it can be observed that besides the corresponding result on the diagonal, it also appears a contribution in the row corresponding to the orbital involved in that combination. In the case of the $d_{x^2-y^2}$ and d_{xy} functions, the results fall off the diagonal since they have become a completely other orbital.

```
# Get 5D rotation matrix:
matrix_5D = get_5D_rot_matrix(get_3D_rot_matrix([0,0,1], 45))

# Apply rotations on d orbitals:
dz2    = np.array([1,0,0,0,0])
dx2_y2 = np.array([0,0,0,1,0])
dxy    = np.array([0,0,0,0,1])
dyz    = np.array([0,0,1,0,0])
dxz    = np.array([0,1,0,0,0])

print('\n','dz2 rotation: ', '\n', dz2*matrix_5D )
print('\n', 'dx2_y2 rotation: ', '\n', dx2_y2*matrix_5D )
print('\n', 'dxy rotation: ', '\n', dxy*matrix_5D )
print('\n', 'dyz rotation: ', '\n', dyz*matrix_5D )
print('\n', 'dxz rotation: ', '\n', dxz*matrix_5D )
```

```
dz2 rotation:
[[ 1.  0.  0.  0.  0.]
 [ 0.  0. -0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -0.]
 [ 0.  0.  0.  0.  0.]]

dx2_y2 rotation:
[[ 0.  0.  0.  0.  0.]
 [ 0.  0. -0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -0.]
 [ 0.  0.  0.  2.  0.]]

dxy rotation:
[[ 0.  0.  0.  0.  0.]
 [ 0.  0. -0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -2.]
 [ 0.  0.  0.  0.  0.]]

dyz rotation:
[[ 0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0. -0.]
 [ 0.  0.  0.  0.  0.]]

dxz rotation:
[[ 0.  0.  0.  0.  0.]
 [ 0.  1. -0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0. -0.]
 [ 0.  0.  0.  0.  0.]]
```

Figure 33. *The result of applying the Python module on d-functions.*

6 Comparison of the two matching procedures:

Once both modules for molecule matching were programmed, they were tried out to check their functionality in different cases.

- *Matching a pair of identical molecules:*
  First, both methods were applied to a pair of identical propenes. The key atoms required (those that the program asks to the user) were the same ones in both approaches, C1 for the translation and C3 to find out the angle of the final rotation. The outcome was the same for both modules, achieving in both cases a perfect match between the two propenes. Figure 34 shows the whole matching process.

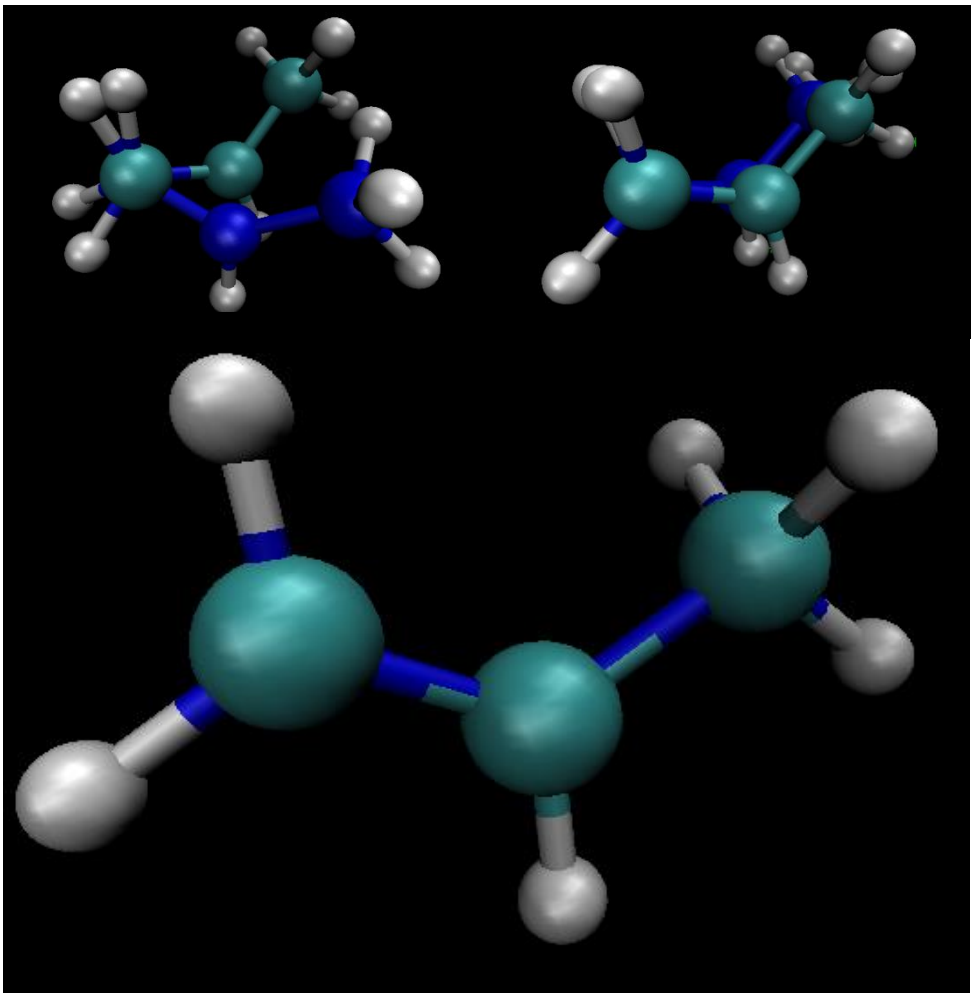

***Figure 34**. Matching two identical propenes. Top Left: after being translated; top right: after applying first rotation; bottom: the result of the perfect match.*

The while-loop procedure works for any first atom defined to be placed in the origin of coordinates. However, some problems were encountered when using the two-plane procedure.

It was seen that, when the selected atoms to be translated were the ones highlighted in Figure 35, the outcome after the final rotation was far off being a perfect match.

Therefore, it can be concluded that the looped procedure will always succeed on matching identical pairs of molecules while the two-plane one requires a good choice for the first atom.

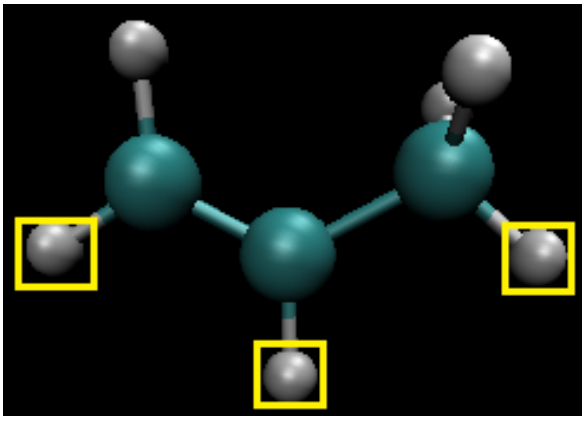

Figure 35. *Propene with highlighted H atoms.*

- *Matching a pair of non-identical molecules*:
  To analyze the behavior of both methods when dealing with pairs of non-identical molecules, two propenes were again selected but this time the structure of one of them was slightly modified. Figure 36 shows the two molecules that were set to be studied.

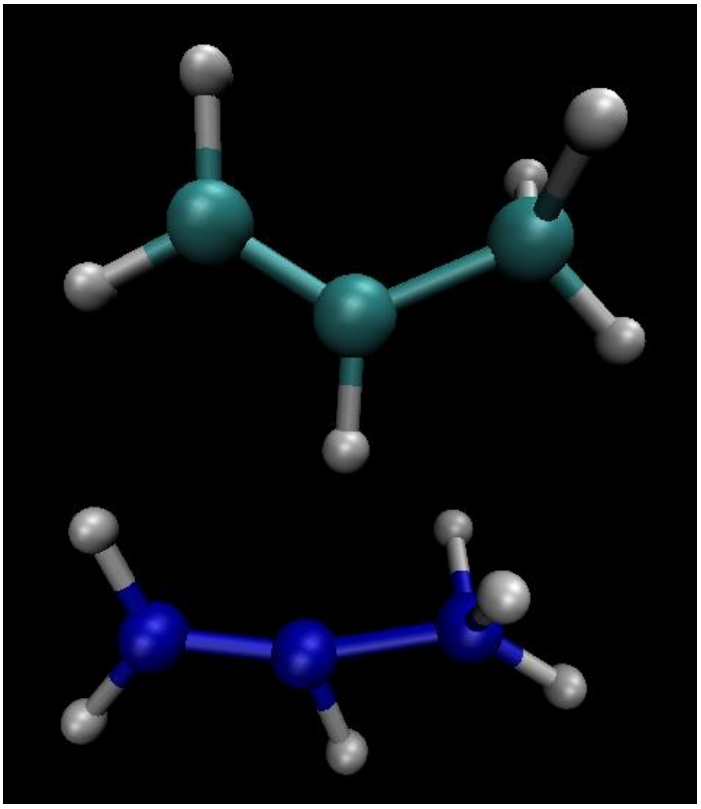

Figure 36. *A pair of non-identical propenes.*

Once more, the same key atoms for both methods were selected being C1 for the translation and C3 for the final rotation.

As already said, the development of the while-loop algorithm was started with the thought that it would give better results than the two-plane one. However, the outcome of this study resulted on being very similar for both methods. Figure 37 shows the result for both approaches. The one in the top corresponds to the quadratic regression search while the bottom one corresponds to the two-plane match.

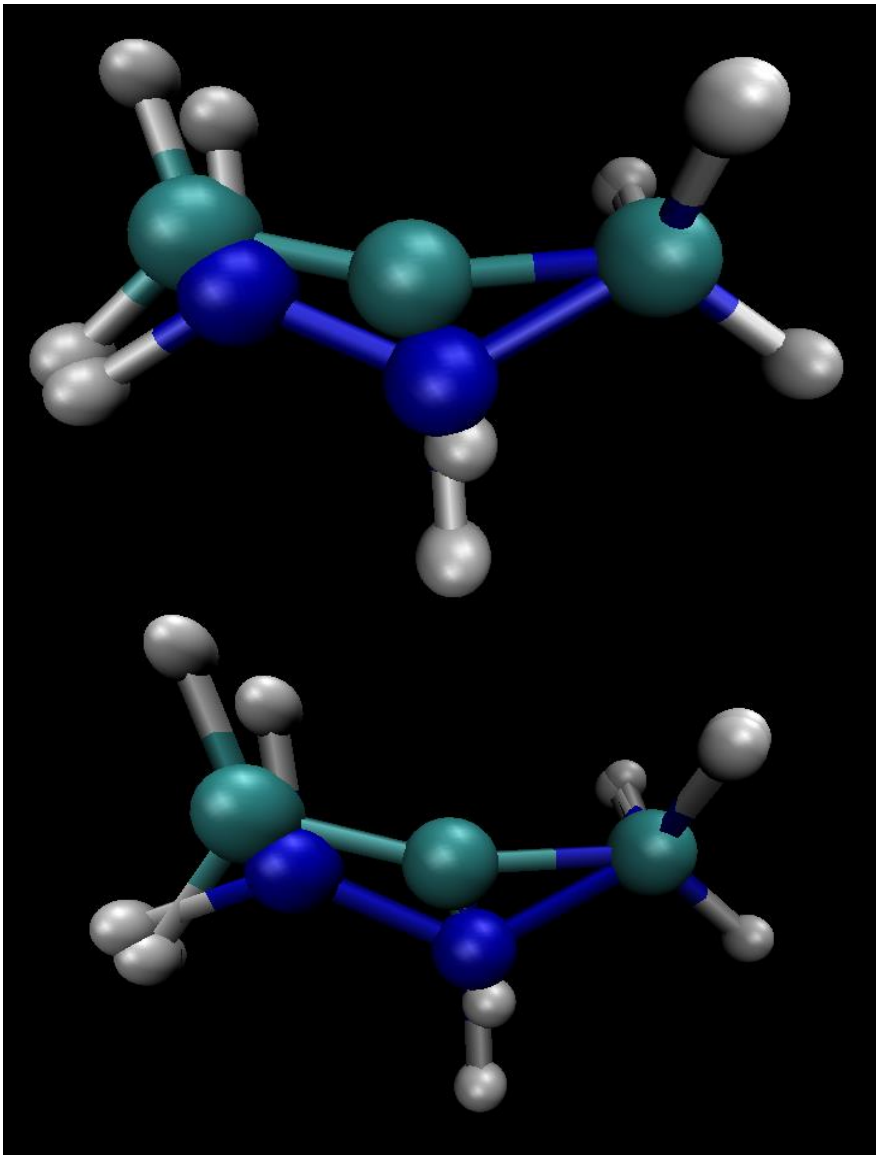

Figure 37. *The outcome on applying both methods for a pair of non-identical molecules. Top: while-loop method; Bottom: two-plane procedure.*

It can be observed that the transformed atoms are slightly closer in the first outcome, however, this was also confirmed by calculating the average distance between both molecules. The results for the calculated distances were 2,772 $\dot{A}$ for the two-plane procedure and 2,739 $\dot{A}$ for the while-loop. So, it can be concluded that a more precise match has been achieved for this last method.

In this case, a good choice of the key atoms would also result in yielding better matches. As shown in Figure 37, if C2 is chosen as the atom to iterate until finding the best angle, the outcome is much more different than the one in Figure 36.

However, both methods have been successfully applied to perform matches between different pairs of molecules.

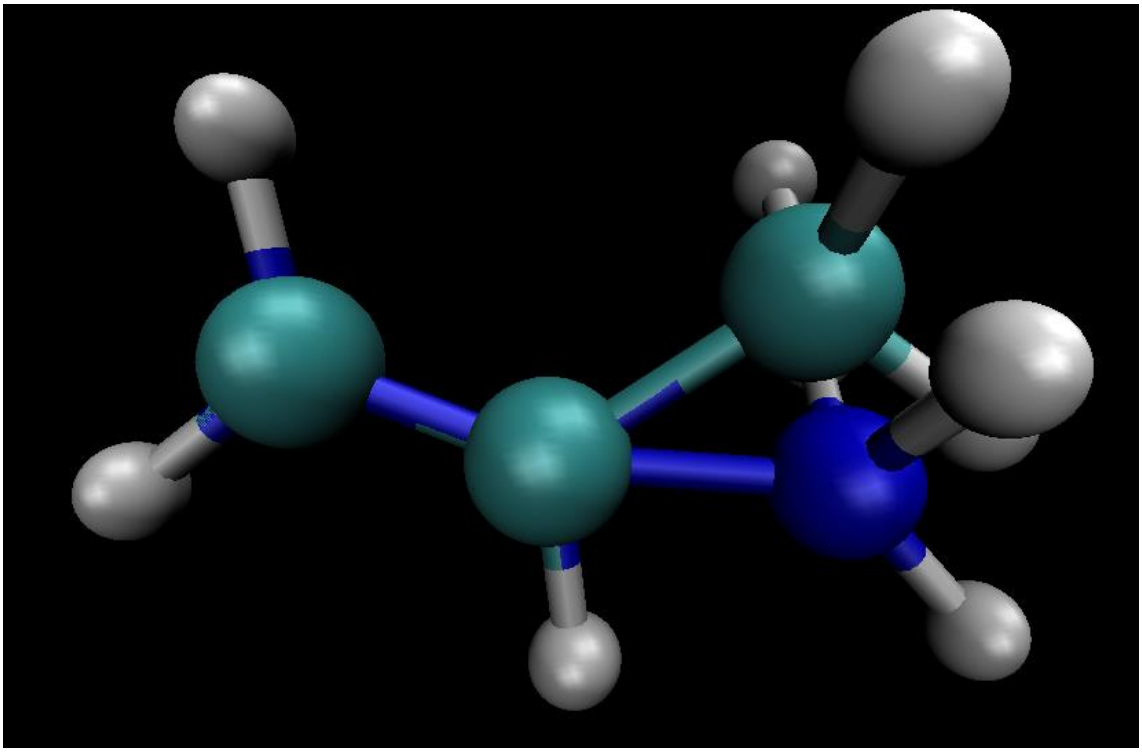

Figure 38. *Alternative match found by defining the key atoms in another way.*

7 Conclusions:

English:

This work has supposed the beginning of the development of a Python toolkit that could be useful in the daily research of any computational chemist.

Starting with the implementation of the code to generate overlaps between molecules, two different methods have been proposed that fulfill the intended function. On one hand, when the module is applied to pairs of identical molecules, both the two-plane and the while-loop approximations work correctly producing exact overlaps. Moreover, when a pair of non-identical molecules is wanted to be studied, although at first it was thought that the loop would generate much better results, they yield quite similar matches.

Despite the fact that the code works correctly in most cases, a good choice of the first atom that determines the translation process will result in more accurate matchings, especially when using the two-plane procedure. That is why this module can be further improved by implementing algorithms that can automatically find the best choice. This is also applicable in the selection of the atom which the while-loop works with to generate distances associated with certain angles to then find the best one by minimizing the quadratic expression.

Regarding the manipulation of p and d orbitals, the created code could be applied in computational studies involving electronic states for a certain molecule in different orientations, by applying rotations to the orbitals describing those states. Following the method used to generate the penta-dimensional matrix, the module could end up being used to describe rotations for the *f*-orbital functions deriving the 5D matrix into an hepta-dimensional one.

To conclude, the outcome of this work presents a good starting point for a more extensive development of a molecule manipulation toolkit with many more attributes. Hence, the main goal that was imposed at the beginning has been achieved.

Catalan:

Aquest treball ha suposat l'inici del desenvolupament d'un mòdul de Python que pot arribar a ser útil per la recerca quotidiana de qualsevol químic computacional.

Començant per la implementació del codi per generar superposicions entre molècules, s'han arribat a plantejar dos mètodes diferents que compleixen amb la funció prevista. Per una banda, quan el mòdul és aplicat a parells de molècules idèntiques, tant l'aproximació dels dos plans com la basada en el while-loop funcionen correctament produint superposicions exactes. Altrament, quan es vol estudiar un parell de molècules no idèntiques, encara que en un inici es va creure que el bucle generaria molt millors resultats, presenten superposicions bastant similars.

Tot i que el codi funciona correctament en la majoria dels casos, una bona tria del primer àtom que determina el procés de translació esdevindrà en aparellaments més precisos, especialment en el mètode dels dos plans. És per això que aquest mòdul pot seguir-se millorant mitjançant la implementació d'algoritmes que

puguin trobar aquesta millor tria automàticament. Això esmentat també és aplicable en la tria del àtom amb el que el bucle treballa per generar distàncies associades a determinats angles per després acabar trobant el millor minimitzant l'expressió quadràtica.

Respecte a la manipulació d'orbitals *p* i *d*, el codi creat podria aplicar-se en estudis computacionals on s'involucrin estats electrònics d'una mateixa molècula en diferents orientacions. Seguint amb el mètode emprat per generar les matrius penta-dimensionals, el mòdul podria acabar utilitzant-se per descriure rotacions de les funcions orbitals *f* generades per una matriu hepta-dimensional.

En definitiva, el resultat d'aquest treball presenta un bon punt de partida per el desenvolupament més extensiu d'un conjunt d'eines per manipular molècules i, per tant, s'ha assolit l'objectiu principal que es va imposar.

8 Bibliography

(1) Quantum Chemistry Group Home Page http://www.quimica.urv.es/w3qf/ (accessed May 21, 2022).

(2) History of Python - GeeksforGeeks https://www.geeksforgeeks.org/history-of-python/ (accessed May 25, 2022).

(3) TIOBE Index - TIOBE http://www.tiobe.com/tiobe-index/ (accessed May 25, 2022).

(4) Python Software Foundation | Python Software Foundation https://www.python.org/psf/ (accessed May 30, 2022).

(5) Reasons to Use Python Over Other Programming Languages - ThePythonGuru.com https://thepythonguru.com/reasons-to-use-python-over-other-programming-languages/ (accessed May 28, 2022).

(6) van Staveren, M. Integrating Python into a Physical Chemistry Lab. *J. Chem. Educ.* **2022**. https://doi.org/10.1021/acs.jchemed.2c00193.

(7) Why Python is Good for Machine Learning | Engineering Education (EngEd) Program | Section https://www.section.io/engineering-education/why-python-is-good-for-machine-learning/ (accessed Jun 13, 2022).

(8) Python Functions (def): Definition with Examples https://www.programiz.com/python-programming/function (accessed Jun 13, 2022).

(9) Praxaena, P. 5 Python Best Practices That Every Programmer Should Follow | by Pranjal Saxena | Towards Data Science. *Towards Data Science*. 2021.

(10) Methods and applications of crystal structure prediction Faraday Discussion | Royal Chemistry Society https://www.rsc.org/events/detail/24508/methods-and-applications-of-crystal-structure-prediction-faraday-discussion (accessed Jun 13, 2022).

(11) Yan, F.; Xing, G.; Wang, R.; Li, L. Tailoring Surface Phase Transition and Magnetic Behaviours in BiFeO3 via Doping Engineering. *Sci. Rep.* **2015**.

(12) Casanova, D. Theoretical Modeling of Singlet Fission. *Chem. Rev.* **2018**, 44.

(13) Tykwinski, R. R.; Guldi, D. M. Singlet Fission. *ChemPhotoChem* **2021**, *5* (5), 392. https://doi.org/10.1002/cptc.202100053.

(14) Rotation Definition & Meaning - Merriam-Webster https://www.merriam-webster.com/dictionary/rotation (accessed Jun 15, 2022).

(15) Euler Angles -- from Wolfram MathWorld https://mathworld.wolfram.com/EulerAngles.html (accessed Jun 13, 2022).

(16) Rotation matrix - Wikipedia https://en.wikipedia.org/wiki/Rotation_matrix (accessed Jun 13, 2022).

(17) Ben-Ari, M. A Tutorial on Euler Angles and Quaternions.

(18) Three-Dimensional Rotation Matrices. In *Physics 216*; 2012.

(19) Quaternion Definition & Meaning | Dictionary.com https://www.dictionary.com/browse/quaternion (accessed Jun 13, 2022).

(20) Buchmann, A.; By Daniele Struppa, A. C. A Brief History of Quaternions and the Theory of Holomorphic Functions of Quaternionic Variables.

(21) Jia, Y.-B. Quaternions and Rotations *. **2013**.

(22) Quaternion - Wikipedia https://en.wikipedia.org/wiki/Quaternion (accessed Jun 13, 2022).

(23) Maths -Quaternion Transforms - Martin Baker https://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/transforms/index.htm (accessed Jun 15, 2022).

(24) Quaternions and spatial rotation - Wikipedia https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation (accessed Jun 7, 2022).

(25) Güna¸stı, G. G. Quaternions Algebra, Their Applications in Rotations and Beyond Quaternions. **2016**.

(26) Beun, R. J. Human-Media Interaction University of Twente Human Media Interaction. No. May 2014.

(27) Programming - While Loop https://www.cs.utah.edu/~germain/PPS/Topics/while_loops.html (accessed Jun 14, 2022).

(28) Shape of p-orbitals in 3D https://www.chemtube3d.com/orbitals-p/ (accessed Jun 14, 2022).

(29) Chemistry: D orbitals http://openchemistryhelp.blogspot.com/2012/08/inorganic-chemistry-shriver-and-atkins_8.html (accessed Jun 15, 2022).

(30) Ivanic, J.; Ruedenberg, K. Rotation Matrices for Real Spherical Harmonics. Direct Determination by Recursion. *J. Phys. Chem.* **1996**, *100* (15), 6342–6347. https://doi.org/10.1021/jp953350u.