PyQchem Documentation

Abel Carreras

Aug 27, 2023

Contents

A python wrapper for Q-Chem software (https://www.q-chem.com).

CHAPTER 1

Introduction

PyQchem in a python interface for Q-Chem, a popular general-purpose quantum chemistry maintained and distributed by Q-Chem, Inc., located in Pleasanton, California, USA.

PyQchem allows to take advantage of Python's simple and powerful syntax to automatize Q-Chem calculations. For this purpose PyQchem implements an input generation class, a calculation submitting function and a set of flexible parsers that extracts the output information and converts in a well structured python dictionary. These parsers are intended to be as homogeneous as possible among the different methods producing a python dictionary that contains similar entries and can be used with the same analysis/visualization functions.

The philosophy of PyQChem is to build a homogeneous input and output interface for the different methods implemented in Q-Chem to make the life of Q-Chem users easier.

1.1 Main features

- Easy to use & clean python interface
- Easy to install in your personal computer or cluster, no special q-chem compilation needed.
- Output parser support for a variety of calculation.
- Modular implementation that allows to easily extend its functionality by writing new parsers and analysis functions.

CHAPTER 2

Installation

PyQchem can be installed directly from the source code (python package) or via PyPI repository. Q-Chem is not necessary to be installed in the system but, of course, it will be necessary later on to perform calculations. Still some of pyqchem features can be used without Q-Chem.

2.1 Requirements

- Python 2.7.x/3.5+
- numpy
- scipy
- matplolib
- requests
- lxml
- wfnsympy (optional: for symmetry analysis)
- paramiko (optional: for remote calculations)

2.2 Install

1) From source code

```
git clone https://github.com/abelcarreras/PyQchem.git pyqchem
cd pyqchem
python setup.py install --user
```

2) From PyPI repository

```
pip install pyqchem --user
```

2.3 Q-Chem setup

PyQchem checks two environment variables to locate Q-Chem installation: $QC and $QCSCRATCH. *$QC* contains the path to the root directory of Q-Chem installation. This directory should contain a */bin* directory where *qchem* run script is placed. *$QCSCRATCH* contains the path to scratch directory.

Note: $QCAUX and $QCREF environment variables are not directly used by pyqchem but they are used by Q-Chem to properly run. Check Q-Chem manual for further information.

CHAPTER 3

Get started

A basic pyqhem script is composed of 4 steps: Defining a molecule, preparing an Q-Chem input, runing the calculation and parsing the information.

3.1 Defining molecule

The definition of the molecule is done creating an instance of Structure class. Its initialization requires the coordinates as a list of lists (or Nx3 numpy array), the atomic symbols of the atoms (in the same order of the coordinates), the charge and the multiplicity. Only coordinates and symbols are mandatory, if charge/multiplicity are not specified they will be defined as neutral/singlet.

Example of hydroxide anion:

```
from pyqchem import Structure

molecule = Structure(coordinates=[[0.0, 0.0, 0.0],
                                  [0.0, 0.0, 0.9]],
                     symbols=['O', 'H'],
                     charge=-1,
                     multiplicity=1)
```

3.2 Preparing Q-Chem input

The Q-Chem input is defined using the QchemInput class. To initialize this class it is necessary an instance of the Structure class as defined in the previous section (molecule). Afterwards the different Q-Chem keywords are set as optional arguments to specify the calculation. In general the name of these keywords have the same name as in Q-Chem. At this moment, only a small set of the keywords available in Q-Chem are implemented in the QchemInput class. Refer at this class definition to check which ones are implemented. New keywords will be implemented under request.

Example of single point calculation using unrestricted Hartree-Fock method with 6-31G basis set:

```
from pyqchem import QchemInput

qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G',
                      unrestricted=True)
```

Note: In case a particular Q-Chem keyword of *$REM* input section in not implemented in PyQchem, *extra_rem_keywords* argument can be used to include it. This argument requires a dictionary containing the keywords as dictionary keys and its values as dictionary values. Values can be either strings or numbers.

```
qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G',
                      extra_rem_keywords={'keyword_1': 'value',
                                          'keyword_2': 34}
```

Also for features that require a non-implemented Q-Chem input section *extra_sections* argument can be used to include it. This argument requires a list of *CustomSection* objects which can be imported from pyqchem.qc_input. These objects are defined by a section title, that corresponds to the title in Q-Chem input section, and keywords that is a dictionary with the name and value of the keywords in the section.

```
from pyqchem.qc_input import CustomSection

custom_section = CustomSection(title='section_title',
                               keywords={'sec_keyword_1': 'text_value',
                                         'sec_keyword_2': 34})


custom_section_2 = CustomSection(title='section_title_2',
                                 keywords={'sec_keyword_1': 'text_value',
                                           'sec_keyword_2': 34})

qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G',
                      extra_sections=[custom_section, custom_section_2]
```

3.3 Running calculations

Once the QchemInput is prepared you can run it using *get_output_from_qchem* function. This function if the core of PyChem and makes the connection between PyQchem and Q-Chem. In order for this function to work properly **$QC** and **$QCSCRATCH** environment variables should be defined. Refer to Q-Chem manual to learn how to set them. By default *get_output_from_qchem* will assume an openMP compilation of Q-Chem were *processors* indicate the number of threads to use in the calculation. If your compilation of Q-Chem is MPI then *use_mpi=True* should be set and *processors* will correspond to the number of processors.

Get_output_from_qchem function has two main ways of operation. If no parser is specified, then the output of this function will be a string containing the full Q-Chem output. This way can be useful to do your own treatment of the output file or if you are not sure about the information you want to parse.

Example of simple parallel(openMP) calculation using 4 threads:

```
from pyqchem import get_output_from_qchem

output = get_output_from_qchem(qc_input,
                               processors=4)
```

The second way is by defining the *parser* optional argument. This indicates that the output will be parsed using the specified parser function. In the following example *basic_parser_qchem* function is used. This is imported from the parser collection located at *pyqchem.parsers.*. Using a parser function the output of this function becomes a python dictionary containing the parsed data.

This is similar to the example shown above using a simple parser (*basic_parser_qchem*) :

```
from pyqchem.parsers.basic import basic_parser_qchem
parsed_data = get_output_from_qchem(qc_input,
                                    processors=4,
                                    parser=basic_parser_qchem,
                                    )
```

This can be done also in two steps, since the parser (*basic_parser_qchem* in this case) is a just regular python function that accepts a string as argument.

```
output = get_output_from_qchem(qc_input, processors=4)
parsed_data = basic_parser_qchem(output)
```

It is simple to create a custom parser by defining a custom function with the following structure:

```
def custom_parser_qchem(output):
    """
    output: contains the full Q-Chem output in a string

    return: a dictionary with the parsed data
    """
    ...
    return {'property_1': prop1,
            'property_2': prop2}
```

Complex parsers may have optional arguments to add more control. This may be used to include parameters such as precision, max number of cycles/states/etc to read, etc..:

```
def custom_parser_qchem(output, custom_option=True, custom_prec=1e-4):
    """
    output: contains the full Q-Chem output in a string
    custom_option: controls option to be used or not
    custom_prec: defines the precision of som data to be read

    return: a dictionary with the parsed data
    """
    ...


    return {'property_1': prop1,
            'property_2': prop2}
```

to define this optional arguments *get_output_from_qchem* function you should include *parser_parameters* argument which requires a python dictionary. Each of the entries in this dictionary should be the name of one of the optional arguments in the parser function whose value is the value of the argument:

```
parsed_data = get_output_from_qchem(qc_input,
                                    processors=4,
                                    parser=custom_parser_qchem,
                                    parser_parameters={'custom_option': True, 'custom_
↪prec': 1e-4}
                                    )
```

Most of the electronic information (molecular orbitals coefficients, electronic density, basis set, etc..) can be found in fchk file generated by Q-Chem. Other information (hessian, Fock matrix, etc..) can be read from binary files generated in the work directory. All this is stored in a dictionaty and returned by *get_output_from_qchem* function if optional argument *return_electronic_structure=True* is used:

```
from pyqchem.parsers.basic import basic_parser_qchem
parsed_data, electronic_structure = get_output_from_qchem(qc_input,
                                                          processors=4,
                                                          parser=basic_parser_qchem,
                                                          return_electronic_
↪structure=True
                                                          )
```

as can be observed in the previous example, the return of *get_output_from_qchem* function contains two elements: *parsed_data* and the *electronic_structure*. *Parsed_data* is a python dictionary that contains the same information as previously described. *Electronic_structure* is another python dictionary that contains the information parsed from the FCHK file.

Note: PyQchem automatically includes the Q-Chem keyword *gui=2* to the input if *return_electronic_structure=True* is requested.

3.4 Reusing data efficiently

Pyqchem is specially focused in the automation and design of complex Q-Chem workflows. For this reason pyqchem implements a feature to avoid redundant calculation by storing the parsed data in a pickle file. This works seamessly, if a calculation is requested with an input *equivalent* to a previous one, the calculation is skip and stored data is output instead. By default only parsed data is stored, therefore if no parser is provided the calculation will be recomputed.

The behavior of this feature is controlled by two arguments in *get_output_from_qchem* function: *force_recalculation* and *store_full_output*. *force_recalculation=True* forces the calculation to be calculated even if a previous *equivalent* calculation already exists. If *store_full_output=True* then the raw outputs are also stored. This may produce a significant increase in size of the storage file, but it can be useful to test new parsers or to use several parsers in the same output.

```
parsed_data = get_output_from_qchem(qc_input,
                                    processors=4,
                                    parser=basic_parser_qchem,
                                    force_recalculation=True,
                                    store_full_output=True
                                    )
```

It is possible to set a custom storage pickle filename by using *redefine_calculation_data_filename* function. This may be written at the beginning of the script to define a different storage file for each script if multiple scripts run in the same directory at the same time.

```
from pyqchem.qchem_core import redefine_calculation_data_filename
redefine_calculation_data_filename('custom_file.pkl')
```

CHAPTER 4

Advanced input

4.1 Custom guess

structure entry contains a Structure type object that can be used, for instance, in *QchemInput* to create a new input. *scf_guess* requires a dictionary {'alpha': [], 'beta':[]} containing the molecular orbitals coefficients matrix. This is exactly what *coefficients* entry contains. Using this two objects it becomes easy to use the electronic structure of a previous calculation as a guess of a new calculation (see **frequencies_simple.py** example) :

```
qc_input = QchemInput(electronic_structure['structure],
                      scf_guess=electronic_structure['coefficients']
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G')
```

4.2 Custom basis set

The same can be done for basis set. *QchemInput* basis argument accepts predefined basis sets included in Q-Chem as a label string (e.g. 'sto-3g', '6-31g(d,p)',..) but also accepts custom basis sets. These basis sets should be written as a python dictionary following the same structure as the one output in *electronic_structure*. These basis can be used directly in *QchemInput*:

```
qc_input = QchemInput(electronic_structure['structure],
                      scf_guess=electronic_structure['coefficients']
                      jobtype='sp',
                      exchange='hf',
                      basis=electronic_structure['basis])
```

However this is may not very useful if the basis in *electronic_structure* is one of the predefined basis in Q-Chem. PyQchem include a helper function to retrieve a basis set from *ccRepo* (http://www.grant-hill.group.shef.ac.uk/ccrepo/) repository. This function require as argument Structure object and the name of the basis set (see: **custom_basis.py** example):

```
from pyqchem.basis import get_basis_from_ccRepo

basis_custom_repo = get_basis_from_ccRepo(molecule, 'cc-pVTZ')
qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis=basis_custom_repo)
```

4.3 Dual basis set

The use of dual basis set can improve the performance of Q-Chem calculations. This can be used, for example, to use as a guess a previous calculations tha uses a smaller basis set. The keyword to use this is *basis2* and works in the same way as *basis*. Usual *basis* keyword defines the new basis and *basis2* keyword defines the previous (and smaller) basis.

```
# Initial calculation using sto-3g basis set
qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='sto-3g',
                      )

_, ee = get_output_from_qchem(qc_input, return_electronic_structure=True)

# Precise calculation with larger 6-31G basis using previous MO as guess
qc_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31g',
                      basis2=ee['basis'],  # previous basis from electronic structure
                      scf_guess=ee['coefficients'] # previous MO coeff as a guess
                      )
```

4.4 Usage of Solvent

Usage of solvent is implemented in pyQchem by the use of *solvent_method* and *solvent_params*. *solvent_method* is a strightforward of the keyword with the same name in Q-Chem while *solvent_params* is a dictionary that contains the keywords in the section **$solvent** in Q-Chem input. For PCM that requiere additional parameters *pcm_params* keyword is used which implements the keywords of **$pcm** section in Q-Chem input.

```
qc_input = create_qchem_input(molecule,
                              jobtype='sp',
                              exchange='hf',
                              basis='sto-3g',
                              unrestricted=True,
                              solvent_method='pcm',
                              solvent_params={'Dielectric': 8.93},  # Cl2CH2
                              pcm_params={'Theory': 'CPCM',
                                          'Method': 'SWIG',
                                          'Solver': 'Inversion',
                                          'Radii': 'Bondi'}
                              )
```

CHAPTER 5

Symmetry

PyQchem is combined with wfnsympy to analyze the symmetry of the wave function calculated using Q-Chem. PyQchem implements several symmetry functions that are compatible with *electronic_structure* dictionary. Due to the limitations of *wfnsympy* at this moment only gaussian type orbitals (GTO) basis can be used.

5.1 Orbital classification

A simple function included in pyQchem is *get_orbital_classification*, this function classify the molecular orbitals between PI and SIGMA. For this function to work properly the user needs to define the center ond orientation of the molecule. Assuming that the molecule is planar *center* should be a point within the plane that the atoms form and *orientation* is a unitary vector perpendicular to this plane.

The return of this function is a list. Each element of this list correspond to a molecular orbital (in energy order from lower to higher) and contains two things: a label that indicates the type of orbitals (SIGMA or PI) and a float number that indicates the degree of accuracy (from 0[None] to 1[Full]).

```
from pyqchem.symmetry import get_orbital_classification

orbital_types = get_orbital_classification(electronic_structure,
                                           center=[0.0, 0.0, 0.0],
                                           orientation=[0.0, 0.0, 1.0])

for i, ot in enumerate(orbital_types):
    print('{:5}:  {:5} {:5.3f}'.format(i + 1, ot[0], ot[1]))
```

Sometimes molecules may not be planar but still some notion of pi/sigma symmetry can be extracted, even if its only local. For this reason *pyqchem* implements several functions to manipulate electronic structures (See **classify_orbitals.py** and **advanced_symmetry.py** as more complete examples)

The following example shows the use of two of these functions (*get_plane*, and *crop_electronic_structure*) to determine the orbital symmetry (PI/SIGMA) of a fragment of a large molecule. *Get_plane* allows to determine the plane and orientation of the fragment. This function assumes that all atoms are in the same plane, so if some atoms are out of the plane it may be more adequate to use this function with the subset of the atoms that are more or less in a plane.

On the other hand, *crop_electronic_structure* modifies the MO coefficients, by setting all the basis functions that are not centered in the atoms of the fragment to zero. This allows to do a symmetry measure of the part of the MO orbitals that is located around the fragment.

```
# define the atoms of the fragment
atoms_list = [0, 1, 2, 3, 4, 5]

# get coordinates of the fragment
coord_fragment = electronic_structure['structure'].get_coordinates(fragment=atoms_
↪list)

# get the plane and orientation of the fragment
center, normal = get_plane(coord_fragment)

# Set zero to all coefficients centered in the atoms that are not part of the fragment
electronic_structure_fragment = crop_electronic_structure(electronic_structure, atoms_
↪list)

# get classified orbitals
orbital_types = get_orbital_classification(electronic_structure_fragment,
                                           center=center,
                                           orientation=normal)
```

CHAPTER 6

Error handling

In the previous sections it is always assumed that Q-Chem calculations always finish successfully, but, in general, this is not true. Calculation may fail for several reasons, incorrect input parameters, slow convergence, problems with the cluster, issues during the parser process, etc.. To handle these errors PyQchem implements two custom error classes: **OutputError** and **ParserError**.

These errors are risen during the *get_output_from_qchem* function execution if something has gone wrong. If **OutputError** is risen means that Q-Chem calculation did not finish correctly. This may mean that the input is incorrect or something happened with the computer that lead the calculation to crash. When this error is risen the traceback will show the last 20 lines of the Q-Chem output where, hopefully, you will find enough information to determine the cause of the error.

```
Traceback (most recent call last):
  File "/Users/abel/PycharmProjects/qchem_scripts/scripts/test_tddft.py", line 51, in
↪<module>
    store_full_output=True,
  File "/Users/abel/PycharmProjects/qchem_scripts/pyqchem/qchem_core.py", line 330,␣
↪in get_output_from_qchem
    raise OutputError(output, err)
pyqchem.errors.OutputError: Error in Q-Chem calculation:
sh: gdb: command not found
Unable to call dbx in QTraceback: No such file or directory
 http://arma.sourceforge.net/

 Q-Chem begins on Fri Jun  5 02:31:23 2020

Host:
0

     Scratch files written to /Users/abel/Scratch/export/qchem66428//

 Name = B3LP

 Q-Chem fatal error occurred in module libdft/dftcodes.C, line 599:
```

(continues on next page)

(continued from previous page)

```
 Unrecognized exchange functional in XCode


 Please submit a crash report at q-chem.com/reporter
```

In some cases, specially when working with complex workflows it becomes useful to be able to capture these errors to handle them automatically without finish all the workflow. This can be done by the usual try/except statements. By capturing the error it is possible to skip the calculation, recover the error_lines and even recover the full Q-Chem output:

```
try:
    parsed_data = get_output_from_qchem(qc_input,
                                        processors=4,
                                        parser=basic_optimization)
except OutputError as e:
    print('These are the error lines:\n', e.error_lines)
    print('This is the full output:\n', e.full_output)
```

Using the full output it is possible to try to parse the information that contains by applying the parser directly:

```
try:
    parsed_data = get_output_from_qchem(qc_input,
                                        processors=4,
                                        parser=basic_optimization)
except OutputError as e:
    print('Something wrong happened!:\n', e.error_lines)
    print('Recovering usable data...')
    parsed_data = basic_optimization(e.full_output)
```

But if the calculation is really incomplete, or the format of Q-Chem output is incompatible with the parser, the parsing process may also fail and a **ParserError** will be risen. In this case the output data cannot be recovered using this parser.

In the same way as **OutputError** a try/except block can be written to capture this error. A sensible to proceed can be either skip the calculation or try another parser by nesting two try/except blocks :

```
try:
    parsed_data = get_output_from_qchem(qc_input,
                                        processors=4,
                                        parser=basic_optimization)
except OutputError as e:
    print('Something wrong happened!:\n', e.error_lines)
    print('Recovering usable data...')

    try:
        parsed_data = basic_optimization(e.full_output)
    except ParserError:
        print('Trying another parser')
        parsed_data = other_parser(e.full_output)
```

CHAPTER 7

Extra information

7.1 Electronic structure

The electronic structure dictionary is designed to contain the data from sources other than the output. Most of its contents are from the fchk file, but also contains data from scratch files such as the hessian and the fock matrix. Other data will be included in this dictionary in the future.

The basic structure of the electronic structure dictionary is the following:

```
root
  ├── basis
  │     ├── name
  │     ├── primitive_type
  │     └── atoms(list)
  │           ├── shells (list)
  │           ├── symbol
  │           └── atomic_number
  ├── coefficients
  │     ├── alpha
  │     └── beta (optional)
  ├── mo_energies
  │     ├── alpha
  │     └── beta (optional)
  ├── number_of_electrons
  │     ├── alpha
  │     └── beta
  ├── nato_coefficients (optional)
  │     ├── alpha
  │     └── beta
  ├── nato_occupancies (optional)
  │     ├── alpha
  │     └── beta
  ├── structure
  └── overlap
```

Using the information of this dictionary a Fchk file can be generated. This may be used to visualize the molecular orbitals, electronic density and other properties using an (external) visualization program.

```
from pyqchem.file_io import build_fchk
with open('file.fchk', 'w') as f:
    f.write(build_fchk(electronic_structure))
```

While electronic structure is a simple dictionary, its elements are designed to be interoperable along the pyqchem code such as guess and basis. Some examples of this interoperability can be found in the examples folder.

CHAPTER 8

Tutorial

PyQchem is python interface for Q-Chem. It allows to create Q-Chem inputs, execute Q-Chem from python, parse its outputs and store the results in convenient python dictionaries. This is especially useful to create complex workflows to automate frequent tasks using python programming language.

As a preparation for the incoming talk I prepared a series of exercises to introduce the very basics of PyQchem. These exercises are preceded by a detailed explanation of the PyQchem functionality that you may need to complete them. For further information you can check the PyQchem manual available online at: https://pyqchem.readthedocs.io/

Once you complete these exercises, you will be able to:

- Submit a simple Q-Chem calculation from python and obtain the desired results
- Use a simple loops to automate the creation of q-Chem inputs.
- Combine two Q-Chem calculation by using the outputs of the former in the input of the later.

8.1 Execution

In order to run Q-Chem calculations using PyQchem a installation of Q-Chem is necessary. Q-Chem is currently installed in ATLAS cluster along with PyQchem, hence if you use ATLAS it is not necessary any further installation to complete these exercises. PyQchem will be loaded along Q-Chem when loading qchem_group/qchem_trunk modules as usual

```
export MODULEPATH=/scratch/user/SOFTWARE/privatemodules:$MODULEPATH
export QCSCRATCH=/scratch/user/QCHEM_SCRATCH
module load qchem_group

python script.py > output.txt
```

Note: Since the exercises contained in this document are quite short, you can run them directly in ATLAS (so you don't have to wait for the queue system) just connect to ATLAS and export MODULEPATH (as shown above) and load qchem_group module.

However, it may be useful to have PyQchem installed locally in your computer to prepare scripts, specially when using some advanced python editors like PyCharm, Clion, VCode, etc. PyQchem can be downloaded and installed in a MAC/Linux machine from the official python repository (https://pypi.org) by using the command

```
pip install pyqchem
```

In some python installations this command may need sudo permissions. If this is the case, then you can specify user's home installation path by

```
pip install pyqchem --user
```

this way the installation will be done in your home and will not require superuser permissions.

8.2 Basic concepts

In order to perform a Q-Chem calculation, two main pieces of information are needed as input:

1) the molecular structure
2) the parameters of the calculation (method/basis set/etc..)

To define the molecular structure, PyQchem uses a *Structure* class that can be imported as:

```
from pyqchem import Structure
```

Using this class you can create an instance of this class. An instance can be understood as a variable with type *Structure*. To create an instance it is necessary to initialize it with the necessary parameters. In the case of *Structure* class these parameters are *coordinates*, *symbols*, *charge* and *multiplicity*:

```
from pyqchem import Structure

hydroxide = Structure(coordinates=[[0.0, 0.0, 0.0],
                                   [0.0, 0.0, 0.9]],

                      symbols=['O', 'H'],
                      charge=-1,
                      multiplicity=1)
```

As can be seen in the example above, both *coordinates* and *symbols* are simple python lists separated by comma and limited by **[** and **]** characters. In the case of *symbols* list, its elements are character strings which are surrounded by quotes. *charge* and *multiplicity* are simple integer numbers and are optional parameters. In case of not being defined, charge will be set to 0 and multiplicity to 1.

This instance has all the methods defined for the *Structure* class. Methods can be understood as functions associated to a particular class instance. To access to these methods the operator '.' is used followed by the method's name. The following example shows some of the methods defined in *Structure* class:

```
# variables
ne = hydroxide.number_of_electrons
alpha = hydroxide.alpha_electrons
beta = hydroxide.beta_electrons

print('Number of electrons:', ne, '(', alpha, beta, ')')

# functions
xyz_file_txt = hydroxide.get_xyz(title='hydroxide anion')
print(xyz_file_txt)
```

Note: In this example xyz_file_txt is a string that is printed on screen using *print()* function. You can write this string into a file using python language, for example:

```
open('file_name.xyz').write(xyz_file_txt)
```

The definition of the parameters is done by the *QchemInput* class. Using this class we define an instance of this class as:

```
from pyqchem import QchemInput

oh_input = QchemInput(molecule,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G',
                      unrestricted=True)
```

In a similar way as the *Structure* class, to initialize a *QchemInput* instance several parameters are necessary. In this case the first parameter is *molecule*. *molecule* is an instance of the *Structure* class, like the one that we defined before (hydroxide). All the other parameters are optional and have default parameters in case of not being defined. The name of these parameters is designed to be equal or similar to the respective Q-Chem keywords. The list of available parameters is being updated continuously and can be found in: https://github.com/abelcarreras/PyQchem/blob/master/pyqchem/qc_input.py

As in the case of *Structure* class, several methods are defined for *QchemInput*. The main one is **get_txt()**. This method returns a string containing the input in Q-Chem format. This can be used to check the exact input that will be submitted to Q-Chem to do the calculation.

```
input_txt = oh_input.get_txt()
print(input_txt)
```

Other useful methods are **get_copy()** and **update_input()**. These methods are useful to modify already created inputs. For example, in case you want to prepare multiple different inputs with few differences you can create a general input, make multiple copies of it and modify them:

```
input_txt = oh_input.get_txt()
print(input_txt)


general_input = QchemInput(molecule,
                           jobtype='sp',
                           exchange='hf')

input_basis_1 = general_input.get_copy()
input_basis_2 = general_input.get_copy()

input_basis_1.update_input({'basis': 'sto-3g', 'mem_total' : 2000})
input_basis_2.update_input({'basis': '6-31G', 'mem_total' : 1000})
```

Finally, to run the calculation **get_output_from_qchem** function is used. The first argument of this function is a *QchemInput* instance. There are several optional parameters for this function mainly related to computer stuff (which do not affect the results of the calculation). A good representative is *processors*, that indicate the number of processor cores to use in the calculation (in openMP compilation) or the number of MPI processes (in MPI compilation).

Note: In ATLAS cluster Q-Chem is compiled using openMP.

```
from pyqchem import get_output_from_qchem

output = get_output_from_qchem(oh_input,
                               processors=4)

print(output)
```

The output of this function is a string containing the full Q-Chem output. In this example the output is printed in the screen. Combining all these functions together we obtain a simple script that runs a single Q-Chem calculation and prints its output:

```
from pyqchem import Structure, QchemInput, get_output_from_qchem

hydroxide = Structure(coordinates=[[0.0, 0.0, 0.0],
                                   [0.0, 0.0, 0.9]],

                      symbols=['O', 'H'],
                      charge=-1,
                      multiplicity=1)

oh_input = QchemInput(hydroxide,
                      jobtype='sp',
                      exchange='hf',
                      basis='6-31G',
                      unrestricted=True)


output = get_output_from_qchem(oh_input,
                               processors=4)

print(output)
```

8.2.1 Practical exercises

a) Use PyQchem to write a script that generates a set of Q-Chem inputs to do a HF calculation of the methane molecule using the following basis sets: STO-3G, 6-31G, DZ, cc-pVDZ and aug-cc-pVDZ. Use python's **open()** function to store these inputs in different files.

b) Modify the previous script to run the generated inputs using **get_output_from_qchem()** function to obtain the corresponding Q-Chem outputs. Store these outputs in different files.

c) [ADVANCED] Make use of python language tools such as *list comprehension* and *for/while* loops to make this exercise, obtaining a cleaner and more extendable code.

8.3 Parsing data

Being able to automatically generate Q-Chem inputs and outputs can be pretty useful. However the key feature of PyQchem is the use of parsers to extract the output information. A parser is just a function that takes a text string, finds the important data and places it in an organized structure. In the case of PyQchem this structure is a python dictionary.

Here an example of such a function:

```
def parser_example(output):
    data_dict = {}
    enum = output.find('Total energy in the final basis set')
    data_dict['scf_energy'] = float(output[enum: enum+100].split()[8])
    return data_dict

print(output)
```

This function does 3 main things:

- Define a python dictionary using {} syntax.
- Get the location of the data that we are interested in the output, in this case the SCF energy.
- Convert the interesting data from text format to number format using *float()* function and store them in the dictionary.

Note: The use of *[ini: fin]* in strings to get a substring is called slicing. This is very useful in parser functions since you can divide a long output string in small strings that contain the data. On the other hand *split()* method divides a text in words and generates a list that can be accessed by indices.

```
text = 'this may be a long text with lots of words'
subtext = text[0: 11]   # Result: 'this may be'
words = subtext.split() # Result: ['this', 'may', 'be']
word = words[1]         # Result: 'may'
```

Note: In contrast to other languages like Fortran Python indices start from 0 (not 1!).

Parser functions can be explicitly written in the python script just after getting the Q-Chem output:

```
def parser_example(output):
    data_dict = {}
    enum = output.find('Total energy in the final basis set')
    data_dict['scf_energy'] = float(output[enum: enum+100].split()[8])
    return data_dict

(...)

output = get_output_from_qchem(oh_input)

parsed_data = parser_example(output)
print(parsed_data)  # Result: {'scf_energy': 1.234567}
```

The above example will print a dictionary with a single item with key **'scf_energy'** and the energy as a value. A python dictionary works in a similar way as list/vectors, but instead of accessing the elements with an integer index we use a key string.

```
energy = parser_data['scf_energy']
print ('The energy is ', energy)
```

As may be expected, a dictionary can contain multiple items so accessing them via keys is a basic functionality. The values of a dictionary can be almost anything: strings, numbers, lists ... and even other dictionaries. This generates a very common structure of dictionaries inside dictionaries used to organize the data in a tree-like structure.

```
sub_dict = {}
dict = {}

sub_dict['inside'] = [4, 3, 5]
dict['outside'] = sub_dict

print(dict['outside']['inside'])  # Result: [4, 3, 5]
print(dict['outside']['inside'][2])  # Result: 5
```

Note: Technically a dictionary key can be other objects aside from strings but to make it simple we will use strings.

The use of parsers in PyQchem is kind of a basic feature, for this reason **get_output_from_qchem()** function has an optional argument that requires a parser function:

```
def parser_example(output):
    data_dict = {}
    enum = output.find('Total energy in the final basis set')
    data_dict['scf_energy'] = float(output[enum: enum+100].split()[8])
    return data_dict

(...)

parsed_data = get_output_from_qchem(oh_input, parser=parser_example)

print(parsed_data)  # Result: {'scf_energy': 1.234567}
```

As can be observed in the example above, using parser argument transforms the output of **get_output_from_qchem** into a dictionary with the parsed output. This makes the script shorter and cleaner. PyQchem package includes parsers written for the most common types of calculations. These can be found in the parsers folder: (https://github.com/abelcarreras/PyQchem/tree/master/pyqchem/parsers). To use them, you just need to use *import* statement:

```
from pyqchem.parsers.basic import basic_parser_qchem

(...)

parsed_data = get_output_from_qchem(oh_input,
                                    parser=basic_parser_qchem)
```

8.3.1 Practical exercises

a) Create a parser function to get the following properties from a HF calculation: *Sum of atomic charges* & *Sum of spin charges*. You can use the same system as in the fist example (methane with STO-3G basis set) to test it.

b) Use the basic parser included in PyQchem (*basic_parser_qchem*) to write a script that calculates and the orbital energies of methane molecule.

c) [ADVANCED] Use a loop (*for/while*) to calculate the scf energy of the hydrogen molecule at different geometries (bond length) to study the dissociation of hydrogen molecule. Print the results as two columns (bond length and scf energy)

Note: During the execution a *calculation_data.pkl* file is generated. This stores data of previous calculations (see manual for more information). Modifying the parser may make this data obsolete, if something unexpected happens modifying the parser try removing this file. Also, see *force_recalculation=True* argument of

get_output_from_qchem() function.

8.4 Linking calculations

One of the strongest reasons to use a library like PyQchem is the ability to link different calculations together. This means prepare inputs from output data of previous calculations. A typical example is the calculation of the normal modes frequencies of a previously optimized structure. This can be done in PyQchem in the following way:

```
from pyqchem.parsers.parser_frequencies import basic_frequencies
from pyqchem.parsers.parser_optimization import basic_optimization

(...)

opt_input = QchemInput(molecule,
                       jobtype='opt',
                       exchange='hf',
                       basis='sto-3g')

parsed_opt_data = get_output_from_qchem(opt_input, parser=basic_optimization)

opt_molecule = parsed_opt_data['optimized_molecule']

freq_input = QchemInput(opt_molecule,
                        jobtype='freq',
                        exchange='hf',
                        basis='sto-3g')

parsed_data = get_output_from_qchem(freq_input, parser=basic_frequencies)

print(parsed_data)
```

In this example, the optimized structure is obtained from the parsed output of the optimization calculation. In this parser the value of **'optimize_molecule'** entry is already an instance of the *Structure* class so it can be used directly in the frequencies calculation input.

This script is pretty convenient but it can be done even better. In order to take maximum profit of a previous calculation, the already optimized electronic structure (molecular orbitals) can be used as a initial guess in the frequencies calculation. To do this, it is necessary to get the orbitals coefficients, which are not present in the usual output. PyQchem obtains the electronic structure data from the *FChk* file. The request of the *FChk* generation is done directly in the **get_output_from_qchem** function by using the argument *return_electronic_structure=True*. This modifies the output of this function returning two pieces of data (a list of two elements): the parsed output & the parsed *FChK* data:

```
from pyqchem.parsers.parser_frequencies import basic_frequencies
from pyqchem.parsers.parser_optimization import basic_optimization

(...)

parsed_opt_data, electronic_structure = get_output_from_qchem(opt_input,
                                                              parser=basic_
↪optimization,
                                                              return_electronic_
↪structure=True)

print(electronic_structure)
```

if you print *electronic_structure* you will notice that it is a dictionary containing the entries of a usual FChk file. Due to the standard format of this file all data is parsed so it is not necessary to indicate a parser. The format of this dictionary is designed for inter-operation with the different functions of PyQchem. A simple example is the use of the molecular orbitals coefficients as an initial guess:

```
mo_coefficients = electronic_structure['coefficients']

freq_input = QchemInput(opt_molecule,
                        jobtype='freq',
                        exchange='hf',
                        basis='sto-3g',
                        scf_guess=mo_coefficients)
```

In this case, **electronic_structure['coefficients']** contains a NxN square matrix with the coefficients of the molecular orbitals where each row corresponds to a molecular orbital.

8.4.1 Practical exercises

a) Use PyQchem to optimize the water molecule (H2O) using HF and minimum basis set (STO-3G). From the optimized structure perform 3 additional optimizations using larger 3 different basis sets: SV, DZ & TZ. Get the scf_energies from each optimization and store the optimized structures in XYZ files.

b) (ADVANCED) Perform a frequencies calculation of the methane molecule (CH4) using PyQchem (with HF/STO-3G) and create a movie in a XYZ file that shows the vibration of each normal mode. Print the results of the **basic_frequencies** parser and investigate its contents to find the necessary information (*displacements*). (https://github.com/abelcarreras/PyQchem/blob/master/pyqchem/parsers/parser_frequencies.py)

Note: To create a movie in XYZ just put all the geometries (one under the other) in the same file. This will be interpreted in most molecular visualization software (Ex. VMD) as frames and you will be able to reproduce them as a movie.

HINT: In python you can combine two strings by the + operator. Ex:

```
video_xyz = frame1_xyz + frame2_xyz + frame3_xyz
```

8.5 The cache system

PyQchem provides a cache system to avoid redundant calculations. This system works seamless in the background storing the data of previous calculations in a cache file (by default: calculation_data.db). This is an SQL database file that can be opened/edited using SQL-compatible utilities. To manually access to with this file PyQchem provides a simple ORM class. Here an example about how to use:

1. Load the cache file

```
from pyqchem.cache import SqlCache

cache = SqlCache(filename='calculation_data.db')
```

2. List the data

```
cache.list_database()
```

output:

```
           ID                       KEYWORD                      DATE
-------------------------------------------------------------------------------
1837001741792534522       basic_optimization        2022-09-22 17:04:17.809714
1922769254804187209       fchk                      2022-09-22 17:14:42.626054
1922769254804187209       basic_parser_qchem        2022-09-22 17:14:42.628925
861204412201835456        fchk                      2022-09-22 17:15:12.654232
861204412201835456        basic_parser_qchem        2022-09-22 17:15:12.657227
924488890157922159        basic_parser_qchem        2022-09-22 17:16:31.919593
```

where *ID* is a unique identifier that corresponds to a particular input, *keyword* is a word to identify a particular output associated to the input (usually corresponds to different parsers) and *date* contains the information relative to the time at which the calculation was performed.

3. Access to the data using *ID* and *keyword*

```
data = cache.retrieve_calculation_data_from_id('1837001741792534522', keyword='basic_
↪optimization')
print(data)
```

data in general contains a Python dictonary with the parsed data.

Note: It is important to note that PyQchem only stores data associated to a particular parser and fchk. The name of the parser is obtained from the parser function name. Using two or more parsers with the same exact name may lead to issues.

If a parser is not provided in **get_output_from_qchem** the full output will not be stored by default. For development purpuses it is possible to store the full output defining **store_full_output=True**.

```
full_ouput = get_output_from_qchem(opt_input, store_full_output=True)
```

Using this option the full output of the calculation will be stored in the database. If the calculation is repetaed using a parser, then the output data will be parsed everytime from the stored full output even if the same data has already been parsed. This can be usefull for developing parsers.

Note: Keep in mind that using **store_full_output=True** may rapidly increase the size of the database file.

It is possible to change the name of the database file to be used for a particular calculation. This may be usefull to run multiple simultaneours calculations in the same directory.

```
from pyqchem.qchem_core import redefine_calculation_data_filename
redefine_calculation_data_filename('database_file.db')
```

8.5.1 Database corruption

Ocassionaly, running multiple simulateneous calculations using the same database file, may lead to corruption of the database file. This may also happend if the calculation cashes during the I/O access to the file. If this happends PyQchem provides a method to fix this file recovering (at least partially) the non corrupted data of the data of the file:

1. Check the integrity of the datafile

```
cache.integrity_check()
```

2. Recover the data in the correupted file and store them in a new database file

```
cache.fix_database('recovered_database.db')
```

CHAPTER 9

Public API

9.1 QcInput

class pyqchem.qc_input.**QchemInput**(*molecule*, *jobtype='sp'*, *method=None*, *exchange=None*, *correlation=None*, *unrestricted=None*, *basis='6-31G'*, *basis2=None*, *thresh=14*, *scf_convergence=8*, *max_scf_cycles=50*, *scf_algorithm='diis'*, *purecart=None*, *ras_roots=None*, *ras_do_hole=True*, *ras_do_part=True*, *ras_act=None*, *ras_act_orb=None*, *ras_elec=None*, *ras_elec_alpha=None*, *ras_elec_beta=None*, *ras_occ=None*, *ras_spin_mult=1*, *ras_sts_tm=False*, *ras_fod=False*, *ras_natorb=False*, *ras_natorb_state=None*, *ras_print=1*, *ras_diabatization_scheme=None*, *ras_diabatization_states=None*, *ras_guess=None*, *use_reduced_ras_guess=False*, *ras_omega=400*, *ras_srdft=None*, *ras_srdft_damp=0.5*, *ras_srdft_exc=None*, *ras_srdft_cor=None*, *ras_srdft_spinpol=0*, *calc_soc=False*, *state_analysis=False*, *ee_singlets=False*, *ee_triplets=False*, *cc_trans_prop=False*, *cc_symmetry=True*, *cc_e_conv=None*, *cc_t_conv=None*, *eom_davidson_conv=5*, *cis_convergence=6*, *cis_n_roots=None*, *cis_singlets=False*, *cis_triplets=False*, *cis_ampl_anal=False*, *loc_cis_ov_separate=False*, *er_cis_numstate=0*, *boys_cis_numstate=0*, *cis_diabath_decompose=False*, *max_cis_cycles=30*, *localized_diabatization=None*, *sts_multi_nroots=None*, *cc_state_to_opt=None*, *cis_state_deriv=None*, *RPA=False*, *set_iter=30*, *gui=0*, *geom_opt_dmax=300*, *geom_opt_update=-1*, *geom_opt_linear_angle=165*, *geom_opt_coords=-1*, *geom_opt_tol_gradient=300*, *geom_opt_tol_displacement=1200*, *geom_opt_tol_energy=100*, *geom_opt_max_cycles=50*, *geom_opt_constrains=None*, *solvent_method=None*, *solvent_params=None*, *pcm_params=None*, *rpath_coords=0*, *rpath_direction=1*, *rpath_max_cycles=20*, *rpath_max_stepsize=150*, *rpath_tol_displacement=5000*, *symmetry=True*, *sym_ignore=False*, *nto_pairs=None*, *n_frozen_core=None*, *n_frozen_virt=None*, *n_frozen_virtual=0*, *mem_static=False*, *re*

get_copy()
Get a copy of the input

Returns

get_txt()
get qchem input in plain text

Return string qchem input in plain text

update_input(*dictionary*)
Update the input from data in a dictionary Note: already existing parameters will be overwritten

Parameters **dictionary** – parameters to add

pyqchem.qc_input.**normalize_values**(*value*)
Set all string values (including keys and values of dictionaries) to lower case

Parameters **value** – the values

Returns normalized values

9.2 Structure

class pyqchem.structure.**Structure**(*coordinates=None*, *symbols=None*, *atomic_numbers=None*, *connectivity=None*, *charge=0*, *multiplicity=1*, *name=None*)
Structure object containing all the geometric data of the molecule

alpha_electrons
returns the alpha electrons

Returns number of alpha electrons

beta_electrons
returns the number of beta electrons

Returns number of beta electrons

charge
returns the charge :return: the charge

get_atomic_masses()
get the atomic masses of the atoms of the molecule

Returns list of atomic masses

get_atomic_numbers()
get the atomic numbers of the atoms of the molecule

Returns list with the atomic numbers

get_connectivity(*thresh=1.2*)
get the connectivity as a list of pairs of indices of atoms from atomic radii

Parameters **thresh** – radii threshold used to determine the connectivity

Returns

get_coordinates(*fragment=None*)
gets the cartesian coordinates

Parameters **fragment** – list of atoms that are part of the fragment

Returns coordinates list

get_number_of_atoms()
get the number of atoms

Returns number of atoms

get_point_symmetry()
Returns the point group of the molecule using pymatgen

Returns point symmetry label

get_symbols()
get the atomic element symbols of the atoms of the molecule

Returns list of symbols

get_valence_electrons()
get number of valence electrons

Returns number of valence electrons

get_xyz(*title=''*)
generates a XYZ formatted file

Parameters **title** – title of the molecule

Returns string with the formatted XYZ file

multiplicity
returns the multiplicity

Returns the multiplicity

name
returns the name :return: structure name

number_of_electrons
returns the total number of electrons

Returns number of total electrons

set_coordinates(*coordinates*)
sets the cartessian coordinates

Parameters **coordinates** – cartesian coordinates matrix

9.3 QcCore

pyqchem.qchem_core.**generate_additional_files**(*input_qchem*, *work_dir*)
Generate additional files on scratch (work dir) for special calculations

Parameters

- **input_qchem** – QChem input object
- **work_dir** – scratch directory

pyqchem.qchem_core.**get_output_from_qchem**(*input_qchem*, *processors=1*, *use_mpi=False*, *scratch=None*, *read_fchk=False*, *return_electronic_structure=False*, *parser=None*, *parser_parameters=None*, *force_recalculation=False*, *fchk_only=False*, *store_full_output=False*, *delete_scratch=True*, *remote=None*, *scratch_read_level=0*)

Runs qchem and returns the output in the following format:

1) **If return_electronic_structure is requested:** [output, parsed_fchk]
2) **If return_electronic_structure is not requested:** [output]

Note: if parser is set then output contains a dictionary with the parsed info else output contains the q-chem output in plain text

Parameters

- **input_qchem** – QcInput object containing the Q-Chem input
- **processors** – number of threads/processors to use in the calculation
- **use_mpi** – If False use OpenMP (threads) else use MPI (processors)
- **scratch** – Full Q-Chem scratch directory path. If None read from $QCSCRATCH
- **return_electronic_structure** – if True, returns the parsed FCHK file containing the electronic structure
- **read_fchk** – same as return_electronic_structure (to be deprecated)
- **parser** – function to use to parse the Q-Chem output
- **parser_parameters** – additional parameters that parser function may have
- **force_recalculation** – Force to recalculate even identical calculation has already performed
- **fchk_only** – If true, returns electronic structure data from cache ignoring output (to be deprecated)
- **remote** – dictionary containing the data for remote calculation (beta)
- **store_full_output** – store full output in plain text in pkl file
- **delete_scratch** – delete all scratch files when calculation is finished

Returns output [, electronic_structure]

pyqchem.qchem_core.**local_run**(*input_file_name*, *work_dir*, *fchk_file*, *use_mpi=False*, *processors=1*)

Run Q-Chem locally

Parameters

- **input_file_name** – Q-Chem input file in plain text format
- **work_dir** – Scratch directory where calculation run
- **fchk_file** – filename of fchk
- **use_mpi** – use mpi instead of openmp

Returns output, err: Q-Chem standard output and standard error

pyqchem.qchem_core.**local_run_stream**(*input_file_name*, *work_dir*, *fchk_file*, *use_mpi=False*, *processors=1*, *print_stream=True*)

Run Q-Chem locally

Parameters

- **input_file_name** – Q-Chem input file in plain text format
- **work_dir** – Scratch directory where calculation run
- **fchk_file** – filename of fchk
- **use_mpi** – use mpi instead of openmp
- **print_stream** – set True to print output stream during execution

Returns output, err: Q-Chem standard output and standard error

pyqchem.qchem_core.**parse_output**(*get_output_function*)

to be deprecated

Parameters **get_output_function** –

Returns parsed output

pyqchem.qchem_core.**remote_run**(*input_file_name*, *work_dir*, *fchk_file*, *remote_params*, *use_mpi=False*, *processors=1*)

Run Q-Chem remotely

Parameters

- **input_file** – Q-Chem input file in plain text format
- **work_dir** – Scratch directory where calculation run
- **fchk_file** – filename of fchk
- **remote_params** – connection parameters for paramiko
- **use_mpi** – use mpi instead of openmp

Returns output, err: Q-Chem standard output and standard error

pyqchem.qchem_core.**retrieve_additional_files**(*input_qchem*, *data_fchk*, *work_dir*, *scratch_read_level=0*)

retrieve data from files in scratch data (on development, currently for test only)

Parameters

- **input_qchem** – QChem input object
- **data_fchk** – FCHK parsed dictionary
- **work_dir** – scratch directory
- **scratch_read_level** – defines what data to retrieve

Returns dictionary with additional data

9.4 Utils

pyqchem.utils.**get_inertia**(*structure*)

returns the inertia moments and main axis of inertia (in rows)

Parameters **structure** – Structure object containg the molecule

Returns eigenvalues, eigenvectors

pyqchem.utils.**get_order_states_list**(*states*, *eps_moment=0.1*, *eps_energy=0.05*)
set higher dipole moment states first if energy gap is lower than eps_energy

pyqchem.utils.**get_plane**(*coords*, *direction=None*)
Returns the center and normal vector of tha plane formed by a list of atomic coordinates

Parameters **coords** – List of atomic coordinates

Returns center, normal_vector

pyqchem.utils.**get_sdm**(*matrix_1*, *matrix_2*)
get differences square matrix between to matrices :param matrix_1: the matrix 1 :param matrix_2: the matrix 2 :return: difference square matrix

pyqchem.utils.**is_rasci_transition**(*configuration*, *reference*, *n_electron=1*, *max_jump=10*)
Determine if a configuration corresponds to a transition of n_electron

Parameters

- **configuration** – dictionary containing the configuration to be analyzed
- **reference** – reference configuration (in general lowest energy Slater determinant)
- **n_electron** –
- **max_jump** – Restrict to transitions with jumps less or equal to max_jump orbitals

Returns True if conditions are met, otherwise False

pyqchem.utils.**is_transition**(*configuration*, *reference*, *n_electron=1*, *max_jump=10*)
Determine if a configuration corresponds to a transition of n_electron

Parameters

- **configuration** – dictionary containing the configuration to be analyzed
- **reference** – reference configuration (in general lowest energy Slater determinant)
- **n_electron** – number of electrons in the transition
- **max_jump** – Restrict to transitions with jumps less or equal to max_jump orbitals

Returns True if conditions are met, otherwise False

pyqchem.utils.**reorder_coefficients**(*occupations*, *coefficients*)
Reorder the coefficients according to occupations. Occupated orbitals will be grouped at the beginning non occupied will be attached at the end

Parameters

- **occupations** – list on integers (0 or 1) or list of Boolean
- **coefficients** – dictionary containing the molecular orbitals coefficients {'alpha': coeff, 'beta:' coeff}. coeff should be a list of lists (Norb x NBas)

Returns

pyqchem.tools.**get_geometry_from_pubchem**(*entry*, *type='name'*)
Get structure form PubChem database

Parameters

- **entry** – entry data
- **type** – data type: 'name', 'cid'

Returns Structure

pyqchem.tools.**plot_rasci_state_configurations**(*states*)
Prints :param states: parsed data (excited states) dictionary entry from RASCI calculation :return: None

pyqchem.tools.**print_excited_states**(*parsed_data*, *include_conf_rasci=False*, *include_mulliken_rasci=False*)
Prints excited states in nice format. It works for CIS/TDDFT/RASCI methods

Parameters

- **parsed_data** – parsed data (excited states) dictionary entry from CIS/RASCI/TDDFT calculation
- **include_conf_rasci** – print also configuration data (only RASCI method)
- **include_mulliken_rasci** – print also mulliken analysis (only RASCI method)

Returns None

pyqchem.tools.**rotate_coordinates**(*coordinates*, *angle*, *axis*, *atoms_list=None*, *center=(0, 0, 0)*)
Rotate the coordinates (or range of coordinates) with respect a given axis

Parameters

- **coordinates** – coordinates to rotate
- **angle** – rotation angle in radians
- **axis** – rotation axis
- **atoms_list** – list of atoms to rotate (if None then rotate all)

Returns rotated coordinates

pyqchem.tools.**submit_notice**(*message*, *service='pushbullet'*, *pb_token=None*, *sp_url=None*, *gc_key=None*, *gc_token=None*, *gc_thread=None*, *slack_token=None*, *slack_channel=None*)
Submit a notification using webhooks

Parameters

- **message** – The message to send
- **service** – pushbullet, samepage, google_chat
- **pb_token** – pushbullet token
- **sp_url** – samepage url
- **gc_key** – google chat key
- **gc_token** – google chat token
- **gc_thread** – google chat thread
- **slack_token** – slack bot token (xoxb-xxx.xxx.xxx),
- **slack_channel** – slack channel

Returns server response

pyqchem.tools.geometry.**get_angle**(*coordinates*, *atoms*)
Compute the angle between 3 atoms

Parameters

- **coordinates** – list of coordinates of the molecule

- **atoms** – list of 3 atom indices to use to calculate the angle (from 1 to N)

Returns the angle

pyqchem.tools.geometry.**get_dihedral**(*coordinates*, *atoms*)
Compute the dihedral angle between 4 atoms

Parameters

- **coordinates** – list of coordinates of the molecule
- **atoms** – list of 4 atom indices to use to calculate the dihedral angle (from 1 to N)

Returns the dihedral angle

pyqchem.tools.geometry.**get_distance**(*coordinates*, *atoms*)
Compute the distance between 2 atoms

Parameters

- **coordinates** – list of coordinates of the molecule
- **atoms** – list of 2 atom indices to use to calculate the distance (from 1 to N)

Returns the distance

pyqchem.tools.geometry.**unit_vector**(*vector*)
Compute unit vector from general vector

Parameters **vector** – the vector

Returns the unit vector

CHAPTER 10

Troubleshooting

In recent versions of macOS, the SIP system may prevent python to access DYLD_LIBRARY_PATH environment variable in sub-shells (https://cmsdk.com/python/python-subprocess-call-can-not-find-dylib-in-mac.html). This can be an issue in Q-Chem when compiled using mkl appearing the following error message during execution

```
dyld: Library not loaded: @rpath/libmkl_intel_lp64.dylib
```

The workaround without disabling the SIP system is to symbolic link the library *libmkl_intel_lp64.dylib* to

```
/usr/local/lib
```

which is used by default to find the needed libraries.

PyQchem is being developed by Abel Carreras within David Casanova's group at Donostia International Physics Center (DIPC), Euskadi, Spain.

Python Module Index

p

Index

Q

R

S

U