QCArchive Documentation

Release 1.0

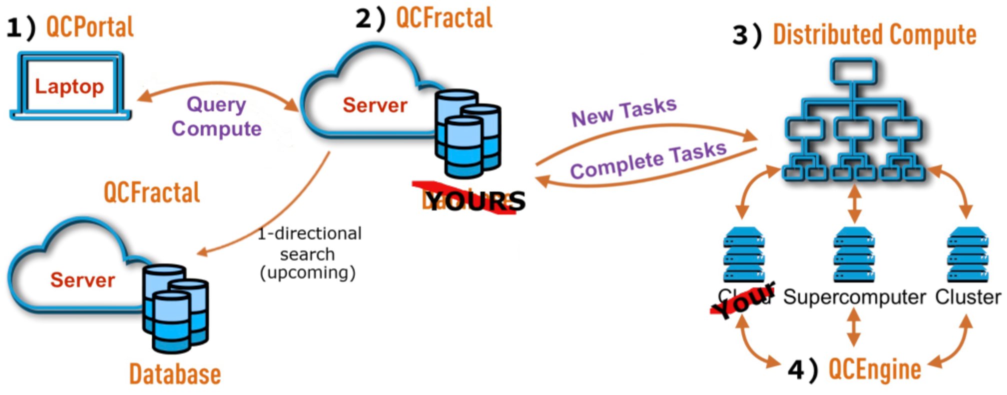
Daniel G. A. Smith

Feb 01, 2020

OVERVIEW

The infrastructure that powers the MolSSI QCArchive is a series of open-source layers that can be used for a variety of projects and use cases. Every piece of software that we develop is available on GitHub and supported by the MolSSI team.

At the outermost scope, the QCArchive infrastructure allows you to privately generate and manage data precisely as we do in QCArchive, running calculation and performing statistical and visual analysis without ever interacting with the MolSSI servers. At a later date (perhaps post-publication), if you would like to upload your data to the MolSSI servers, it is a very quick and easy process!



CHAPTER

ONE

LAYERED INFRASTRUCTURE

While the infrastructure projects can be used together to manage data at scale, each library we support has capabilities that can be used independently:

- QCElemental - Periodic table information, version-controlled physical constants, molecule parsing, testing infrastructure, and MolSSI QCSchema models.
- QCEngine - Quantum chemistry program executor and IO standardizer (QCSchema).
- QCFractal - Distributed task scheduler and executor, database store for chemistry results, and organization of results at scale.
- QCPortal - Visualization, organization, and statistical analysis for chemistry-related results and a front-end client for QCFractal.

CHAPTER

TWO

PROJECT HIGHLIGHTS

A few software projects using the QCArchive infrastructure are:

- Psi4 - A quantum chemistry package which uses QCElemental for physical constants and QCEngine to compute additive properties such as DFTD3.
- geometric - A backend-agnostic geometry optimizer which uses QCEngine to evaluate gradients for a variety of different quantum chemistry-like packages.
- MultirefPredict - Multireference character prediction using QCElemental for model implementations and QCEngine to harness many quantum chemistry backends simultaneously.

CHAPTER

THREE

PROJECT DEMOS

To highlight the capabilities of each project, short demos were created. Full details are available in each project's Documentation page which can be found in the dropdown at the top of this page.

- QCElemental
- QCEngine
- QCFractal
- QCPortal

CHAPTER

FOUR

SUPPORTED BY

The QCArchive infrastructure is supported by the Molecular Science Software Institute and the National Science Foundation.

Additional features are sponsored by our partners:

- Open Force Field Initiative

Consider becoming a feature sponsor today!

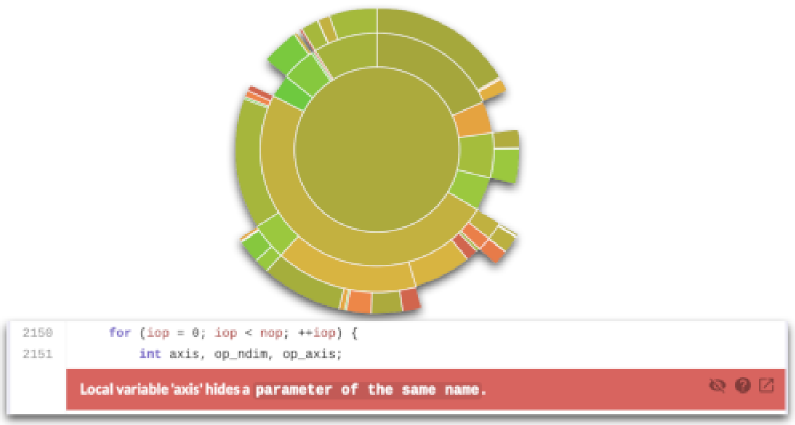
4.1 Software Philosophy

4.1.1 Open-source infrastructure

The QCArchive project is designed from the start to be open-source and available to the computational molecular sciences community. The entire code base is public through GitHub repositories, and we are contributing back to other Open Source projects as good members of the community. Rather than a monolithic piece of software, the QCArchive software infrastructure is decomposed into several components so that it can be used as a full ecosystem to build on top of or in support of community software.

4.1.2 Code quality is of utmost importance

The quality of the QCArchive code is checked through rigorous continuous integration, code linters, and test coverage tools which help reduce the chance that bugs are introduced. We take data accuracy very seriously and all results and calculations are versioned, provenance tracked, and will be preserved over any updates of the database.



```
    for (iop = 0; iop < nop; ++iop) {
        int axis, op_ndim, op_axis;
```

Local variable 'axis' hides a parameter of the same name.

4.1.3 We support software best practices

QCArchive's projects are all based on the same software best practices. Both users and developers can follow the code structure between all of the projects, and these practices are provided back to the community through external projects like the CMS-Cookiecutter. This cookiecutter supports:

- Continuous integration with automated testing and testing coverage.
- Automated versionsing and package distribution via Conda-Forge and PyPI.
- Code quality and linting checks for beutiful code.
- Automatic package organization and documentation setup.

The cookiecutter is available for all Python projects! Check it out here!

4.2 Community

The QCArchive is an open community sponsored by The Molecular Sciences Software Institute. However, this is a community-driven project which requires feature requests, user feedback, and code support. There are a variety of ways to help support the QCArchive project as seen below.

4.2.1 Discussion

- QCArchive Slack is a great place to get feedback and advice from the community. Join our Slack group to get started.
- The QCArchive GitHub repositories contain future roadmaps, current code updates, and a list of issues that are being worked and provide an excellent overview of the development status of the project.

4.2.2 Work with us!

The QCArchive project is actively looking for early collaborations to use our tools, help us shake out the bugs, and be evangelist within the computational molecular science community for this code ecosystem. In return you will receive the following benefits:

- Work directly with MolSSI Software Scientist who will discuss your problem and provide ideas.
- Develop the requirements and potential solutions for your use case within the QCArchive ecosystem.
- Setup monthly meetings to ensure your project stays on track.
- Highlight your project within the QCArchive ecosystem.

If you are interested in working with us, please send an email to QCArchive@molssi.org and we will set up a meeting to discuss specifics.

You can run this notebook online in a session or view it on Github.

4.3 QCElemental

QCElemental is a general purpose utility library that covers many fundamental areas of practical quantum chemistry:

- QCSchema models for input and output
- Molecule parsing
- Unit support
- Canonical physical quantities

Full QCElemental documentation is available.

4.3.1 Molecule Parsing and Models

```
[1]: import qcelemental as qcel
```

The MolSSI QCSchema has been implemented in utility classes within QCElemental. Models can be created in a number of different ways, here we pull caffeine from pubchem.

```
[2]: qcel.models.Molecule.from_data("pubchem:caffeine")
```

```
        Searching PubChem database for caffeine (single best match returned)
        Found 1 result(s)
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[2]: <Molecule(name='IUPAC 1,3,7-trimethylpurine-2,6-dione' formula='C8H10N4O2' hash=
↪'a9e3599')>
```

More canonical forms are also available, for example we can import a geometry from a string:

```
[3]: water = qcel.models.Molecule.from_data("""
-1 2
O 0 0 0
H 0 0 1
H 0 1 0
""")
water
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[3]: <Molecule(name='H2O' formula='H2O' hash='647b11e')>
```

In addition to visualization, there are many helper functions which can provide data about the molecule or can perform a variety of actions. It should be noted that all quantities are in atomic units.

```
[4]: print(f"Molecular Charge:       {water.molecular_charge}")
print(f"Molecular Multiplicity: {water.molecular_multiplicity}")
print(f"O-H distance (Bohr):    {water.measure([0, 1])}")
print(f"H-O-H angle (degrees):  {water.measure([1, 0, 2])}")
print("\nWater coordinates (Bohr):")
print(water.geometry)
```

```
Molecular Charge:       -1.0
Molecular Multiplicity: 2
O-H distance (Bohr):    1.88972613
H-O-H angle (degrees):  90.0

Water coordinates (Bohr):
[[0.         0.         0.        ]
 [0.         0.         1.88972613]
 [0.         1.88972613 0.        ]]
```

4.3.2 Unit Support

Unit conversion between arbitrary units is supported out of the box. By default, 2014 CODATA is used.

```
[5]: qcel.constants.conversion_factor("bohr", "Angstrom")
```

```
[5]: 0.52917721067
```

```
[6]: qcel.constants.bohr2angstroms
```

```
[6]: 0.52917721067
```

Compound units are input with python-like syntax.

```
[7]: qcel.constants.conversion_factor("hartree / bohr ** 2", "eV / inches ** 2")
```

```
[7]: 6.269253498179915e+18
```

In addition to straight conversion factors, QCElemental supports Pint `Quantity` objects. Python floats and numpy arrays are supported for values.

```
[8]: distance = qcel.constants.Quantity("3 Bohr")
     print(distance)
     print(distance.to("mile"))
```

```
3 bohr
9.864464228965342e-14 mile
```

QCElemental's implementation of a Pint context includes many computational chemistry specific translations.

```
[9]: print(f"hartree -> kelvin:   {qcel.constants.conversion_factor('hartree', 'kelvin')}")
     print(f"hartree -> kilogram: {qcel.constants.conversion_factor('hartree', 'kilogram')}
     ↪")
```

```
hartree -> kelvin:   315775.13
hartree -> kilogram: 4.850870129000001e-35
```

4.3.3 Physical Quantities

QCElemental can provide many canonical quantities from a variety of standard names:

[10]:
```
print(qcel.periodictable.to_mass(6))
print(qcel.periodictable.to_mass("C"))
print(qcel.periodictable.to_mass("C12"))
print(qcel.periodictable.to_mass("Carbon"))
```

```
12.0
12.0
12.0
12.0
```

4.3.4 Conclusion

These are some of the capabilities QCElemental offers, check out more documentation. If you like the project, consider starring us on GitHub or if you have any questions, join our Slack channel.

You can run this notebook online in a session or view it on Github.

4.4 QCEngine

Full QCEngine documentation is available.

QCEngine is a quantum chemistry abstraction layer where many different quantum chemistry (or quantum-chemistry-like!) programs can be run with identical input and output abstractions that match the MolSSI QCSchema.

Begin by importing `qcengine`.

[1]:
```
import qcelemental as qcel
import qcengine as qcng
```

We can list all programs that QCEngine currently supports. It should be noted that there are many programs which provide force field or machine learning potential evaluation (e.g. `rdkit` and `torchani`) in addition to the traditional quantum chemistry programs.

[2]:
```
qcng.list_all_programs()
```

[2]:
```
{'dftd3',
 'entos',
 'molpro',
 'mopac',
 'mp2d',
 'psi4',
 'rdkit',
 'terachem',
 'torchani'}
```

We can then list all programs that QCEngine has detected on the current resource. This list will vary depending on installed packages. As a note, QCEngine does not install programs by default, and these must be installed separately.

[3]:
```
qcng.list_available_programs()
```

[3]:
```
{'dftd3', 'mopac', 'psi4', 'rdkit'}
```

4.4.1 Single Computations

QCEngine makes the distinction between a "single" evaluation which corresponds to a single molecular geometry and a "procedure" which involves multiple geometries or multiple molecules. "Single" evaluations include energy, gradient, Hessian, and property quantities. "Procedures" include geometry optimization and other complex multi-step procedures.

First, we can build a Molecule object using the QCElemental molecule builder:

```
[4]: mol = qcel.models.Molecule(geometry=[[0, 0, 0], [0, 1.5, 0], [0, 0, 1.5]],
                                symbols=["O", "H", "H"],
                                connectivity=[[0, 1, 1], [0, 2, 1]])
     mol
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[4]: <Molecule(name='H2O' formula='H2O' hash='b41d0c5')>
```

We can then provide minimal input for a quantum chemistry job which specifies the molecule, driver, and model that the computation should be run under:

```
[5]: computation = {
         "molecule": mol,
         "driver": "energy",
         "model": {"method": "B3LYP", "basis": "6-31g"}
     }
     ret = qcng.compute(computation, "psi4")
```

The result contains many attributes that hold relevant data. We can access the `return_result` which contains the desired value as determined by the `driver` input field. In this case, it is the B3LYP/6-31g energy (in Hartree):

```
[6]: ret.return_result
```

```
[6]: -76.2741297206346
```

QCEngine automatically parses additional data about the state of the computation and pulls several other component fields. Here we can see the energy breakdown as well as the basis information:

```
[7]: ret.properties.dict()
```

```
[7]: {'calcinfo_nbasis': 13,
      'calcinfo_nmo': 13,
      'calcinfo_nalpha': 5,
      'calcinfo_nbeta': 5,
      'calcinfo_natom': 3,
      'nuclear_repulsion_energy': 11.138071187457696,
      'return_energy': -76.2741297206346,
      'scf_one_electron_energy': -126.25159666378747,
      'scf_two_electron_energy': 46.556895182916136,
      'scf_xc_energy': -7.717499427220989,
      'scf_dipole_moment': [0.0, 0.0, 2.660795024634264],
      'scf_total_energy': -76.2741297206346,
      'scf_iterations': 6}
```

Finally, QCEngine records much of the state of the computation such as the hardware it was run on, the program it was run with, and the versions of programs used:

```
[8]: ret.provenance.dict()
```

```
[8]: {'creator': 'Psi4',
      'version': '1.3.2',
      'routine': 'psi4.json.run_json',
      'memory': 2.266,
      'nthreads': 2,
      'qcengine_version': 'v0.9.0',
      'wall_time': 2.847166061401367,
      'hostname': 'Daniels-MacBook-Pro.local',
      'cpu': 'Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz',
      'username': 'daniel'}
```

4.4.2 Procedures

Since we created a pretty poor molecule to start with, we should optimize it first under a force field method to have a reasonable geometry. Here, we will use the standalone `geomeTRIC` package coupled with the `rdkit` force field evaluator.

```
[9]: oh_bond, hoh_angle = mol.measure([[0, 1], [1, 0, 2]])
     print(f"O-H Bond length (Bohr): {oh_bond}")
     print(f"H-O-H Angle (degrees):  {hoh_angle}")
```

```
O-H Bond length (Bohr): 1.5
H-O-H Angle (degrees):  90.0
```

```
[10]: opt_input = {
          "keywords": {
              "program": "rdkit"
          },
          "input_specification": {
              "driver": "gradient",
              "model": {"method": "UFF"},
          },
          "initial_molecule": mol
      }
      opt = qcng.compute_procedure(opt_input, "geometric")
      opt
```

```
[10]: <Optimization(model='{'method': 'UFF', 'basis': None}' molecule_hash='b41d0c5')>
```

We can first check the geometry of the final molecule and see that it is something much more reasonable:

```
[11]: opt_mol = opt.final_molecule
      oh_bond, hoh_angle = opt_mol.measure([[0, 1], [1, 0, 2]])
      print(f"O-H Bond length (Bohr): {oh_bond}")
      print(f"H-O-H Angle (degrees):  {hoh_angle}")
```

```
O-H Bond length (Bohr): 1.8713297962085038
H-O-H Angle (degrees):  104.5102429904286
```

We can explore additional data generated with this geometry optimization including details of every gradient evaluation performed:

```
[12]: opt.trajectory
```

```
[12]: [<Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'b41d0c5')>,
       <Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'1ad5fe3')>,
       <Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'04ec4cf')>,
       <Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'58054eb')>,
       <Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'f2a154b')>,
       <Result(driver='gradient' model='{'method': 'UFF', 'basis': None}' molecule_hash=
      ↪'3c65f4c')>]
```

If desired, we can also look at the standard output of the `geomeTRIC` program:

```
[13]: print(opt.stdout)
```

```
9 internal coordinates being used (instead of 9 Cartesians)
Internal coordinate system (atoms numbered from 1):
Distance 1-2
Distance 1-3
Angle 2-1-3
Translation-X 1-3
Translation-Y 1-3
Translation-Z 1-3
Rotation-A 1-3
Rotation-B 1-3
Rotation-C 1-3
<class 'geometric.internal.Distance'> : 2
<class 'geometric.internal.Angle'> : 1
<class 'geometric.internal.TranslationX'> : 1
<class 'geometric.internal.TranslationY'> : 1
<class 'geometric.internal.TranslationZ'> : 1
<class 'geometric.internal.RotationA'> : 1
<class 'geometric.internal.RotationB'> : 1
<class 'geometric.internal.RotationC'> : 1
Step    0 : Gradient = 1.686e-01/1.827e-01 (rms/max) Energy =  0.0180063792
Hessian Eigenvalues: 5.00000e-02 5.00000e-02 5.00000e-02 ... 1.60000e-01 1.14610e+00␣
↪1.14610e+00
Step    1 : Displace = 1.073e-01/1.303e-01 (rms/max) Trust = 1.000e-01 (=) Grad = 1.
↪080e-01/1.234e-01 (rms/max) E (change) =  0.0065007416 (-1.151e-02) Quality = 0.413
Hessian Eigenvalues: 5.00000e-02 5.00000e-02 5.00000e-02 ... 1.95811e-01 4.90425e-01␣
↪1.14610e+00
Step    2 : Displace = 8.518e-02/1.017e-01 (rms/max) Trust = 1.000e-01 (=) Grad = 1.
↪592e-02/2.161e-02 (rms/max) E (change) =  0.0001690891 (-6.332e-03) Quality = 0.307
Hessian Eigenvalues: 5.00000e-02 5.00000e-02 5.00000e-02 ... 2.02540e-01 4.27767e-01␣
↪1.14610e+00
Step    3 : Displace = 1.326e-02/1.817e-02 (rms/max) Trust = 1.000e-01 (=) Grad = 1.
↪744e-03/2.373e-03 (rms/max) E (change) =  0.0000051046 (-1.640e-04) Quality = 0.317
Hessian Eigenvalues: 5.00000e-02 5.00000e-02 5.00000e-02 ... 1.68278e-01 4.49086e-01␣
↪1.14610e+00
Step    4 : Displace = 4.295e-03/4.443e-03 (rms/max) Trust = 1.000e-01 (=) Grad = 1.
↪311e-04/1.413e-04 (rms/max) E (change) =  0.0000000164 (-5.088e-06) Quality = 0.287
Hessian Eigenvalues: 5.00000e-02 5.00000e-02 5.00000e-02 ... 1.63098e-01 4.53449e-01␣
↪1.14610e+00
Step    5 : Displace = 2.091e-04/2.541e-04 (rms/max) Trust = 1.000e-01 (=) Grad = 9.
↪256e-06/1.015e-05 (rms/max) E (change) =  0.0000000001 (-1.632e-08) Quality = 0.294
Converged! =D
```

(continues on next page)

(continued from previous page)

```
```

4.4.3 Conclusion

These are some of the capabilities QCEngine offers, check out more documentation. If you like the project, consider starring us on GitHub or if you have any questions, join our Slack channel.

You can run this notebook online in a session or view it on Github.

4.5 QCFractal

Full QCFractal documentation is available.

This tutorial will go over general QCFractal usage to give a feel for the ecosystem. In this tutorial, we employ Snowflake, a simple QCFractal stack which runs on a local machine for demonstration and exploration purposes.

4.5.1 Installation

To begin this quickstart tutorial, first install the QCArchive Snowflake environment from conda:

```
conda env create qcarchive/qcf-snowflake -n snowflake
conda activate snowflake
```

If you have a pre-existing environment with `qcfractal`, ensure that `rdkit` and `geometric` are installed from the `conda-forge` channel and `psi4` from the `psi4` channel. It should be noted that QCFractal does not come with any compute backend by default and they must be installed individually.

4.5.2 Importing QCFractal

First let us import two items from the ecosystem:

- FractalSnowflakeHandler - This is a FractalServer that is temporary and is used for trying out new things.
- `qcfractal.interface` is the QCPortal module, but if using QCFractal it is best to import it locally.

Typically we alias `qcportal` as `ptl`. We will do the same for `qcfractal.interface` so that the code can be used anywhere.

[1]:
```
from qcfractal import FractalSnowflakeHandler
import qcfractal.interface as ptl
```

We can now build a temporary server which acts just like a normal server, but we have a bit more direct control of it.

Warning! All data is lost when this notebook shuts down! This is for demonstration purposes only! For information about how to setup a permanent QCFractal server, see the Setup Quickstart Guide.

[2]:
```
server = FractalSnowflakeHandler()
server
```

[2]:
```
FractalSnowflakeHandler(name='db_eca84388_570c_449a_9b72_44ac0885ea66' uri='https://
↪localhost:55332')
```

We can then build a typical FractalClient to automatically connect to this server using the client() helper command. Note that the server names and addresses are identical in both the server and client.

```
[3]: client = server.client()
     client
```

```
[3]: FractalClient(server_name='db_eca84388_570c_449a_9b72_44ac0885ea66', address='https://
     ↪localhost:55332/', username='None')
```

4.5.3 Adding and Querying data

A server starts with no data, so let's add some! We can do this by adding a water molecule at a poor geometry from XYZ coordinates. Note that all internal QCFractal values are stored and used in atomic units; whereas, the standard Molecule.from_data() assumes an input of Angstroms. We can switch this back to Bohr by adding a `units` command in the text string.

```
[4]: mol = ptl.Molecule.from_data("""
     O 0 0 0
     H 0 0 2
     H 0 2 0
     units bohr
     """)
     mol
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[4]: <Molecule(name='H2O' formula='H2O' hash='58e5adb')>
```

We can then measure various aspects of this molecule to determine its shape. Note that the `measure` command will provide a distance, angle, or dihedral depending if 2, 3, or 4 indices are passed in.

This molecule is quite far from optimal, so let's run an geometry optimization!

```
[5]: print(mol.measure([0, 1]))
     print(mol.measure([1, 0, 2]))
```

```
2.0
90.0
```

4.5.4 Evaluating a Geometry Optimization

We originally installed `psi4` and `geometric`, so we can use these programs to perform a geometry optimization. In QCFractal, we call a geometry optimization a `procedure`, where `procedure` is a generic term for a higher level operation that will run multiple individual quantum chemistry energy, gradient, or Hessian evaluations. Other `procedure` examples are finite-difference computations, n-body computations, and torsiondrives.

We provide a JSON-like input to the client.add_procedure() command to specify the method, basis, and program to be used. The `qc_spec` field is used in all procedures to determine the underlying quantum chemistry method behind the individual procedure. In this way, we can use any program or method that returns a energy or gradient quantity to run our geometry optimization! (See also add_compute().)

```
[6]: spec = {
         "keywords": None,
         "qc_spec": {
```

(continues on next page)

(continued from previous page)

```
        "driver": "gradient",
        "method": "b3lyp-d3",
        "basis": "6-31g",
        "program": "psi4"
    },
}

# Ask the server to compute a new computation
r = client.add_procedure("optimization", "geometric", spec, [mol])
print(r)
print(r.ids)
```

```
ComputeResponse(nsubmitted=1 nexisting=0)
['1']
```

We can see that we submitted a single task to be evaluated and the server has not seen this particular procedure before. The `ids` field returns the unique `id` of the procedure. Different procedures will always have a unique `id`, while identical procedures will always return the same `id`. We can submit the same procedure again to see this effect:

[7]:
```
r2 = client.add_procedure("optimization", "geometric", spec, [mol])
print(r)
print(r.ids)
```

```
ComputeResponse(nsubmitted=1 nexisting=0)
['1']
```

4.5.5 Querying Procedures

Once a task is submitted, it will be placed in the compute queue and evaluated. In this particular case the FractalSnowflakeHandler uses your local hardware to evaluate these jobs. We recommend avoiding large tasks!

In general, the server can handle anywhere between laptop-scale resources to many hundreds of thousands of concurrent cores at many physical locations. The amount of resources to connect is up to you and the amount of compute that you require.

Since we did submit a very small job it is likely complete by now. Let us query this procedure from the server using its `id` like so:

[10]:
```
proc = client.query_procedures(id=r.ids)[0]
proc
```

[10]:
```
<OptimizationRecord(id='1' status='COMPLETE')>
```

This OptimizationRecord object has many different fields attached to it so that all quantities involved in the computation can be explored. For this example, let us pull the final molecule (optimized structure) and inspect the physical dimensions.

Note: if the status does not say `COMPLETE`, these fields will not be available. Try querying the procedure again in a few seconds to see if the task completed in the background.

[11]:
```
final_mol = proc.get_final_molecule()
```

[12]:
```
print(final_mol.measure([0, 1]))
print(final_mol.measure([1, 0, 2]))
final_mol
```

```
1.8441303967690752
108.31440111097281
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[12]: <Molecule(name='H2O' formula='H2O' hash='573ea85')>
```

This water molecule has bond length and angle dimensions much closer to expected values. We can also plot the optimization history to see how each step in the geometry optimization affected the results. Though the chart is not too impressive for this simple molecule, it is hopefully illuminating and is available for any geometry optimization ever completed.

```
[13]: proc.show_history()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

4.5.6 Collections

Submitting individual procedures or single quantum chemistry tasks is not typically done as it becomes hard to track individual tasks. To help resolve this, `Collections` are different ways of organizing standard computations so that many tasks can be referenced in a more human-friendly way. In this particular case, we will be exploring an intermolecular potential dataset.

To begin, we will create a new dataset and add a few intermolecular interactions to it.

```
[14]: ds = ptl.collections.ReactionDataset("My IE Dataset", ds_type="ie", client=client,␣
      ↪default_program="psi4")
```

We can construct a water dimer that has fragments used in the intermolecular computation with the `--` divider. A single water molecule with ghost atoms can be extracted like so:

```
[15]: water_dimer = ptl.Molecule.from_data("""
      O 0.000000 0.000000  0.000000
      H 0.758602 0.000000  0.504284
      H 0.260455 0.000000 -0.872893
      --
      O 3.000000 0.500000  0.000000
      H 3.758602 0.500000  0.504284
      H 3.260455 0.500000 -0.872893
      """)

      water_dimer.get_fragment(0, 1)
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

```
[15]: <Molecule(name='H4O2 ([0],[1])' formula='H4O2' hash='8248da9')>
```

Many molecular entries can be added to this dataset where each is entry is a given intermolecular complex that is given a unique name. In addition, the `add_ie_rxn` method to can automatically fragment molecules.

```
[16]: ds.add_ie_rxn("water dimer", water_dimer)
      ds.add_ie_rxn("helium dimer", """
      He 0 0 -3
      --
      He 0 0 3
      """)
```

```
[16]: <ReactionRecord(ProtoModel)>
```

Once the Collection is created it can be saved to the server so that it can always be retrived at a future date

```
[17]: ds.save()
```

```
[17]: '1'
```

The client can list all Collections currently on the server and retrive collections to be manipulated:

```
[18]: client.list_collections()
```

```
[18]:                                tagline
      collection      name
      ReactionDataset My IE Dataset     None
```

```
[19]: ds = client.get_collection("ReactionDataset", "My IE Dataset")
```

4.5.7 Computing with collections

Computational methods can be applied to all of the reactions in the dataset with just a few simple lines:

```
[20]: ds.compute("B3LYP-D3", "def2-SVP")
```

```
[20]: <ComputeResponse(nsubmitted=10 nexisting=0)>
```

By default this collection evaluates the non-counterpoise corrected interaction energy which typically requires three computations per entry (the complex and each monomer). In this case we compute the B3LYP and -D3 additive correction separately, nominally 12 total computations. However the collection is smart enough to understand that each Helium monomer is identical and does not need to be computed twice, reducing the total number of computations to 10 as shown here. We can continue to compute additional methods. Again, this is being evaluated on your computer! Be careful of the compute requirements.

```
[21]: ds.compute("PBE-D3", "def2-SVP")
```

```
[21]: <ComputeResponse(nsubmitted=10 nexisting=0)>
```

A list of all methods that have been computed for this dataset can also be shown:

```
[22]: ds.list_history()
```

```
[22]:                                           stoichiometry
      driver program method   basis    keywords
      energy psi4    b3lyp    def2-svp None            default
                     b3lyp-d3 def2-svp None            default
                     pbe      def2-svp None            default
                     pbe-d3   def2-svp None            default
```

The above only shows what has been computed and does not pull this data from the server to your computer. To do so, the `get_history` command can be used. As the commands are being executed in the backend we need to wait a bit to get the history again when the computations are complete. The `force=True` flag will re-query the database rather than using cached data.

```
[23]: ds.get_history(force=True)
```

```
[23]:    driver program    method     basis  keywords stoichiometry  \
0  energy    psi4     b3lyp  def2-svp       NaN       default
1  energy    psi4  b3lyp-d3  def2-svp       NaN       default
2  energy    psi4       pbe  def2-svp       NaN       default
3  energy    psi4    pbe-d3  def2-svp       NaN       default

                name
0     B3LYP/def2-svp
1  B3LYP-D3/def2-svp
2       PBE/def2-svp
3    PBE-D3/def2-svp
```

Underlying the Collection is a pandas DataFrame which can show all results:

```
[25]: print(f"DataFrame units: {ds.units}")
ds.df
```

```
DataFrame units: kcal / mol
```

```
[25]:               B3LYP/def2-svp  B3LYP-D3/def2-svp  PBE/def2-svp  PBE-D3/def2-svp
water dimer          -4.751916          -5.577718     -5.115871        -5.632224
helium dimer         -0.000346          -0.000848     -0.000387        -0.000864
```

You can also visualize results and more!

```
[26]: ds.visualize(["B3LYP-D3", "PBE-D3"], "def2-SVP", bench="B3LYP/def2-svp", kind="violin
↪")
```

```
Data type cannot be displayed: application/vnd.plotly.v1+json, text/html
```

4.5.8 Conclusion

These are just some of the capabilities QCFractal offers, check out more documentation. If you like the project, consider starring us on GitHub or if you have any questions, join our Slack channel.

You can run this notebook online in a session or view it on Github.

4.6 QCPortal

Full QCPortal documentation is available.

QCPortal is a data viewer for a QCFractal server. QCPortal can be used with any QCFractal server including one that you build on your own, but for these examples we will use the MolSSI QCArchive server. To begin we first construct a client that points to a QCFractal server instance, here we explicitly add the `molssi.org` server.

```
[1]: import qcportal as ptl
client = ptl.FractalClient("api.qcarchive.molssi.org")
client
```

[1]:
```
FractalClient(server_name='The MolSSI QCArchive Server', address='https://api.
↪qcarchive.molssi.org/', username='None')
```

4.6.1 Finding Collections

One of the main ways to explore the QCArchive is to examine `Collections` which are structures that allow easy manipulation of data in preset ways. Several example of `Collections` contained within the QCArchive are as follows:

- `Dataset` - A dataset where each record is a single molecule and many methods can be applied to each record.
- `ReactionDataset` - A dataset where each record is a combinations of molecule (e.g. interaction and reaction energies) and many methods can be applied to each record.
- `OpenFFWorkflow` - A workflow collection for torsiondrives and contstrained optimization developed with the Open Force Field Iniative.
- `TorsionDriveDataset` - A datas set which organizes many molecular torsion scans together for data exploration, analysis, and methodology comparison (see the TorsionDrive Dataset example for more details).

[2]:
```
client.list_collections().head()
```

[2]:
```
                                          tagline
collection name
Dataset     OpenFF Discrepancy Benchmark 1    None
            OpenFF NCI250K Boron 1            None
            OpenFF Optimization Set 1         None
            OpenFF VEHICLe Set 1              None
            SMIRNOFF Coverage Set 1           None
```

Specific `Collection` types can be queried to limit the amount of collections to browse through:

[4]:
```
client.list_collections("ReactionDataset").head()
```

[4]:
```
                                                                     tagline
collection      name
ReactionDataset A21        Equilibrium complexes from A24 database of sma...
                A24        Interaction energies for small bimolecular com...
                ACONF                     Conformation energies for alkanes
                AlkBind12  Binding energies of saturated and unsaturated ...
                AlkIsod14     Isodesmic reaction energies for alkanes N=3--8
```

4.6.2 Exploring Collections

Collections can be obtained by pulling their data from the ceneral server. A collection is primarily metadata and extremely large collections can be pulled in a few seconds. For this example we will explore S22 dataset which is a small interaction energy dataset of 22 common dimers such as the water dimer, methane dimer, and more. To obtain this collection:

[5]:
```
ds = client.get_collection("ReactionDataset", "S22")
print(ds)
```

```
ReactionDataset(name=`S22`, id='1', client='https://api.qcarchive.molssi.org/')
```

4.6.3 Stastistics and Visualization

Visual statics and plotting can be generated by the `visualize` command:

```
[6]: ds.visualize(method="B2PLYP", basis=["def2-svp", "def2-tzvp"], bench="S220", kind=
     ↪"violin")
```

```
Data type cannot be displayed: application/vnd.plotly.v1+json, text/html
```

4.6.4 Conclusion

These are just some of the capabilities QCPortal offers; check out more examples and the QCPortal documentation. If you like the project, consider starring us on GitHub or if you have any questions, join our Slack channel.