## CONTENTS:

1 Program Execution ............................................. 3

2 Backends .......................................................... 5

3 Configuration Determination .................................. 7

4 Program and Procedure Information .......................... 9

5 Index .................................................................. 11
   5.1 Install QC Engine ............................................ 11
   5.2 Single Compute ............................................... 12
   5.3 Environment Detection .................................... 16
   5.4 Command Line Interface ................................. 17
   5.5 Program Overview .......................................... 19
   5.6 Semiempirical Quantum Mechanics .................. 20
   5.7 Molecular Mechanics ...................................... 22
   5.8 QC Engine API .............................................. 23
   5.9 Changelog .................................................... 36
   5.10 Adding a New Program Harness ...................... 50

Python Module Index .................................................. 53

Index .................................................................. 55
Quantum chemistry program executor and IO standardizer (QCSchema) for quantum chemistry.
A simple example of QC Engine’s capabilities is as follows:

```python
>>> import qcengine as qcng
>>> import qcelemental as qcel

>>> mol = qcel.models.Molecule.from_data('"
>>> O  0.0  0.000  -0.129
>>> H  0.0  -1.494  1.027
>>> H  0.0   1.494  1.027
>>> 
>>> model = qcel.models.AtomicInput(
>>>     molecule=mol,
>>>     driver="energy",
>>>     model= {"method": "SCF", "basis": "sto-3g"},
>>>     keywords={"scf_type": "df"}
>>> )

These input specifications can be executed with the `compute` syntax along with a program specifier:

```python
>>> ret = qcng.compute(model, "psi4")
```

The results contain a complete record of the computation:

```python
>>> ret.return_result
-74.45994963230625

>>> ret.properties.scf_dipole_moment
[0.0, 0.0, 0.6635967188869244]

>>> ret.provenance.cpu
Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
```
Currently available compute backends for single results are as follow:

- **Quantum Chemistry:**
  - adcc
  - Entos
  - Molpro
  - Psi4
  - Terachem
- **Semi-Emperical:**
  - MOPAC
  - xtb
- **AI Potential:**
  - TorchANI
- **Molecular Mechanics:**
  - RDKit
- **Analytical Corrections:**
  - DFTD3

In addition, several procedures are available:

- **Geometry Optimization:**
  - geomeTRIC
  - Pyberny
In addition, QCEngine can automatically determine the following quantities:

- The number of physical cores on the system and to use.
- The amount of physical memory on the system and the amount to use.
- The provenance of a computation (hardware, software versions, and compute resources).
- Location of scratch disk space.
- Location of quantum chemistry programs binaries or Python modules.

Each of these options can be specified by the user as well.

```python
>>> qcng.get_config()
<JobConfig ncores=2 memory=2.506 scratch_directory=None>

>>> qcng.get_config(local_options={"scratch_directory": "/tmp"})
<JobConfig ncores=2 memory=2.506 scratch_directory='/tmp'>

>>> os.environ["SCRATCH"] = "/my_scratch"
>>> qcng.get_config(local_options={"scratch_directory": "$SCRATCH"})
<JobConfig ncores=2 memory=2.506 scratch_directory='/my_scratch'>
```
Available programs and procedures may be printed using the *CLI*:

```bash
>>> qcengine info
>> Version information
QCEngine version: v0.11.0
QCElemental version: v0.11.0

>> Program information
Available programs:
mopac v2016
psi4 v1.3.2
rdkit v2019.03.4

Other supported programs:
cfour dftd3 entos gamess molpro mp2d nwchem terachem torchani
...
5.1 Install QC Engine

You can install qcengine with conda or with pip.

5.1.1 Conda

You can install qcengine using conda:

```bash
>>> conda install qcengine -c conda-forge
```

This installs QC Engine and its dependencies. The qcengine package is maintained on the conda-forge channel.

5.1.2 Pip

You can also install QC Engine using pip:

```bash
>>> pip install qcengine
```

5.1.3 Test the Installation

**Note:** QC Engine is a wrapper for other quantum chemistry codes. The tests for QCEngine will only test the wrapper for a given code if its detected in the $PATH or current Python Environment, otherwise the tests for that package are skipped. Keep this in mind if you see many skip or s codes output from PyTest.

You can test to make sure that Engine is installed correctly by first installing pytest.

From conda:

```bash
>>> conda install pytest -c conda-forge
```

From pip:
>>> pip install pytest

Then, run the following command:

```python
g>> pytest --pyargs qcengine
```

## 5.1.4 Developing from Source

If you are a developer and want to make contributions to Engine, you can access the source code from [GitHub](https://github.com).

### User Interface

- **Single Compute**
- **Environment Detection**
- **Command Line Interface**

## 5.2 Single Compute

QCEngine’s primary purpose is to consume the MolSSI QCSchema and produce QCSchema results for a variety of quantum chemistry, semiempirical, and molecular mechanics programs. Single QCSchema representation comprises of a single energy, gradient, hessian, or properties evaluation.

### 5.2.1 Input Description

An input description has the following fields:

- **molecule** - A QCSchema compliant dictionary or Molecule model.
- **driver** - The energy, gradient, hessian, or properties option.
- **model** - A description of the evaluation model. For quantum chemistry this is typically method and basis. However, non-quantum chemistry models are often a simple method as in method = 'UFF' for forcefield evaluation.
- **keywords** - a dictionary of keywords to pass to the underlying program. These are program-specific keywords.

An example input is as follows:

```python
>>> import qcengine as qcng
>>> import qcelemental as qcel

>>> mol = qcel.models.Molecule.from_data("O 0.0 0.000 -0.129
>>> H 0.0 -1.494 1.027
>>> H 0.0 1.494 1.027
>>> 
>>> inp = qcel.models.AtomicInput(
>>>     molecule=mol,
>>>     driver="energy",
>>>     model=("method": "SCF", "basis": "sto-3g"),
>>>     keywords=("scf_type": "df")
>>> )
```
5.2.2 Computation

A single computation can be evaluated with the `compute` function as follows:

```python
>>> ret = qcng.compute(inp, "psi4")
```

By default the job is given resources relating to the compute environment it is in; however, these variables can be overridden:

```python
>>> ret = qcng.compute(inp, "psi4", local_options={"memory": 2, "ncores": 3})
```

5.2.3 Results

The results contain a complete record of the computation:

```python
>>> ret.return_result
-74.45994963230625

>>> ret.properties.scf_dipole_moment
[0.0, 0.0, 0.6635967188869244]

>>> ret.provenance.cpu
Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
```

5.2.4 Input Fields

```python
```

The MolSSI Quantum Chemistry Schema Parameters

- **id (str, Optional)** – The optional ID for the computation.
- **schema_name (ConstrainedStrValue, Default: qcschema_input)** – The QCSchema specification this model conforms to. Explicitly fixed as qcschema_input.
- **schema_version (int, Default: 1)** – The version number of schema_name to which this model conforms.
- **molecule (Molecule)** – The molecule to use in the computation.
• **driver** (*energy, gradient, hessian, properties*) – Allowed computation driver values.
• **model** (*Model*) – The computational molecular sciences model to run.
• **keywords** (*Dict[Any], Default: {}*) – The program-specific keywords to be used.
• **protocols** (*AtomicResultProtocols, Optional*) – Protocols regarding the manipulation of computational result data.
• **extras** (*Dict[Any], Default: {}*) – Additional information to bundle with the computation. Use for schema development and scratch space.
• **provenance** (*Provenance, Optional*) – Provenance information.

### 5.2.5 Returned Fields

```python
```

Results from a CMS program execution.

**Parameters**

- **id** (*str, Optional*) – The optional ID for the computation.
- **schema_name** (*ConstrainedStrValue, Default: qcschema_output*) – The QCSchema specification this model conforms to. Explicitly fixed as qcschema_output.
- **schema_version** (*int, Default: 1*) – The version number of schema_name to which this model conforms.
- **molecule** (*Molecule*) – The molecule to use in the computation.
- **driver** (*energy, gradient, hessian, properties*) – Allowed computation driver values.
- **model** (*Model*) – The computational molecular sciences model to run.
- **keywords** (*Dict[Any], Default: {}*) – The program-specific keywords to be used.
• **protocols** (*AtomicResultProtocols, Optional*) – Protocols regarding the manipulation of computational result data.

• **extras** (*Dict[Any], Default: {}) – Additional information to bundle with the computation. Use for schema development and scratch space.

• **provenance** (*Provenance*) – Provenance information.

• **properties** (*AtomicResultProperties*) – Named properties of quantum chemistry computations following the MolSSI QCSchema.

  All arrays are stored flat but must be reshappable into the dimensions in attribute shape, with abbreviations as follows:
  – nao: number of atomic orbitals = calcinfo_nbasis
  – nmo: number of molecular orbitals

• **wavefunction** (*WavefunctionProperties, Optional*) – Wavefunction properties resulting from a computation. Matrix quantities are stored in column-major order. Presence and contents configurable by protocol.

• **return_result** (*Union[float, Array, Dict[Any]]*) – The primary return specified by the driver field. Scalar if energy; array if gradient or hessian; dictionary with property keys if properties.

• **stdout** (*str, Optional*) – The primary logging output of the program, whether natively standard output or a file. Presence vs. absence (or null-ness?) configurable by protocol.

• **stderr** (*str, Optional*) – The standard error of the program execution.

• **native_files** (*Dict[Any], Optional*) – DSL files.

• **success** (*bool*) – The success of program execution. If False, other fields may be blank.

• **error** (*ComputeError, Optional*) – Complete description of the error from an unsuccessful program execution.

### 5.2.6 FAQ

1. Where is scratch so I can access the CMS code’s files?

   The QCArchive philosophy is that you shouldn’t go looking in scratch for CMS-code-written files since the scratch directory is deleted automatically by QCEngine and even if preserved may be subject to autodeletion if run from a cluster. Instead, QCEngine brings back the primary input and output and any ancillary files from which it can harvest results. Whether these are returned to the user in *AtomicResult* can be controlled through protocols in the input like `atomicinput.protocols.stdout = True` and eventually (https://github.com/MolSSI/QCElemental/pull/275) `atomicinput.protocols.native_files = "all"`.

   Nevertheless, you can, of course, access the scratch directory and CMS-code-written files. Pass an existing directory to the compute command (this directory will be parent) and tell it to not delete after the run: `qcng.compute(..., local_options={"scratch_directory": "/existing/parent/dir", "scratch_messy": True})`.

2. sdffs
5.3 Environment Detection

QCEngine can inspect the current compute environment to determine the resources available to it.

5.3.1 Node Description

QCEngine can detect node descriptions to obtain general information about the current node.

```python
>>> qcng.config.get_node_descriptor()
<NodeDescriptor hostname_pattern='*' name='default' scratch_directory=None
    memory=5.568 memory_safety_factor=10 ncores=4 jobs_per_node=2>
```

5.3.2 Config

The configuration file operated based on the current node descriptor and can be overridden:

```python
>>> qcng.get_config()
<JobConfig ncores=2 memory=2.506 scratch_directory=None>
```
```python
>>> qcng.get_config(local_options={"scratch_directory": "/tmp"})
<JobConfig ncores=2 memory=2.506 scratch_directory="/tmp">
```
```python
>>> os.environ['SCRATCH'] = "/my_scratch"
```
```python
>>> qcng.get_config(local_options={"scratch_directory": "$SCRATCH"})
<JobConfig ncores=2 memory=2.506 scratch_directory='/my_scratch'>
```

5.3.3 Global Environment

The global environment can also be inspected directly.

```python
>>> qcng.config.get_global()
|
  'hostname': 'qcarchive.molssi.org',
  'memory': 5.568,
  'username': 'user',
  'ncores': 4,
  'cpuinfo': {
    'python_version': '3.6.7.final.0 (64 bit)',
    'brand': 'Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz',
    'hz_advertised': '2.9000 GHz',
    ...
  },
  'cpu_brand': 'Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz'
```
5.3.4 Configuration Files

The computational environment defaults can be overridden by configuration files.

Configuration files must be named `qcengine.yaml` and stored either in the directory from which you run QCEngine, a folder named `.qcarchive` in your home directory, or in a folder specified by the `DQM_CONFIG_PATH` environmental variable. Only one configuration file will be used if multiple are available. The `DQM_CONFIG_PATH` configuration file takes precedence over the current directory, which takes precedence over the `.qcarchive` folder.

The configuration file is a YAML file that contains a dictionary of different node configurations. The keys in the YAML file are human-friendly names for the configurations. The values are dictionaries that define configurations for different nodes, following the `NodeDescriptor` schema:

```python
class qcengine.config.NodeDescriptor(*, hostname_pattern: str, name: str, scratch_directory: str = None, memory: float = None, memory_safety_factor: int = 10, ncores: int = None, jobs_per_node: int = 2, retries: int = 0, is_batch_node: bool = False, mpiexec_command: str = None)
```

Description of an individual node

When running QCEngine, the proper configuration for a node is determined based on the hostname of the node and matching the `hostname_pattern` to each of the configurations defined in `qcengine.yaml`.

An example `qcengine.yaml` file that sets the scratch directory for all nodes is as follows:

```yaml
all:
    hostname_pattern: "*
    scratch_directory: ./scratch
```

5.3.5 Cluster Configuration

A node configuration file is required when using node-parallel tasks on a compute cluster. The configuration file must contain a description of the command used to launch MPI tasks and, in some cases, the designation that a certain node is a compute node. See the descriptions for `mpiexec_command` and `is_batch_node` in the `NodeDescriptor` documentation for further details.

5.4 Command Line Interface

QCEngine provides a command line interface with three commands:

- `qcengine info` displays information about the environment detected by QCEngine.
- `qcengine run` runs a program.
- `qcengine run-procedure` runs a procedure.
5.4.1 Info Command

Command Invocation

```
qcengine info <options>
```

Command Description

This command prints information about the QCityEngine environment.

Arguments

category The information categories to show. Choices include:
- version: Print version of QCityEngine and QCElemental.
- programs: Print detected and supported programs.
- procedures: Print detected and supported procedures.
- config: Print host, compute, and job configuration
- all: Print all available information.

By default, all available information is printed.

5.4.2 Run Command

Command Invocation

```
qcengine run <program> <data>
```

Command Description

This command runs a program on a given task and outputs the result as a JSON blob.

Arguments

program The program to run.
data Data describing the task. One of:
- A JSON blob.
- A file name.
- '-', indicating data will be read from STDIN.
5.4.3 Run-Procedure Command

Command Invocation

```
qcengine run-procedure <program> <data>
```

Command Description

This command runs a procedure on a given task and outputs the result as a JSON blob.

Arguments

- **procedure** The procedure to run.
- **data** Data describing the task. One of:
  - A JSON blob.
  - A file name.
  - `-' , indicating data will be read from STDIN.

Programs

- *Program Overview*
- *Molecular Mechanics*

5.5 Program Overview

The general capabilities available through QCityEngine for each program can be found below:

5.5.1 Quantum Chemistry

<table>
<thead>
<tr>
<th>Program</th>
<th>Production</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>Properties</th>
<th>Wavefunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CFOUR</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qcore (Entos)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GAMESS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MRChem</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Molpro</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>NWChem</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Psi4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Q-Chem</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Terachem</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terachem PBS</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turbomole</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.5.2 Semi-Empirical

<table>
<thead>
<tr>
<th>Program</th>
<th>Production</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>Properties</th>
<th>Wavefunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOPAC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>xtb</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.5.3 AI Potential

<table>
<thead>
<tr>
<th>Program</th>
<th>Production</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>TorchANI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

5.5.4 Molecular Mechanics

<table>
<thead>
<tr>
<th>Program</th>
<th>Production</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RDKit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

5.5.5 Analytical Corrections

<table>
<thead>
<tr>
<th>Program</th>
<th>Production</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTD3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFTD4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gCP</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6 Semiempirical Quantum Mechanics

For semiempirical quantum mechanics (SQM) engines to fit the AtomicInput/Result schema the following convention is used:

- Method: The unique method name (PM7 or GFN2-xTB) including the parametrisation information is provided, no basis is needed

As for quantum mechanical methods a minimal Molecule object is sufficient as input.

**Note**: Semiempirical engines might not handle the concept of ghost atoms correctly, check carefully how the used engine handles ghost atoms. To be sure remove ghost atoms from input to semiempirical engines beforehand.
5.6.1 Example

For example, running a calculation with the GFN2-xTB method using the \texttt{xtb} engine would work like any other QM engine with

```python
>>> import qcelemental as qcel
... mol = qcel.models.Molecule(
... symbols=["O", "H", "H"],
... geometry=[
... [ 0.00000000000000, 0.00000000000000,-0.73578586109551],
... [ 1.44183152868459, 0.00000000000000, 0.36789293054775],
... [-1.44183152868459, 0.00000000000000, 0.36789293054775],
... ],
... )
... model = qcel.models.AtomicInput(
... molecule=mol,
... driver="energy",
... model=\{"method": "GFN2-xTB"\},
... )
... >>> import qcengine as qcng
... >>> ret = qcng.compute(model, "xtb")
... >>> ret.return_result
-5.070451354836705
```

5.6.2 MOPAC

The following semiempirical Hamiltonians are supported with the MOPAC engine.

<table>
<thead>
<tr>
<th>Method</th>
<th>Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>mndo</td>
<td>None</td>
</tr>
<tr>
<td>am1</td>
<td>None</td>
</tr>
<tr>
<td>pm3</td>
<td>None</td>
</tr>
<tr>
<td>rm1</td>
<td>None</td>
</tr>
<tr>
<td>mndod</td>
<td>None</td>
</tr>
<tr>
<td>pm6</td>
<td>None</td>
</tr>
<tr>
<td>pm6-d3</td>
<td>None</td>
</tr>
<tr>
<td>pm6-dh+</td>
<td>None</td>
</tr>
<tr>
<td>pm6-dh2</td>
<td>None</td>
</tr>
<tr>
<td>pm6-dh2x</td>
<td>None</td>
</tr>
<tr>
<td>pm6-d3h4</td>
<td>None</td>
</tr>
<tr>
<td>pm6-3dh4x</td>
<td>None</td>
</tr>
<tr>
<td>pm7</td>
<td>None</td>
</tr>
<tr>
<td>pm7-ts</td>
<td>None</td>
</tr>
</tbody>
</table>
5.6.3 xtb

The following extended tight binding Hamiltonians are available with the xtb engine.

<table>
<thead>
<tr>
<th>Method</th>
<th>Basis</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFN2-xTB</td>
<td>None</td>
<td>10.1021/acs.jctc.8b01176</td>
</tr>
<tr>
<td>GFN1-xTB</td>
<td>None</td>
<td>10.1021/acs.jctc.7b00118</td>
</tr>
<tr>
<td>GFN0-xTB</td>
<td>None</td>
<td>10.26434/chemrxiv.8326202.v1</td>
</tr>
</tbody>
</table>

5.7 Molecular Mechanics

For Molecular Mechanics (MM) engines to fit the AtomicInput/Result schema the following convention is used:

- **Method**: The force field used such as MMFF94, GAFF, OpenFF-1.0.0.
- **Basis**: The typing engine used to find the required parameters.

For all MM computations the input Molecule object must have connectivity and this will not be automatically assigned for you.

5.7.1 Example

```python
>>> mol = qcel.models.Molecule(
    symbols=['O', 'H', 'H'],
    geometry=[[0, 0, 0], [0, 0, 2], [0, 2, 0]],
    connectivity=[[0, 1, 1], [0, 2, 1]],
)

>>> model = qcel.models.AtomicInput(
    molecule=mol,
    driver="energy",
    model={'method': 'openff-1.0.0', 'basis': 'smirnoff'},
)

>>> ret = qcng.compute(model, "openmm")
>>> ret.return_result
0.011185654397410195
```

5.7.2 OpenMM

Currently OpenMM only supports the smirnoff typing engine from the openff-toolkit. Currently available force fields are the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>smirnoff99Frosst-1.1.0</td>
<td>smirnoff</td>
</tr>
<tr>
<td>openff-1.0.0</td>
<td>smirnoff</td>
</tr>
<tr>
<td>openff_unconstrained-1.0.0</td>
<td>smirnoff</td>
</tr>
</tbody>
</table>

Other forcefields may be available depending on your version of the openff-toolkit, see their docs for more information.
5.7.3 RDKit

RDKit force fields currently do not require a typing engine and the basis is omitted in all computations. Currently available force fields are the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>UFF</td>
<td>None</td>
</tr>
<tr>
<td>MMFF94</td>
<td>None</td>
</tr>
<tr>
<td>MMFF94s</td>
<td>None</td>
</tr>
</tbody>
</table>

5.7.4 xtb

Experimental access to force fields are available with the xtb engine. Note that the xtb engine will not require nor use a topology information provided in the input schema.

<table>
<thead>
<tr>
<th>Method</th>
<th>Basis</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFN-FF</td>
<td>None</td>
<td>10.1002/anie.202004239</td>
</tr>
</tbody>
</table>

Developer Documentation

- QCEngine API
- Changelog

5.8 QCEngine API

5.8.1 qcengine Package

Base file for the dqm_compute module.

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute(input_data, program[, raise_error, ...])</td>
<td>Executes a single CMS program given a QCSchema input.</td>
</tr>
<tr>
<td>compute_procedure(input_data, procedure[, ...])</td>
<td>Runs a procedure (a collection of the quantum chemistry executions)</td>
</tr>
<tr>
<td>get_config(*[hostname, local_options])</td>
<td>Returns the configuration key for qcengine.</td>
</tr>
<tr>
<td>get_molecule(name)</td>
<td>Returns a QC JSON representation of a test molecule.</td>
</tr>
<tr>
<td>get_procedure(name)</td>
<td>Returns a procedures executor class</td>
</tr>
<tr>
<td>get_program(name[, check])</td>
<td>Returns a program’s executor class</td>
</tr>
<tr>
<td>list_all_procedures()</td>
<td>List all procedures registered by QCEngine.</td>
</tr>
<tr>
<td>list_all_programs()</td>
<td>List all programs registered by QCEngine.</td>
</tr>
<tr>
<td>list_available_procedures()</td>
<td>List all procedures that can be executed (found) by QCEngine.</td>
</tr>
<tr>
<td>list_available_programs()</td>
<td>List all programs that can be executed (found) by QCEngine.</td>
</tr>
<tr>
<td>register_program(entry_point)</td>
<td>Register a new ProgramHarness with QCEngine.</td>
</tr>
<tr>
<td>unregister_program(name)</td>
<td>Unregisters a given program.</td>
</tr>
</tbody>
</table>
**compute**

qcengine.compute(input_data: Union[Dict[str, Any], qcelemental.models.results.AtomicInput], program: str, raise_error: bool = False, local_options: Optional[Dict[str, Any]] = None, return_dict: bool = False) → qcelemental.models.results.AtomicResult

Executes a single CMS program given a QCSchema input.

The full specification can be found at: http://molssi-qc-schema.readthedocs.io/en/latest/index.html#

**Parameters**

- **input_data** – A QCSchema input specification in dictionary or model from QCElemental.models
- **program** – The CMS program with which to execute the input.
- **raise_error** – Determines if compute should raise an error or not.
- **retries (int, optional)** – The number of random tries to retry for.
- **local_options** – A dictionary of local configuration options
- **return_dict** – Returns a dict instead of qcelemental.models.AtomicResult

**Returns**  A computed AtomicResult object.

**Return type**  result

**compute_procedure**

qcengine.compute_procedure(input_data: Union[Dict[str, Any], BaseModel], procedure: str, raise_error: bool = False, local_options: Optional[Dict[str, str]] = None, return_dict: bool = False) → BaseModel

Runs a procedure (a collection of the quantum chemistry executions)

**Parameters**

- **input_data** (dict or qcelemental.models.OptimizationInput) – A JSON input specific to the procedure executed in dictionary or model from QCElemental.models
- **procedure** ("geometric", "berny") – The name of the procedure to run
- **raise_error** (bool, option) – Determines if compute should raise an error or not.
- **local_options** (dict, optional) – A dictionary of local configuration options
- **return_dict** (bool, optional, default True) – Returns a dict instead of qcelemental.models.AtomicInput

**Returns**  A QC Schema representation of the requested output, type depends on return_dict key.

**Return type**  dict, OptimizationResult, FailedOperation
get_config

qcengine.get_config(*, hostname: Optional[str] = None, local_options: Optional[Dict[str, Any]] = None) → qcengine.config.TaskConfig

Returns the configuration key for qcengine.

get_molecule

qcengine.get_molecule(name)

Returns a QC JSON representation of a test molecule.

get_procedure

qcengine.get_procedure(name: str) → ProcedureHarness

Returns a procedures executor class

get_program

qcengine.get_program(name: str, check: bool = True) → ProgramHarness

Returns a program’s executor class

Parameters check — True Do raise error if program not found. False is handy for the specialized case of calling non-execution methods (like parsing for testing) on the returned Harness.

list_all_procedures

qcengine.list_all_procedures() → Set[str]

List all procedures registered by QCEngine.

list_all_programs

qcengine.list_all_programs() → Set[str]

List all programs registered by QCEngine.

list_available_procedures

qcengine.list_available_procedures() → Set[str]

List all procedures that can be executed (found) by QCEngine.
**list_available_programs**

```
qcengine.list_available_programs() → Set[str]
```

List all programs that can be executed (found) by QCEngine.

**register_program**

```
qcengine.register_program(entry_point: ProgramHarness) → None
```

Register a new ProgramHarness with QCEngine.

**unregister_program**

```
qcengine.unregister_program(name: str) → None
```

Unregisters a given program.

**Classes**

**MDIServer**

```
MDIServer(mdi_options, program, molecule, ...)
```

**class qcengine.MDIServer**

```
class qcengine.MDIServer(mdi_options: str, program: str, molecule, model, keywords, raise_error: bool = False, local_options: Optional[Dict[str, Any]] = None)
```

Bases: `object`

**Methods Summary**

- `recv_coords([coords])` Receive a set of nuclear coordinates through MDI and assign them to the atoms in the current molecule
- `recv_elements([elements])` Receive a set of atomic numbers through MDI and assign them to the atoms in the current molecule
- `recv_masses([masses])` Receive a set of nuclear masses through MDI and assign them to the atoms in the current molecule
- `recv_multiplicity([multiplicity])` Receive the electronic multiplicity through MDI
- `recv_total_charge([charge])` Receive the total system charge through MDI
- `run_energy()` Run an energy calculation
- `send_coords()` Send the nuclear coordinates through MDI
- `send_elements()` Send the atomic number of each nucleus through MDI
- `send_energy()` Send the total energy through MDI
- `send_forces()` Send the nuclear forces through MDI
- `send_masses()` Send the nuclear masses through MDI
- `send_multiplicity()` Send the electronic multiplicity through MDI
- `send_natoms()` Send the number of atoms through MDI
- `send_node()` Send the name of the current node through MDI

continues on next page
send_total_charge()  Send the total system charge through MDI
start()  Receive commands through MDI and respond to them as defined by the MDI Standard
stop()  Stop listening for MDI commands
update_molecule(key, value)  Update the molecule

Methods Documentation

recv_coords (coords: Optional[numpy.ndarray] = None) → None
Receive a set of nuclear coordinates through MDI and assign them to the atoms in the current molecule

Parameters coords (np.ndarray, optional) – New nuclear coordinates. If None, receive through MDI.

recv_elements (elements: Optional[List[int]] = None)
Receive a set of atomic numbers through MDI and assign them to the atoms in the current molecule

Parameters elements (list of int, optional) – New element numbers. If None, receive through MDI.

recv_masses (masses: Optional[List[float]] = None) → None
Receive a set of nuclear masses through MDI and assign them to the atoms in the current molecule

Parameters masses (list of float, optional) – New nuclear masses. If None, receive through MDI.

recv_multiplicity (multiplicity: Optional[int] = None) → None
Receive the electronic multiplicity through MDI

Parameters multiplicity (int, optional) – New multiplicity of the system. If None, receive through MDI.

recv_total_charge (charge: Optional[float] = None) → None
Receive the total system charge through MDI

Parameters charge (float, optional) – New charge of the system. If None, receive through MDI.

run_energy () → None
Run an energy calculation

send_coords () → numpy.ndarray
Send the nuclear coordinates through MDI

Returns coords – Nuclear coordinates

Return type np.ndarray

send_elements ()
Send the atomic number of each nucleus through MDI

Returns elements – Element of each atom

Return type list of int

send_energy () → float
Send the total energy through MDI

Returns energy – Energy of the system

Return type float
\textbf{send\_forces}() → \texttt{numpy.ndarray}

Send the nuclear forces through MDI

\textbf{Returns} forces – Forces on the nuclei

\textbf{Return type} np.ndarray

\textbf{send\_masses}() → \texttt{numpy.ndarray}

Send the nuclear masses through MDI

\textbf{Returns} masses – Atomic masses

\textbf{Return type} np.ndarray

\textbf{send\_multiplicity}() → int

Send the electronic multiplicity through MDI

\textbf{Returns} multiplicity – Multiplicity of the system

\textbf{Return type} int

\textbf{send\_natoms}() → int

Send the number of atoms through MDI

\textbf{Returns} natom – Number of atoms

\textbf{Return type} int

\textbf{send\_node}() → str

Send the name of the current node through MDI

\textbf{Returns} node – Name of the current node

\textbf{Return type} str

\textbf{send\_total\_charge}() → float

Send the total system charge through MDI

\textbf{Returns} charge – Total charge of the system

\textbf{Return type} float

\textbf{start}() → None

Receive commands through MDI and respond to them as defined by the MDI Standard

\textbf{stop}() → None

Stop listening for MDI commands

\textbf{update\_molecule} (key: \texttt{str}, value)

Update the molecule

\textbf{Parameters}

\begin{itemize}
  \item key \texttt{(str)} – Key of the molecular element to update
  \item value – Update value
\end{itemize}
5.8.2 `qcengine.compute` Module

Integrates the computes together

### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute</code></td>
<td>Executes a single CMS program given a QCSchema input.</td>
</tr>
<tr>
<td><code>compute_procedure</code></td>
<td>Runs a procedure (a collection of the quantum chemistry executions)</td>
</tr>
</tbody>
</table>

#### compute

```python
def compute(input_data, program[, raise_error, ...]):
    ...# returns qcelemental.models.results.AtomicResult
```

Executes a single CMS program given a QCSchema input.


**Parameters**

- **input_data** – A QCSchema input specification in dictionary or model from QCElemental.models.
- **program** – The CMS program with which to execute the input.
- **raise_error** – Determines if compute should raise an error or not.
- **retries** *(int, optional)* – The number of random tries to retry for.
- **local_options** – A dictionary of local configuration options
- **return_dict** – Returns a dict instead of qcelemental.models.AtomicResult

**Returns** A computed AtomicResult object.

**Return type** result
compute_procedure

qcengine.compute.compute_procedure(input_data: Union[Dict[str, Any], BaseModel], procedure: str, raise_error: bool = False, local_options: Optional[Dict[str, str]] = None, return_dict: bool = False) → BaseModel

Runs a procedure (a collection of the quantum chemistry executions)

Parameters

- **input_data** (dict or qcelemental.models.OptimizationInput) – A JSON input specific to the procedure executed in dictionary or model from QCEelemental.models
- **procedure** ("geometric", "berny") – The name of the procedure to run
- **raise_error** (bool, option) – Determines if compute should raise an error or not.
- **local_options** (dict, optional) – A dictionary of local configuration options
- **return_dict** (bool, optional, default True) – Returns a dict instead of qcelemental.models.AtomicInput

Returns A QC Schema representation of the requested output, type depends on return_dict key.

Return type dict, OptimizationResult, FailedOperation

5.8.3 qcengine.config Module

Creates globals for the qcengine module

Functions

```python
get_config(*[, hostname, local_options]) Returns the configuration key for qcengine.
get_provenance_augments()

global_repr() A representation of the current global configuration.
```

get_config

qcengine.config.get_config(*, hostname: Optional[str] = None, local_options: Optional[Dict[str, Any]] = None) → qcengine.config.TaskConfig

Returns the configuration key for qcengine.
get_provenance_augments

cqengine.config.get_provenance_augments() → Dict[str, str]

global_repr

cqengine.config.global_repr() → str
    A representation of the current global configuration.

Classes

NodeDescriptor(*, hostname_pattern, name, ...)  Description of an individual node

NodeDescriptor

class cqengine.config.NodeDescriptor(*, hostname_pattern: str, name: str, scratch_directory: str = None, memory: float = None, memory_safety_factor: int = 10, ncores: int = None, jobs_per_node: int = 2, retries: int = 0, is_batch_node: bool = False, mpiexec_command: str = None)

    Bases: pydantic.main.BaseModel
    Description of an individual node

Class Inheritance Diagram
5.8.4 qcengine.util Module

Several import utilities

Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute_wrapper</code></td>
<td>Wraps compute for timing, output capturing, and raise protection</td>
</tr>
<tr>
<td><code>model_wrapper</code></td>
<td>Wrap input data in the given model, or return a controlled error</td>
</tr>
<tr>
<td><code>handle_output_metadata</code></td>
<td>Fuses general metadata and output together.</td>
</tr>
<tr>
<td><code>create_mpi_invocation</code></td>
<td>Create the launch command for an MPI-parallel task</td>
</tr>
<tr>
<td><code>execute</code></td>
<td>Runs a process in the background until complete.</td>
</tr>
</tbody>
</table>

**compute_wrapper**

```python
cqengine.util.compute_wrapper(capture_output: bool = True, raise_error: bool = False) \rightarrow Dict[str, Any]
```

Wraps compute for timing, output capturing, and raise protection

**model_wrapper**

```python
cqengine.util.model_wrapper(input_data: Dict[str, Any], model: BaseModel) \rightarrow BaseModel
```

Wrap input data in the given model, or return a controlled error

**handle_output_metadata**

```python
cqengine.util.handle_output_metadata(output_data: Union[Dict[str, Any], BaseModel], meta-data: Dict[str, Any], raise_error: bool = False, return_dict: bool = True) \rightarrow Union[Dict[str, Any], BaseModel]
```

Fuses general metadata and output together.

- **Returns result** – Output type depends on return_dict or a dict if an error was generated in model construction
- **Return type** `dict` or `pydantic.models.AtomicResult`
create_mpi_invocation

qcengine.util.create_mpi_invocation(exe:

   Parameters

   • executable (str) – Path to executable
   • task_config (TaskConfig) – Specification for number of nodes, cores per node, etc.

execute


Runs a process in the background until complete.

Returns True if exit code <= exit_code (default 0)

Parameters

• command (list of str)
• infiles (Dict[str] = str) – Input file names (names, not full paths) and contents. to be written in scratch dir. May be {}.
• outfiles (List[str] = None) – Output file names to be collected after execution into values. May be {}.
• as_binary (List[str] = None) – Keys of infiles or outfiles to be treated as bytes.
• scratch_name (str, optional) – Passed to temporary_directory
• scratch_directory (str, optional) – Passed to temporary_directory
• scratch_suffix (str, optional) – Passed to temporary_directory
• scratch_messy (bool, optional) – Passed to temporary_directory
• scratch_exist_ok (bool, optional) – Passed to temporary_directory
• blocking_files (list, optional) – Files which should stop execution if present beforehand.
• timeout (int, optional) – Stop the process after n seconds.
• interrupt_after (int, optional) – Interrupt the process (not hard kill) after n seconds.
• environment (dict, optional) – The environment to run in
• shell (bool, optional) – Run command through the shell.
• exit_code (int, optional) – The exit code above which the process is considered failure.

Raises FileExistsError – If any file in blocking is present
Examples

```python
# execute multiple commands in same dir >>> success, dexe = qcng.util.execute(['command_1'], infiles, [],
                    scratch_messy=True) >>> success, dexe = qcng.util.execute(['command_2'], {}, outfiles, scratch_messy=False,
                    scratch_name=Path(dexe['scratch_directory']).name, scratch_exist_ok=True)
```

5.8.5 qcengine.programs Package

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_program(name[, check])</code></td>
<td>Returns a program’s executor class</td>
</tr>
<tr>
<td><code>list_all_programs()</code></td>
<td>List all programs registered by QC Engine.</td>
</tr>
<tr>
<td><code>list_available_programs()</code></td>
<td>List all programs that can be executed (found) by QC Engine.</td>
</tr>
<tr>
<td><code>register_program(entry_point)</code></td>
<td>Register a new ProgramHarness with QC Engine.</td>
</tr>
<tr>
<td><code>unregister_program(name)</code></td>
<td>Unregisters a given program.</td>
</tr>
</tbody>
</table>

get_program

```python
qcengine.programs.get_program(name: str, check: bool = True) → ProgramHarness
```

*Parameters*

- `check` – True: Do raise error if program not found. False is handy for the specialized case of calling non-execution methods (like parsing for testing) on the returned Harness.

list_all_programs

```python
qcengine.programs.list_all_programs() → Set[str]
```

List all programs registered by QC Engine.

list_available_programs

```python
qcengine.programs.list_available_programs() → Set[str]
```

List all programs that can be executed (found) by QC Engine.

register_program

```python
qcengine.programs.register_program(entry_point: ProgramHarness) → None
```

Register a new ProgramHarness with QC Engine.
unregister_program

qcengine.programs.unregister_program(name: str) → None
Unregisters a given program.

Classes

ProgramHarness(*, name, scratch,...)

ProgramHarness


Bases: pydantic.main.BaseModel, abc.ABC

Methods Summary

build_input(input_model, config[, template])
compute(input_data, config)
execute(inputs[, extra_outfiles, ...])
found([raise_error]) Checks if the program can be found.
get_version() Finds program, extracts version, returns normalized version string.
parsen_output(outfiles, input_model)

Methods Documentation


abstract compute (input_data: AtomicInput, config: TaskConfig) → AtomicResult

eexecute (inputs: Dict[str, Any], extra_outfiles: Optional[List[str]] = None, extra_commands: Optional[List[str]] = None, scratch_name: Optional[str] = None, timeout: Optional[int] = None) → Tuple[bool, Dict[str, Any]]

abstract static found (raise_error: bool = False) → bool
Checks if the program can be found.

Parameters raise_error (bool, optional) – If True, raises an error if the program cannot be found.

Returns Returns True if the program was found, False otherwise.

Return type bool
get_version() → str
Finds program, extracts version, returns normalized version string.

Returns Return a valid, safe python version string.

Return type str

parse_output(outfiles: Dict[str, str], input_model: qcelemental.models.results.AtomicInput) → qcelemental.models.results.AtomicResult

Class Inheritance Diagram

5.9 Changelog

5.9.1 v0.21.0 / 2021-11-22

Enhancements

• (GH#321) CFOUR, GAMESS, NWChem, Psi4, DFTD3, MP2D, gCP - learned to return certain native text files under control of the native_files protocol. GAMESS users are strongly advised to at least set protocols.native_files = "input" so that the job is reproducible. @loriab

• (GH#325) Torsiondrive - learned to use multiple molecules as input to torsiondrives. @jthorton

• (GH#327) TorchANI - learned to use GPUs if available. @kexul

• (GH#330, GH#332) NWChem - learned to restart from existing scratch if QCEngine is killed. @WardLT

5.9.2 v0.20.1 / 2021-10-08

Bug Fixes

• (GH#322) Psi4 - allowed more test cases with gradients and Hessians after a compatibility PR started saving them. @loriab

• (GH#323) Turbomole - learned to store calcinfo_natom so that gradients and Hessians can be computed after QCElemental started using that quantity for shape checking in [v0.22.0](https://github.com/MolSSI/QCElemental/blob/master/docs/source/changelog.rst#0220–2021-08-26) @eljost
5.9.3 v0.20.0 / 2021-10-01

New Features

- (GH#305) TorsionDrive - new procedure to automate constrained optimizations along a geometry grid. Akin to the longstanding QCFractal TorsionDrive service. @SimonBoothroyd

Enhancements

- (GH#307) NWChem - learns to automatically increase the number of iterations when SCF, CC, etc. fails to converge. @WardLT
- (GH#309) qcengine info learned to print the location of found CMS programs, and geometric, OpenMM, and RDKit learned to return their versions. @loriab
- (GH#311) CFOUR, GAMESS, NWChem harnesses learned to notice which internal module performs a calc (e.g., tce/cc for NWChem) and to store it in AtomicResult.provenance.module. Psi4 already does this. @loriab
- (GH#312) CFOUR, GAMESS, NWChem harnesses learned to run and harvest several new methods in the MP, CC, Cl, DFT families. @loriab
- (GH#316) Config - TaskConfig learned a new field scratch_messy to instruct a qcng.compute() run to not clean up the scratch directory at the end. @loriab
- (GH#316) GAMESS - harness learned to obey ncores and scratch_messy local_config options. When ncores > 1, the memory option is partitioned into replicated and non after exetyp=check trials. @loriab
- (GH#316) Psi4 - harness learned to obey scratch_messy and memory local_config options. Memory was previously off by a little (GB vs GiB). @loriab
- (GH#316) CFOUR - harness learned to obey scratch_messy and memory local_config options. Memory was previously off by a little. @loriab
- (GH#316) NWChem - harness learned to obey scratch_messy and memory local_config options. Memory was previously very off for v7. @loriab
- (GH#315) CFOUR, GAMESS, NWChem – learned to return in AtomicInput or program native orientation depending on fix_com & fix_orientation= T or F. Psi4 already did this. Previously these three always returned AtomicInput orientation. Note that when returning program native orientation, the molecule is overwritten, so AtomicResult is not a superset of AtomicInput. @loriab
- (GH#315) CFOUR, GAMESS, NWChem – learned to harvest gradients and Hessians. @loriab
- (GH#317) Docs - start “new harness” docs, so contributors have a coarse roadmap. @loriab
- (GH#320) CFOUR, NWChem – learned to run with ghost atoms, tentatively. @loriab
Bug Fixes

- (GH#313, GH#319) OpenMM - accommodate both old and new simtk/openmm import patterns. @dotsdl

5.9.4 v0.19.0 / 2021-05-16

New Features

- (GH#290) MCTC-GCP - harness for new implementation of gCP, mctc-gcp, whose cmdline interface is drop-in replacement. @loriab
- (GH#291) DFTD4 - new harness for standalone DFT-D4 executable. @awvwgk
- (GH#289) TeraChem - new harness for TeraChem Protocol Buffer Server mode. @coltonbh

Enhancements

- (GH#288) GAMESS, Cfour, NWChem - add calcinfo harvesting, HF and MP2 gradient harvesting. @loriab

Bug Fixes

- (GH#288) Avert running model.basis = BasisSet schema even though they validate. @loriab
- (GH#294) NWChem - fixed bug where was retrieving only the first step in a geometry relaxation with line-search off. @WardLT
- (GH#297) MDI - Update interface for v1.2. @loriab

5.9.5 v0.18.0 / 2021-02-16

New Features

- (GH#206) OptKing - new procedure harness for OptKing optimizer. @AlexHeide
- (GH#269) MRChem - new multiresolution chemistry program harness. @robertodr
- (GH#277) ADCC - new program harness for ADC-connect. (Requires Psi4 for SCF.) @maxscheurer
- (GH#278) gCP - new program harness for geometric counterpoise. @hokru
- (GH#280) Add framework to register identifying known outfile errors, modify input schema, and rerun. @WardLT
- (GH#281) NWChem - new procedure harness to use NWChem’s DRIVER geometry optimizer with NWChem’s program harness gradients. @WardLT
- (GH#282) DFTD3 - added D3m and D3m(bj) parameters for SAPT0/HF. Allow pairwise analysis to be returned. @jeffschriber
Enhancements

- (GH#274) Entos/Qcore - renamed harness and updated to new Python bindings. @dgasmith
- (GH#283) OpenMM - transition harness from openforcefield packages on omnia channel to openff.toolkit packages on conda-forge channel. @SimonBoothroyd
- (GH#286, GH#287) CI - moves from Travis-CI to GHA for open-source testing. @loriab

Bug Fixes

- (GH#273) TeraChem - fixed bug of missing method field. @stvogt

5.9.6 v0.17.0 / 2020-10-02

New Features

- (GH#262) Add project authors information. @loriab

Enhancements

- (GH#264) Turbomole - add analytic and finite difference Hessians. @eljost
- (GH#266) Psi4- error messages from Psi4Harness no longer swallowed by KeyError. @dotsdl

Bug Fixes

- (GH#264) Turbomole - fix output properties handling. @eljost
- (GH#265) xtb - ensure extra tags are preserved in XTB harness. @WardLT
- (GH#270) TorchANI - now lazily loads models as requested for compute. @dotsdl

5.9.7 v0.16.0 / 2020-08-19

New Features

Enhancements

- (GH#241) NWChem - improved performance by turning on atoms_map=True, which does seem to be true. @WardLT
- (GH#257) TorchANI - learned the ANI2x model and to work with v2. @farhadrgh
- (GH#259) Added MP2.5 & MP3 energies and HF, MP2.5, MP3, LCCD gradients reference data to stdsuite. @loriab
- (GH#261) Q-Chem - learned to return more informative Provenance, learned to work with v5.1. @loriab
- (GH#263) NWChem - learned how to turn off automatic Z-Matrix coordinates with geometry__noautoz = True. @WardLT
Bug Fixes

• (GH#261) Molpro - learned to error cleanly if version too old for XML parsing. @loriab
• (GH#261) Q-Chem - learned to extract version from output file instead of `qchem -h` since command isn’t available from a source install. @loriab

5.9.8 v0.15.0 / 2020-06-26

New Features

• (GH#232) PyBerny - new geometry optimizer procedure harness. @jhrmmn
• (GH#238) Set up testing infrastructure, “stdsuite”, where method reference values and expected results names (e.g., total energy and correlation energy from MP2) are stored here in QCEngine but may be used from anywhere (presently, Psi4). Earlier MP2 and CCSD tests here converted to new scheme, removing `test_standard_suite_mp2.py` and `ccsd`.
• (GH#249, GH#254) XTB - new harness for xtb-python that natively speaks QCSchema. @awvwgk

Enhancements

• (GH#230) NWChem - improved dipole, HOMO, LUMO harvesting.
• (GH#233) `qcng.util.execute` learned argument `exit_code` above which to fail, rather than just `!= 0`.
• (GH#234) MDI - harness updated to support release version v1.0.0.
• (GH#238) Cfour, GAMESS, NWChem – harnesses updated to collect available spin components for MP2 and CCSD. Also updated to set appropriate `qcel.models.AtomicProperties` from collected QCVariables.
• (GH#239) OpenMM - OpenMM harness now looks for cmiles information in the molecule extras field when typing. Also we allow for the use of gaff forcefields. @jthorton
• (GH#243) NWChem - more useful stdout error return.
• (GH#244) Added CCSD(T), LCCD, and LCCSD reference data to stdsuite. @loriab
• (GH#246) TorchANI - harness does not support v2 releases.
• (GH#251) DFTD3 - added D3(0) and D3(BJ) parameters for PBE0-DH functional.

Bug Fixes

• (GH#244) Psi4 - fixed bug in `extras["psiapi"] == True` mode where if calc failed, error not handled by QCEngine. @loriab
• (GH#245) Added missing import to sys for `test_standard_suite.py`. @sjrl
• (GH#248) NWChem - fix HFexch specification bug.
• (GH#253) Make compatible with both py-cpuinfo 5 & 6, fixing issue 252.
5.9.9 v0.14.0 / 2020-02-06

New Features

- (GH#212) NWChem - Adds CI for the NWChem harness.
- (GH#226) OpenMM - Moves the OpenMM harness to a canonical forcefield based method/basis language combination.
- (GH#228) RDKit - Adds MMFF94 force field capabilities.

Enhancements

- (GH#201) Psi4 - psi4 --version collection to only grab the last line.
- (GH#202) Entos - Adds wavefunction parsing.
- (GH#203) NWChem - Parses DFT empirical dispersion energy.
- (GH#204) NWChem - Allows custom DFT functionals to be run.
- (GH#205) NWChem - Improved gradient output and added Hessian support for NWChem.
- (GH#215) Psi4 - if Psi4 location can be found by either PATH or PYTHONPATH, harness sets up both subprocesses and API execution.
- (GH#215) get_program shows the helpful “install this” messages from found() rather than just saying “cannot be found”.

Bug Fixes

- (GH#199) Fix typo breaking NWChem property parsing.
- (GH#215) NWChem complains before a calculation if the necessary networkx package not available.
- (GH#207) NWChem - Minor bug fixes for NWChem when more than core per MPI rank is used.
- (GH#209) NWChem - Fixed missing extras tags in NWChem harness.

5.9.10 v0.13.0 / 2019-12-10

New Features

- (GH#151) Adds a OpenMM Harness for evaluation of SMIRNOFF force fields.
- (GH#189) General MPI support and MPI CLI generator.
Enhancements

- (GH#175) Allows specifications for `nnodes` to begin MPI support.
- (GH#177) NWChem - Parsing updates including Hessian abilities.
- (GH#180) GAMESS - Output properties improvements.
- (GH#181) NWChem - Output properties improvements.
- (GH#183) Entos - Hessian and XTB support.
- (GH#185) Entos - Improved subcommand support.
- (GH#187) QChem - Support for raw log files without the binary file requirements and improved output properties support.
- (GH#188) Automatic buffer reads to prevent deadlocking of process for very large outputs.
- (GH#194) DFTD3 - Improved error message on failed evaluations.
- (GH#195) Blackens the code base add GHA-based lint checks.

Bug Fixes

- (GH#179) QChem - fixes print issue when driver is of an incorrect value.
- (GH#190) Psi4 - fixes issues for methods without basis sets such as HF-3c.

5.9.11 v0.12.0 / 2019-11-13

New Features

- (GH#159) Adds MolSSI Driver Interface support.
- (GH#160) Adds Turbomole support.
- (GH#164) Adds Q-Chem support.

Enhancements

- (GH#155) Support for Psi4 Wavefunctions using v1.4a2 or greater.
- (GH#162) Adds test for geometry optimization with trajectory protocol truncation.
- (GH#167) CFOUR and NWChem parsing improvements for CCSD(T) properties.
- (GH#168) Standardizes on `dispatch.out` for the common output files.
- (GH#170) Increases coverage and begins a common documentation page.
- (GH#171) Add Molpro to the standard suite.
Bug Fixes

5.9.12 v0.11.0 / 2019-10-01

New Features

• (GH#162) Adds a test to take advantage of Elemental’s Protocols. Although this PR does not technically change anything in Engine, bumping the minor version here allows upstream programs to note when this feature was available because the minimum version dependency on Elemental has been bumped as well.

Enhancements

• (GH#143) Updates to Entos and Molpro to allow Entos to execute functions from the Molpro Harness. Also helps the two drivers to conform to GH#86.
• (GH#145, GH#148) Initial CLI tests have been added to help further ensure Engine is running proper.
• (GH#149) The GAMESS Harness has been improved by adding testing.
• (GH#150, GH#153) TorchANI has been improved by adding a Hessian driver to it and additional information is returned in the extra field when energy is the driver. This also bumped the minimum version of TorchANI Engine supports from 0.5 to 0.9.
• (GH#154) Molpro’s harness has been improved to support callinfo_X properties, unrestricted HF and DFT calculations, and the initial support for parsing local correlation calculations.
• (GH#158) Entos’ output parsing has been improved to read the json dictionary produced by the program directly. Also updates the input file generation.
• (GH#161) Updates MOPAC to have more sensible quantum-chemistry like keywords by default.

Bug Fixes

• (GH#156) Fixed a compatibility bug in specific version of Intel-OpenMP by skipping version 2019.5-281.
• (GH#161) Improved error handling in MOPAC if the execution was incorrect.

5.9.13 v0.10.0 / 2019-08-25

New Features

• (GH#132) Expands CLI for info, run, and run-procedure options.
• (GH#137) A new CI pipeline through Azure has been developed which uses custom, private Docker images to house non-public code which will enable us to test Engine through integrated CI on these codes securely.
• (GH#140) GAMESS, CFOUR, NWChem preliminary implementations.
Enhancements

- (GH#138) Documentation on Azure triggers.
- (GH#139) Overhauls install documentation and clearly defines dev install vs production installs.

5.9.14 v0.9.0 / 2019-08-14

New Features

- (GH#120) Engine now takes advantage of Elemental’s new Msgpack serialization option for Models. Serialization defaults to msgpack when available (conda install msgpack-python [-c conda-forge]), falling back to JSON otherwise. This results in substantial speedups for both serialization and deserialization actions and should be a transparent replacement for users within Engine and Elemental themselves.

Enhancements

- (GH#112) The MolproHarness has been updated to handle DFT and CCSD(T) energies and gradients.
- (GH#116) An environment context manager has been added to catch NumPy style parallelization with Python functions.
- (GH#117) MOPAC and DFTD3 can now accept an extras field which can pass around additional data, conforming to the rest of the Harnesses.
- (GH#119) Small visual improvements to the docs have been made.
- (GH#120) Lists inside models are now generally converted to numpy arrays for internal storage to maximize the benefit of the new Msgpack feature from Elemental.
- (GH#133) The GAMESS Harness now collects the CCSD as part of its output.

Bug Fixes

- (GH#127) Removed unused imports from the NWChem Harvester module.
- (GH#129) Missing type hints from the MolproHarness have been added.
- (GH#131) A code formatting redundancy in the GAMESS input file parser has been removed.

5.9.15 v0.8.2 / 2019-07-25

Bug Fixes

- (GH#114) Make compute and compute_procedure not have required kwargs while debugging a Fractal serialization issue. This is intended to be a temporary change and likely reverted in a later release.
5.9.16 v0.8.1 / 2019-07-22

Enhancements

- (GH#110) Psi4’s auto-retry exception handlers now catch more classes of random errors

Bug Fixes

- (GH#109) Geometric auto-retry settings now correctly propagate through the base code.

5.9.17 v0.8.0 / 2019-07-19

New Features

- (GH#95, GH#96, GH#97, and GH#98) The NWChem interface from QCDB has been added. Thanks to @vi-vacebelles and @jygrace for this addition!
- (GH#100) The MOPAC interface has now been added to QCEngine thanks help to from @godotalgorithm.

Enhancements

- (GH#94) The gradient and molecule parsed from a GAMESS calculation output file are now returned in `parse_output`
- (GH#101) Enabled extra files in TeraChem scratch folder to be requested by users, collected after program execution, and recorded in the `Result` object as extras.
- (GH#103) Random errors can now be retried a finite, controllable number of times (current default is zero retries). Geometry optimizations automatically set retries to 2. This only impacts errors which are categorized as `RandomError` by QCEngine and all other errors are raised as normal.

Bug Fixes

- (GH#99) QCEngine now manages an explicit folder for each Psi4 job to write into and passes the scratch directory via `-s` command line. This resolves a key mismatch which could cause an error.
- (GH#102) DFTD3 errors are now correctly returned as a `FailedOperation` instead of a raw dict.

5.9.18 v0.7.1 / 2019-06-18

Bug Fixes

- (GH#92) Added an `__init__.py` file to the `programs/tests` directory so they are correctly bundled with the package.
5.9.19 v0.7.0 / 2019-06-17

Breaking Changes

- (GH#85) The resource file programs.dftd3.dashparam.py has relocated and renamed to programs.empirical Dispersion_resources.py.
- (GH#89) Function util.execute forgot str argument scratch_location and learned scratch_directory in the same role of existing directory within which temporary directories are created and cleaned up. Non-user-facing function util.scratch_directory renamed to util.temporary_directory.

New Features

- (GH#60) WIP: QCEngine interface to GAMESS can run the program (after light editing of rungms) and parse selected output (HF, CC, FCI) into QCschema.
- (GH#73) WIP: QCEngine interface to CFOUR can run the program and parse a variety of output into QC-Schema.
- (GH#59, GH#71, GH#75, GH#76, GH#78, GH#88) Molpro improvements: Molpro can be run by QCEngine; and the input generator and output parser now supports CCSD energy and gradient calculations. Large thanks to @sjrl for many of the improvements.
- (GH#69) Custom Exceptions have been added to QCEngine’s returns which will make parsing and diagnosing them easier and more programmatic for codes which invoke QCEngine. Thanks to @dgasmith for implementation.
- (GH#82) QCEngine interface to entos can create input files (dft energy and gradients), run the program, and parse the output.
- (GH#85) MP2D interface switched to upstream repo (https://github.com/Chandemonium/MP2D v1.1) and now produces correct analytic gradients.

Enhancements

- (GH#62, GH#67, GH#83) A large block of TeraChem improvements thanks to @ffangliu contributions. Changed the input parser to call qcelemental to_string method with bohr unit, improved output of parser to turn stdout into Result, and modified how version is parsed.
- (GH#63) QCEngine functions util.which, util.which_version, util.parse_version, and util.safe_version removed after migrating to QCElemental.
- (GH#65) Torchani can now handle the ANI1-x and ANI1-ccx models. Credit to @dgasmith for implementation.
- (GH#74) Removes caching and reduces pytorch overhead from Travis CI. Credit to @dgasmith for implementation.
- (GH#77) Rename ProgramExecutor to ProgramHarness and BaseProcedure to ProcedureHarness.
- (GH#77) Function util.execute(...) learned to collect output files matching a globbed filename.
- (GH#81) Function util.execute learned list argument as_binary to handle input or output files as binary rather than string.
- (GH#81) Function util.execute learned bool argument scratch_exist_ok to run in a preexisting directory. This is handy for stringing together execute calls.
• (GH#84) Function `util.execute` learned str argument `scratch_suffix` to identify temp dictionaries for debugging.

• (GH#90) DFTD3 now supports preliminary parameters for zero and Becke-Johnson damping to use with SAPT0-D

Bug Fixes

• (GH#80) Fix “psi4:qcvars” handling for older Psi4 versions.

5.9.20 v0.6.4 / 2019-03-21

Bug Fixes

• (GH#54) Psi4’s Engine implementation now checks its key words in a case insensitive way to give the same value whether you called Psi4 or Engine to do the compute.

• (GH#55) Fixed an error handling routine in Engine to match Psi4.

• (GH#56) Complex inputs are now handled better through Psi4’s wrapper which caused Engine to hang while trying to write to `stdout`.

5.9.21 v0.6.3 / 2019-03-15

New Features

• (GH#28) TeraChem is now a registered executor in Engine! Thanks to @ffangliu for implementing.

• (GH#46) MP2D is now a registered executor in Engine! Thanks to @loriab for implementing.

Enhancements

• (GH#46) `dftd3`’s workings received an overhaul. The `mol` keyword has been replaced with `dtype=2`, full Psi4 support is now provided, and an MP2D interface has been added.

Bug Fixes

• (GH#50 and GH#51) Executing Psi4 on a single node with multiprocessing is more stable because Psi4 temps are moved to scratch directories. This behavior is now better documented with an example as well.

• (GH#52) Psi4 calls are now executed through the `subprocess` module to prevent possible multiprocessing issues and memory leak after thousands of runs. A trade off is this adds about 0.5 seconds to task start-up, but its safe. A future Psi4 release will correct this issue and the change can be reverted.
5.9.22 v0.6.2 / 2019-03-07

Enhancements

- (GH#38 and GH#39) Documentation now pulls from the custom QC Archive Sphinx Theme, but can fall back to the standard RTD theme. This allows all docs across QCA to appear consistent with each other.
- (GH#43) Added a base model for all `Procedure` objects to derive from. This allows procedures’ interactions with compute programs to be more unified. This PR also ensured GeomeTRIC provides Provenance information.

Bug Fixes

- (GH#40) This PR improved numerous back-end and testing quality of life aspects. Fixed `setup.py` to call `pytest` instead of `unittest` when running tests on install. Some conda packages for Travis-CI are cached to reduce the download time of the larger computation codes. Psi4 is now pinned to the 1.3 version to fix build-level pin of libint. Conda-build recipe removed to avoid possible confusion for everyone who isn’t a Conda-Forge recipe maintainer. Tests now rely exclusively on the `conda env` setups.

5.9.23 v0.6.1 / 2019-02-20

Bug Fixes

- (GH#37) Fixed an issue where RDKit methods were not case agnostic.

5.9.24 v0.6.0 / 2019-02-28

Breaking Changes

- (GH#36) breaking change Model objects are returned by default rather than a dictionary.

New Features

- (GH#18) Add the `dftd3` program to available computers.
- (GH#29) Adds preliminary support for the `Molpro` compute engine.
- (GH#31) Moves all computation to `ProgramExecutor` to allow for a more flexible input generation, execution, output parsing interface.
- (GH#32) Adds a general `execute` process which safely runs subprocess jobs.
Enhancements

- (GH#33) Moves the dftd3 executor to the new ProgramExecutor interface.
- (GH#34) Updates models to the more strict QCElemental v0.3.0 model classes.
- (GH#35) Updates CI to avoid pulling CUDA libraries for torchani.
- (GH#36) First pass at documentation.

5.9.25 v0.5.2 / 2019-02-13

Enhancements

- (GH#24) Improves load times dramatically by delaying imports and cpuutils.
- (GH#30) Ensures Psi4 output is already returned and Pydantic v0.20+ changes.

5.9.26 v0.5.1 / 2019-01-29

Enhancements

- (GH#22) Compute results are now returned as a dict of Python Primals which have been serialized-deserialized through Pydantic instead of returning un-processed Python objects or json-compatible string.

5.9.27 v0.5.0 / 2019-01-28

New Features

- (GH#8) Adds the TorchANI program for ANI-1 like energies and potentials.
- (GH#16) Adds QCElemental models based off QCSchema to QCEngine for both validation and object-based manipulation of input and output data.

Enhancements

- (GH#14) Migrates option to Pydantic objects for validation and creation.
- (GH#14) Introduces NodeDescriptor (for individual node description) and JobConfig (individual job configuration) objects.
- (GH#17) NodeDescriptor overhauled to work better with Parsl/Balsam/Dask/etc.
5.10 Adding a New Program Harness

Program harnesses are for community CMS codes that can independently (or with a SCF bootstrap) compute single-point energies, derivatives, or properties or components thereof (e.g., dispersion corrections).

A single CMS code generally has one program harness. However, if there are drastically different ways of running a code (e.g., TeraChem text input file and TeraChem PBS), separate harnesses may be created. Also, if there are specialty capabilities not fitting into “single-point energies . . . ” (e.g., GAMESS makefp task), an additional procedure harness may be created.

This guide is a coarse path through adding a new program harness.

1. Open up communication with the QCEngine maintainers. Post an issue to GitHub and join the Slack channel (link off GH README) so you can get advice.

2. Copy a similar harness. Choose a program that your code behaves roughly like (mostly consider parsed vs. API access) and copy that harness, renaming it as your own and commenting out all but the structure. Search for the (copied) harness name to register it.

3. Fill in the defaults section with program name and characteristics.

4. Fill in the def found function using which and which_import from QCElemental. See NWChem and OpenMM for examples of handling additional dependencies.

5. If your code’s version can be extracted short of parsing an output file, fill in def get_version next. After this, > qengine info should show your code (provided it’s in path).

6. To get a string output of QCSchema Molecule in your code’s format, you may need to add a dtype at qcelemental/molparse/to_string.py.

7. If your code’s of the common translate-QCSchema-to-input, run, translate-output-to-QCSchema variety, next work on the def execute function. This is fairly simple because it calls the powerful qcengine.util.execute to handle scratch, timeout, file writing and collection, etc. The harness function needs the names of input files (hard-code a string for now), the execution command, and the names of any scratch files to return for processing. Once ready, fill in def get_version if not done above.

8. Now fill in the short def compute entirely and def build_input and def parse_output skeletally. Set up a simple molecule-and-model AtomicInput dictionary and run it with qcng. compute(atomicinput, "yourcode") to get something to iterate on.

9. Fill in def build_input to form your code’s usual input format from the fields of AtomicInput.

10. Fill in def parse_output to take results and put them into AtomicResult. Most important is the return_result field. AtomicResultProperties can be populated when convenient. WavefunctionProperties is great but save for a later pass.

11. At this point your harness can correctly run one or more QCSchema inputs of your devising. Time to put it through paces. Register your code in _programs in testing.py. Most tests will then need a @using("yourcode") decorator so that they don’t run (and fail the test suite) when your code isn’t available.

12. Add basic tests to qcengine/tests/test_harness_canonical.py * def test_compute_energy(program, model, keywords): * def test_compute_gradient(program, model, keywords): * def test_compute_energy_qcsk_basis(program, model, keywords):

13. Add basic failure tests to qcengine/tests/test_harness_canonical.py * def test_compute_bad_models(program, model):

14. Add tests for the runtime config and to qcengine/programs/tests/test_canonical_config.py

15. For QM codes, consider adding lines to the qcengine/programs/tests/test_standard_suite.py to check energies, gradients, and Hessians against other codes.
1. For codes that can produce a Hartree–Fock, add lines to the qcengine/program/tests/test_alignment.py to check molecular and properties orientation handling.

1. If your code is available as a visible binary (e.g., pip, conda, docker download with no or trivial build), create a testing lane by adding to devtools/conda-envs/ and .github/workflows/CI.yml. This will check your code for every PR. We’re looking into private testing for codes that aren’t available.

1. Throughout, talk with the maintainers with questions. Error handling, especially, is intricate.
q
qcengine, 23
qcengine.compute, 29
qcengine.config, 30
qcengine.programs, 34
qcengine.util, 32
INDEX

A
AtomicInput (class in qcelemental.models), 13
AtomicResult (class in qcelemental.models), 14

B
build_input () (qcengine.programs.ProgramHarness method), 35

C
compute () (in module qcengine), 24
compute () (in module qcengine.compute), 29
compute () (qcengine.programs.ProgramHarness method), 35
compute_procedure () (in module qcengine), 24
compute_procedure () (in module qcengine.compute), 30
compute_wrapper () (in module qcengine.util), 32
create_mpi_invocation () (in module qcengine.util), 33

E
execute () (in module qcengine.util), 33
execute () (qcengine.programs.ProgramHarness method), 35

F
found () (qcengine.programs.ProgramHarness static method), 35

G
getับ Config () (in module qcengine), 25
get_config () (in module qcengine.config), 30
get_molecule () (in module qcengine), 25
get_procedure () (in module qcengine), 25
get_program () (in module qcengine), 25
get_program () (in module qcengine.programs), 34
get_provenance_augments () (in module qcengine.config), 31
get_version () (qcengine.programs.ProgramHarness method), 35
global_repr () (in module qcengine.config), 31

H
handle_output_metadata () (in module qcengine.util), 32

L
list_all_procedures () (in module qcengine), 25
list_all_programs () (in module qcengine), 25
list_all_programs () (in module qcengine.programs), 34
list_available_procedures () (in module qcengine), 25
list_available_programs () (in module qcengine), 26
list_available_programs () (in module qcengine.programs), 34

M
MDIServer (class in qcengine), 26
model_wrapper () (in module qcengine.util), 32
module
cqengine, 23
cqengine.compute, 29
cqengine.config, 30
cqengine.programs, 34
cqengine.util, 32

N
NodeDescriptor (class in qcengine.config), 17, 31

P
parse_output () (qcengine.programs.ProgramHarness method), 36
ProgramHarness (class in qcengine.programs), 35

Q
qcengine
module, 23
cqengine.compute
module, 29
cqengine.config
module, 30
qcengine.programs
    module, 34
qcengine.util
    module, 32

R
recv_coords() (qcengine.MDIServer method), 27
crecv_elements() (qcengine.MDIServer method), 27
crecv_masses() (qcengine.MDIServer method), 27
crecv_multiplicity() (qcengine.MDIServer method), 27
crecv_total_charge() (qcengine.MDIServer method), 27

register_program() (in module qcengine), 26
register_program() (in module qcengine.programs), 34
crun_energy() (qcengine.MDIServer method), 27

S
send_coords() (qcengine.MDIServer method), 27
send_elements() (qcengine.MDIServer method), 27
send_energy() (qcengine.MDIServer method), 27
send_forces() (qcengine.MDIServer method), 27
send_masses() (qcengine.MDIServer method), 28
send_multiplicity() (qcengine.MDIServer method), 28
send_natoms() (qcengine.MDIServer method), 28
send_node() (qcengine.MDIServer method), 28
send_total_charge() (qcengine.MDIServer method), 28

start() (qcengine.MDIServer method), 28
stop() (qcengine.MDIServer method), 28

U
unregister_program() (in module qcengine), 26
unregister_program() (in module qcengine.programs), 35
update_molecule() (qcengine.MDIServer method), 28