1 Program Execution 3
2 Backends 5
3 Configuration Determination 7
4 Index 9
  4.1 Installing QC Engine 9
  4.2 Community 10
  4.3 Single Compute 11
  4.4 Environment Detection 12
  4.5 QC Engine API 13
  4.6 Changelog 20
  4.7 Development Guidelines 24

Python Module Index 25

Index 27
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
"")

>>> input = qcel.models.ResultInput(
    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(input, "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:***
  - Terachem
  - Molpro
  - Psi4

- **AI Evaluation:**
  - TorchANI

- **Molecular Mechanics:**
  - RDKit

In addition, several procedures are available:

- **Geometry Optimization:**
  - geomeTRIC
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.

```bash
>>> 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'>
```
Getting Started

- Installing QC Engine
- Community

## 4.1 Installing QC Engine

You can install `qcengine` with `conda`, with `pip`, or by installing from source.

### 4.1.1 Conda

You can update `qcengine` using `conda`:

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

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

### 4.1.2 Pip

To install `qcengine` with `pip` there are a few options, depending on which dependencies you would like to keep up to date:

```bash
pip install qcengine
```

### 4.1.3 Install from Source

To install `qcengine` from source, clone the repository from `github`:

```bash
git clone https://github.com/MolSSI/QCityEngine.git
cd qcengine
python setup.py install
```

or use `pip` for a local install:

```bash
pip install -e .
```

We recommend building a development environment with the following lines:
cd qcengine
python devtools/scripts/conda_env.py -n=qcngdev -p=3.6 devtools/conda-envs/psi.yaml
conda activate qcngdev

This will build out a new environment with several compute backends for qcengine which provides a platform to test and develop the code.

### 4.1.4 Test

Test a qcengine local install with pytest:

```bash
cd qcengine
pytest -v
```

### 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. Click [here](#) 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. See the following links:
  - QCSchema
  - QCElemental
  - QCEngine
  - QCFractal

#### 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 evangelists within the computational molecular science community for this code ecosystem. In return you will receive the following benefits:

- Work directly with a 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.

**User Interface**
• Single Compute
• Environment Detection

4.3 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.

4.3.1 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.ResultInput(
    molecule=mol,
    driver="energy",
    model={"method": "SCF", "basis": "sto-3g"},
    keywords={"scf_type": "df"})
```

4.3.2 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})
```
4.3.3 Results

The results contain a complete record of the computation:

```python
gt.return_result
-74.45994963230625

gt.properties.scf_dipole_moment
[0.0, 0.0, 0.6635967188869244]

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

A short description of the fields is as follow:

- **return_result** - the direct return of the driver input. That is energy and gradient call, respectively.
- **properties** - Values associated with the return_result such as the scf_one_electron_energy.
- **stdout** - The stdout or log of a programs run.
- **provenance** - A description of the calling program, version, wall time, etc.

A complete description of the input is also available in the output:

```python
gt.driver
energy
```

4.3.4 Fields

A list of all fields is available through the fields property on the input and output:

```python
gt.fields
['molecule', 'driver', 'model', 'id', 'schema_name', 'schema_version', 'keywords',
 'extras', 'provenance', 'return_result', 'success', 'properties', 'stdout', 'stderr',
 'error']
```

4.4 Environment Detection

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

4.4.1 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>
```
4.4.2 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>

>>> 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">
```

4.4.3 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'
}
```

Developer Documentation

- QCEngine API
- Changelog
- Development Guidelines

4.5 QCEngine API

4.5.1 qcengine Package

Base file for the dqm_compute module.

Functions

```python
compute(input_data, program[, raise_error, ...])
```

Executes a single quantum chemistry program given a QC Schema input.

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute_procedure</code></td>
<td>Runs a procedure (a collection of the quantum chemistry executions)</td>
</tr>
<tr>
<td><code>get_config</code></td>
<td>Returns the configuration key for qcengine.</td>
</tr>
<tr>
<td><code>get_molecule</code></td>
<td>Returns a QC JSON representation of a test molecule.</td>
</tr>
<tr>
<td><code>get_procedure</code></td>
<td>Returns a procedures executor class</td>
</tr>
<tr>
<td><code>get_program</code></td>
<td>Returns a program’s executor class</td>
</tr>
<tr>
<td><code>list_all_procedures</code></td>
<td>List all procedures registered by QCEngine.</td>
</tr>
<tr>
<td><code>list_all_programs</code></td>
<td>List all programs registered by QCEngine.</td>
</tr>
<tr>
<td><code>list_available_procedures</code></td>
<td>List all procedures that can be executed (found) by QCEngine.</td>
</tr>
<tr>
<td><code>list_available_programs</code></td>
<td>List all programs that can be executed (found) by QCEngine.</td>
</tr>
</tbody>
</table>

**compute**

qcengine.compute(input_data: Union[Dict[str, Any], ResultInput], program: str, raise_error: bool = False, local_options: Optional[Dict[str, Any]] = None, return_dict: bool = False) → Result

Executes a single quantum chemistry program given a QC Schema 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 program to execute the input with
- **raise_error** – Determines if compute should raise an error or not.
- **capture_output** – Determines if stdout/stderr should be captured.
- **local_options** – A dictionary of local configuration options
- **return_dict** – Returns a dict instead of qcelemental.models.ResultInput

**Returns** A QCSchema output or type depending on return_dict key

**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") – The name of the procedure to run
- **raise_error** (bool, optional) – 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.ResultInput

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

**Return type** dict, Optimization, FailedOperation

### get_config

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

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**

```python
cqengine.list_available_programs() → Set[str]
```

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

### 4.5.2 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 quantum chemistry program given a QC Schema 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
cqengine.compute.compute(input_data: Union[Dict[str, Any], ResultInput], program: str, raise_error: bool = False, local_options: Optional[Dict[str, Any]] = None, return_dict: bool = False) → Result
```

Executes a single quantum chemistry program given a QC Schema input.

**Parameters**

- `input_data` – A QCSchema input specification in dictionary or model from QCElemental.models
- `program` – The program to execute the input with
- `raise_error` – Determines if compute should raise an error or not.
- `capture_output` – Determines if stdout/stderr should be captured.
- `local_options` – A dictionary of local configuration options
- `return_dict` – Returns a dict instead of qcelemental.models.ResultInput

**Returns**

A QCSchema output or type depending on return_dict key

**Return type**

Result

#### compute_procedure

```python
cqengine.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 QCElemental.models

---

Chapter 4. Index
• **procedure** ("geometric") – 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.ResultInput

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

**Return type** dict, Optimization, FailedOperation

### 4.5.3 qcengine.config Module

Creates globals for the qcengine module

**Functions**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_config(*[, hostname, local_options])</td>
<td>Returns the configuration key for qcengine.</td>
</tr>
<tr>
<td>get_provenance_augments()</td>
<td>A representation of the current global configuration.</td>
</tr>
<tr>
<td>global_repr()</td>
<td></td>
</tr>
</tbody>
</table>

**get_config**

qcengine.config.get_config(*, hostname: Optional[str] = None, local_options: Dict[str, Any] = None) \[jobconfig\]  
Returns the configuration key for qcengine.

**get_provenance_augments**

qcengine.config.get_provenance_augments() \[Dict[str, str]\]

**global_repr**

qcengine.config.global_repr() \[str\]

A representation of the current global configuration.

**Classes**

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeDescriptor(<strong>data</strong>)</td>
<td>Description of an individual node</td>
</tr>
</tbody>
</table>

**NodeDescriptor**

class qcengine.config.NodeDescriptor(**data**)

Bases: pydantic.main.BaseModel

Description of an individual node

### 4.5. QCEngine API
4.5.4 qcengine.util Module

Several import utilities

Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Parameters</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_wrapper</td>
<td>Wraps compute for timing, output capturing, and raise protection</td>
<td>([capture_output, raise_error])</td>
<td>Dict[str, Any]</td>
</tr>
<tr>
<td>get_module_function</td>
<td>Obtains a function from a given string</td>
<td>(module, func_name[, subpackage=None])</td>
<td>Callable[... Any]</td>
</tr>
<tr>
<td>model_wrapper</td>
<td>Wrap input data in the given model, or return a controlled error</td>
<td>(input_data, model)</td>
<td></td>
</tr>
<tr>
<td>handle_output_metadata</td>
<td>Fuses general metadata and output together.</td>
<td>(output_data, metadata)</td>
<td></td>
</tr>
</tbody>
</table>

**compute_wrapper**

qcengine.util.compute_wrapper(capture_output: bool = True, raise_error: bool = False) → Dict[str, Any]

Wraps compute for timing, output capturing, and raise protection

**get_module_function**

qcengine.util.get_module_function(module: str, func_name: str, subpackage=None) → Callable[[..., Any]

Obtains a function from a given string

Parameters

- **module** (str) – The module to pull the function from
- **func_name** (str) – The name of the function to acquire, can be in a subpackage
- **subpackage** (None, optional) – Explicitly import a subpackage if required

Returns **ret** – The requested functions

Return type **function**
Example

```python
# Import numpy.linalg.eigh
f = get_module_function("numpy", "linalg.eigh")
f(np.ones((2, 2)))
```

model_wrapper

```python
qcengine.util.model_wrapper(input_data: Dict[str, Any], model: BaseModel) → BaseModel
```
Wrap input data in the given model, or return a controlled error

handle_output_metadata

```python
qcengine.util.handle_output_metadata(output_data: Union[Dict[str, Any], BaseModel], metadata: Dict[str, Any], raise_error: bool = False, return_dict: bool = True) → 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.Result

### 4.5.5 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 QCEngine.</td>
</tr>
<tr>
<td><code>list_available_programs()</code></td>
<td>List all programs that can be executed (found) by QCEngine.</td>
</tr>
<tr>
<td><code>register_program(entry_point)</code></td>
<td>Register a new ProgramHarness with QCEngine.</td>
</tr>
</tbody>
</table>

#### get_program

```python
qcengine.programs.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_programs

```python
qcengine.programs.list_all_programs() → Set[str]
```
List all programs registered by QCEngine.

#### list_available_programs

```python
qcengine.programs.list_available_programs() → Set[str]
```
List all programs that can be executed (found) by QCEngine.
register_program

qcengine.programs.register_program(entry_point: ProgramHarness) → None
Register a new ProgramHarness with QEngine.

4.6 Changelog

4.6.1 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.

4.6.2 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 @srl 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) QC Engine 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(..., outfiles=[]) 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.

4.6.3 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.

4.6.4 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.

4.6.5 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.

4.6.6 v0.6.1 / 2019-02-20

Bug Fixes

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

4.6.7 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.

4.6.8 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.

4.6.9 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.

4.6.10 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.
4.7 Development Guidelines

Core development guidelines go here.
PYTHON MODULE INDEX

q

qcengine, 13
qcengine.compute, 16
qcengine.config, 17
qcengine.programs, 19
qcengine.util, 18
| C   | compute() (in module qcengine), 14                   |
|     | compute() (in module qcengine.compute), 16         |
|     | compute_procedure() (in module qcengine), 14       |
|     | compute_procedure() (in module qcengine.compute), 16 |
|     | compute_wrapper() (in module qcengine.util), 18    |
| G   | get_config() (in module qcengine), 15              |
|     | get_config() (in module qcengine.config), 17       |
|     | get_module_function() (in module qcengine.util), 18 |
|     | get_molecule() (in module qcengine), 15            |
|     | get_procedure() (in module qcengine), 15           |
|     | get_program() (in module qcengine), 15             |
|     | get_program() (in module qcengine.programs), 19    |
|     | get_provenance_augments() (in module qcengine.config), 17 |
|     | global_repr() (in module qcengine.config), 17     |
| H   | handle_output_metadata() (in module qcengine.util), 19 |
| L   | list_all_procedures() (in module qcengine), 15     |
|     | list_all_programs() (in module qcengine), 15       |
|     | list_all_programs() (in module qcengine.programs), 19 |
|     | list_available_procedures() (in module qcengine), 15 |
|     | list_available_programs() (in module qcengine), 16 |
|     | list_available_programs() (in module qcengine.programs), 19 |
| M   | model_wrapper() (in module qcengine.util), 19     |
| N   | NodeDescriptor (class in qcengine.config), 17     |
| Q   | qcengine (module), 13                             |
|     | qcengine.compute (module), 16                     |
|     | qcengine.config (module), 17                      |
|     | qcengine.programs (module), 19                    |
|     | qcengine.util (module), 18                        |
| R   | register_program() (in module qcengine.programs), 20 |