QCEngine Documentation

Release 0.14.0+0.gc8ae155.dirty

The QCArchive Development Team

Feb 05, 2020

CONTENTS:

1 Program Execution **3**

2 Backends **5**

3 Configuration Determination **7**

4 Program and Procedure Information **9**

5 Index **11**

- 5.1 Install QCEngine 11
- 5.2 Single Compute 12
- 5.3 Environment Detection 14
- 5.4 Command Line Interface 16
- 5.5 Program Overview 18
- 5.6 Molecular Mechanics 19
- 5.7 QCEngine API 20
- 5.8 Changelog 32

Python Module Index **41**

Index **43**

Quantum chemistry program executor and IO standardizer (QCSchema) for quantum chemistry.

CHAPTER ONE

PROGRAM EXECUTION

A simple example of QCEngine's capabilities is as follows:

```
>>> 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:

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

The results contain a complete record of the computation:

```
>>> 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
```

CHAPTER TWO

BACKENDS

Currently available compute backends for single results are as follow:

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

In addition, several procedures are available:

- Geometry Optimization:
  - geomeTRIC

CHAPTER

THREE

CONFIGURATION DETERMINATION

In addition, QCEngine can automatically determine the following quantites:

- 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.

```
>>> 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'>
```

CHAPTER

FOUR

PROGRAM AND PROCEDURE INFORMATION

Available programs and procedures may be printed using the *CLI*:

```
>>> 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
...
```

CHAPTER

FIVE

INDEX

Getting Started

- *Install QCEngine*

5.1 Install QCEngine

You can install qcengine with `conda` or with `pip`.

5.1.1 Conda

You can install qcengine using conda:

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

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

5.1.2 Pip

You can also install QCEngine using `pip`:

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

5.1.3 Test the Installation

Note: QCEngine 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`:

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

From `pip`:

```
>>> pip install pytest
```

Then, run the following command:

```
>>> pytest --pyargs qcengine
```

5.1.4 Developing from Source

If you are a developer and want to make contributions Engine, you can access the source code from github.

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:

```
>>> 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:

```
>>> 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:

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

5.2.3 Results

The results contain a complete record of the computation:

```
>>> 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

class `qcelemental.models.`**`AtomicInput`**

The MolSSI Quantum Chemistry Schema

Parameters

- **id** (*str, Optional*) – An optional ID of the ResultInput object.
- **schema_name** (*ConstrainedStrValue, Default: qcschema_input*)
- **schema_version** (*int, Default: 1*)
- **molecule** (`Molecule`) – The molecule to use in the computation.
- **driver** (*{energy,gradient,hessian,properties}*) – Allowed quantum chemistry driver values.
- **model** (`Model`) – The quantum chemistry model specification for a given operation to compute against
- **keywords** (*Dict[str, Any], Default: {}*) – The program specific keywords to be used.
- **protocols** (`AtomicResultProtocols`, Optional) – Protocols regarding the manipulation of a Result output data.
- **extras** (*Dict[str, Any], Default: {}*) – Extra fields that are not part of the schema.
- **provenance** (`Provenance`, Optional) – Provenance information.

5.2.5 Returned Fields

class qcelemental.models.**AtomicResult**

Parameters

- **id** (*str, Optional*) – An optional ID of the ResultInput object.
- **schema_name** (*ConstrainedStrValue, Default: qcschema_output*)
- **schema_version** (*int, Default: 1*)
- **molecule** (`Molecule`) – The molecule to use in the computation.
- **driver** (*{energy,gradient,hessian,properties}*) – Allowed quantum chemistry driver values.
- **model** (`Model`) – The quantum chemistry model specification for a given operation to compute against
- **keywords** (*Dict[str, Any], Default: {}*) – The program specific keywords to be used.
- **protocols** (`AtomicResultProtocols`, Optional) – Protocols regarding the manipulation of a Result output data.
- **extras** (*Dict[str, Any], Default: {}*) – Extra fields that are not part of the schema.
- **provenance** (`Provenance`) – Provenance information.
- **properties** (`AtomicResultProperties`) – Named properties of quantum chemistry computations following the MolSSI QCSchema.
- **wavefunction** (`WavefunctionProperties`, Optional) – None
- **return_result** (*Union[float, Array, Dict[str, Any]]*) – The value requested by the 'driver' attribute.
- **stdout** (*str, Optional*) – The standard output of the program.
- **stderr** (*str, Optional*) – The standard error of the program.
- **success** (*bool*) – The success of a given programs execution. If False, other fields may be blank.
- **error** (`ComputeError`, Optional) – A complete description of the error.

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.

```
>>> 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:

```
>>> 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'>
```

5.3.3 Global Environment

The global environment can also be inspected directly.

```
>>> 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 `NodeDescription` schema:

class `qcengine.config.`**NodeDescriptor**(***data: Dict[str, Any]*)
    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 QCEngine environment.

Arguments

`category` The information categories to show. Choices include:

- `version`: Print version of QCEngine 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 QCEngine for each program can be found below:

5.5.1 Quantum Chemistry

Program	Production	E	G	H	Properties	Wavefunction
CFOUR		✓	✓		✓	
Entos		✓	✓	✓		✓
GAMESS		✓	✓		✓	
Molpro	✓	✓	✓		✓	
NWChem		✓	✓	✓	✓	
Psi4	✓	✓	✓	✓	✓	✓
Q-Chem		✓	✓	✓		
Terachem		✓	✓			
Turbomole		✓	✓			

5.5.2 Semi-Empirical

Program	Production	E	G	H	Properties	Wavefunction
MOPAC	✓	✓	✓		✓	

5.5.3 AI Potential

Program	Production	E	G	H	Properties
TorchANI	✓	✓	✓		✓

5.5.4 Molecular Mechanics

Program	Production	E	G	H	Properties
OpenMM		✓	✓		✓
RDKit	✓	✓	✓		✓

5.5.5 Analytical Corrections

Program	Production	E	G	H	Properties
DFTD3	✓	✓	✓		✓

5.6 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 paramters.

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

5.6.1 Example

```
>>> 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
```

5.6.2 OpenMM

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

Method	Basis
smirnoff99Frosst-1.1.0	smirnoff
openff-1.0.0	smirnoff
openff_unconstrained-1.0.0	smirnoff

Other forcefields may be available depending on your version of the `openforcefield` toolkit, see their docs for more information.

5.6.3 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:

Method	Basis
UFF	None
MMFF94	None
MMFF94s	None

Developer Documentation

- *QCEngine API*
- *Changelog*

5.7 QCEngine API

5.7.1 qcengine Package

Base file for the dqm_compute module.

Functions

`compute`(input_data, program[, raise_error, ...])	Executes a single quantum chemistry program given a QC Schema input.
`compute_procedure`(input_data, procedure[, ...])	Runs a procedure (a collection of the quantum chemistry executions)
`get_config`(*[, hostname, local_options])	Returns the configuration key for qcengine.
`get_molecule`(name)	Returns a QC JSON representation of a test molecule.
`get_procedure`(name)	Returns a procedures executor class
`get_program`(name[, check])	Returns a program's executor class
`list_all_procedures`()	List all procedures registered by QCEngine.
`list_all_programs`()	List all programs registered by QCEngine.
`list_available_procedures`()	List all procedures that can be exectued (found) by QCEngine.
`list_available_programs`()	List all programs that can be exectued (found) by QCEngine.
`register_program`(entry_point)	Register a new ProgramHarness with QCEngine.
`unregister_program`(name)	Unregisters a given program.

compute

qcengine.**compute**(*input_data: Union[Dict[str, Any], 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 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** (*Union[Dict[str, Any], 'AtomicInput']*) – A QCSchema input specification in dictionary or model from QCElemental.models
- **program** (*str*) – The program to execute the input with.
- **raise_error** (*bool, optional*) – Determines if compute should raise an error or not.
- **retries** (*int, optional*) – The number of random tries to retry for.
- **local_options** (*Optional[Dict[str, Any]], optional*) – A dictionary of local configuration options
- **return_dict** (*bool, optional*) – Returns a dict instead of qcelemental.models.AtomicInput

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"}*) – 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: 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 exectued (found) by QCEngine.

list_available_programs

qcengine.**list_available_programs**() → Set[str]
List all programs that can be exectued (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(mdi_options, program, molecule, . . .)	

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_commands()	Send the supported MDI commands through MDI
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_ncommands()	Send the number of supported MDI commands through MDI
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_commands() → str
Send the supported MDI commands through MDI

Returns **command_string** – String containing the name of each supported command

Return type str

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

send_forces() → numpy.ndarray
Send the nuclear forces through MDI

Returns **forces** – Forces on the nuclei

Return type np.ndarray

send_masses() → numpy.ndarray
Send the nuclear masses through MDI

Returns **masses** – Atomic masses

Return type np.ndarray

send_multiplicity() → int
Send the electronic multiplicity through MDI

Returns **multiplicity** – Multiplicity of the system

Return type int

send_natoms() → int

Send the number of atoms through MDI

Returns **natom** – Number of atoms

Return type int

send_ncommands() → int

Send the number of supported MDI commands through MDI

Returns **ncommands** – Number of supported commands

Return type int

send_total_charge() → float

Send the total system charge through MDI

Returns **charge** – Total charge of the system

Return type float

start() → None

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

stop() → None

Stop listening for MDI commands

update_molecule(*key: str*, *value*)

Update the molecule

Parameters

- **key** (*str*) – Key of the molecular element to update
- **value** – Update value

Class Inheritance Diagram

MDIServer

5.7.2 qcengine.compute Module

Integrates the computes together

Functions

compute(input_data, program[, raise_error, …])	Executes a single quantum chemistry program given a QC Schema input.
compute_procedure(input_data, procedure[, …])	Runs a procedure (a collection of the quantum chemistry executions)

compute

qcengine.compute.**compute**(*input_data: Union[Dict[str, Any], 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 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** (*Union[Dict[str, Any], 'AtomicInput']*) – A QCSchema input specification in dictionary or model from QCElemental.models
- **program** (*str*) – The program to execute the input with.
- **raise_error** (*bool, optional*) – Determines if compute should raise an error or not.
- **retries** (*int, optional*) – The number of random tries to retry for.
- **local_options** (*Optional[Dict[str, Any]], optional*) – A dictionary of local configuration options
- **return_dict** (*bool, optional*) – Returns a dict instead of qcelemental.models.AtomicInput

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 QCElemental.models
- **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.AtomicInput

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

Return type dict, OptimizationResult, FailedOperation

5.7.3 qcengine.config Module

Creates globals for the qcengine module

Functions

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: Dict[str, Any] = None*) → qcengine.config.TaskConfig

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

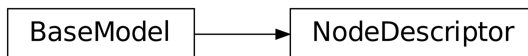
NodeDescriptor(**data)	Description of an individual node

NodeDescriptor

class qcengine.config.**NodeDescriptor**(***data: Dict[str, Any]*)
Bases: pydantic.main.BaseModel

Description of an individual node

Class Inheritance Diagram



5.7.4 qcengine.util Module

Several import utilities

Functions

compute_wrapper([capture_output, raise_error])	Wraps compute for timing, output capturing, and raise protection
model_wrapper(input_data, model)	Wrap input data in the given model, or return a controlled error
handle_output_metadata(output_data, metadata)	Fuses general metadata and output together.
create_mpi_invocation(executable, task_config)	Create the launch command for an MPI-parallel task
execute(command[, infiles, outfiles, ...])	Runs a process in the background until complete.

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

model_wrapper

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

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.AtomicResult

create_mpi_invocation

qcengine.util.**create_mpi_invocation**(*executable: str, task_config: qcengine.config.TaskConfig*) → List[str]
Create the launch command for an MPI-parallel task

Parameters

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

execute

qcengine.util.**execute**(*command: List[str], infiles: Optional[Dict[str, str]] = None, outfiles: Optional[List[str]] = None, *, as_binary: Optional[List[str]] = None, scratch_name: Optional[str] = None, scratch_directory: Optional[str] = None, scratch_suffix: Optional[str] = None, scratch_messy: bool = False, scratch_exist_ok: bool = False, blocking_files: Optional[List[str]] = None, timeout: Optional[int] = None, interupt_after: Optional[int] = None, environment: Optional[Dict[str, str]] = None, shell: Optional[bool] = False*) → Tuple[bool, Dict[str, Any]]
Runs a process in the background until complete.

Returns True if exit code zero

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.
- **interupt_after** (*int, optional*) – Interupt the process (not hard kill) after n seconds.
- **environment** (*dict, optional*) – The environment to run in
- **shell** (*bool, optional*) – Run command through the shell.

Raises `FileExistsError` – If any file in *blocking* is present

Examples

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)

5.7.5 qcengine.programs Package

Functions

`get_program`(name[, check])	Returns a program’s executor class
`list_all_programs`()	List all programs registered by QCEngine.
`list_available_programs`()	List all programs that can be exectued (found) by QCEngine.
`register_program`(entry_point)	Register a new ProgramHarness with QCEngine.
`unregister_program`(name)	Unregisters a given program.

get_program

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

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

List all programs registered by QCEngine.

list_available_programs

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

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

register_program

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

Register a new ProgramHarness with QCEngine.

unregister_program

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

Unregisters a given program.

Classes

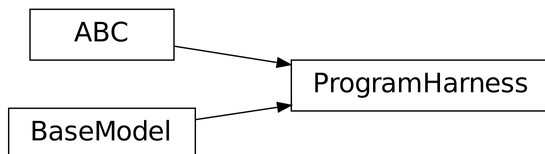
ProgramHarness(**kwargs)

ProgramHarness

`class qcengine.programs.ProgramHarness(**kwargs)`

Bases: `pydantic.main.BaseModel`, `abc.ABC`

Class Inheritance Diagram



5.8 Changelog

5.8.1 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.

5.8.2 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.

5.8.3 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.
- (GH#172) Models renamed according to https://github.com/MolSSI/QCElemental/issues/155, particularly `ResultInput -> AtomicInput`, `Result -> AtomicResult`, `Optimization -> OptimizationResult`.

Bug Fixes

5.8.4 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.

5.8.5 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.

5.8.6 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.

5.8.7 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

5.8.8 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.

5.8.9 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 @vivacebelles 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`.

5.8.10 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.

5.8.11 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 QCSchema.
- (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(..., 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.

5.8.12 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`.

5.8.13 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.

5.8.14 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.

5.8.15 v0.6.1 / 2019-02-20

Bug Fixes

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

5.8.16 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.

5.8.17 v0.5.2 / 2019-02-13

Enhancements

- (GH#24) Improves load times dramatically by delaying imports and cpuutils.
- (GH#25) Code base linting.
- (GH#30) Ensures Psi4 output is already returned and Pydantic v0.20+ changes.

5.8.18 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.

5.8.19 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.

PYTHON MODULE INDEX

q

qcengine, 20
qcengine.compute, 26
qcengine.config, 27
qcengine.programs, 30
qcengine.util, 28

INDEX

A

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

C

compute() (*in module qcengine*), 20
compute() (*in module qcengine.compute*), 26
compute_procedure() (*in module qcengine*), 21
compute_procedure() (*in module qcengine.compute*), 26
compute_wrapper() (*in module qcengine.util*), 28
create_mpi_invocation() (*in module qcengine.util*), 29

E

execute() (*in module qcengine.util*), 29

G

get_config() (*in module qcengine*), 21
get_config() (*in module qcengine.config*), 27
get_molecule() (*in module qcengine*), 21
get_procedure() (*in module qcengine*), 21
get_program() (*in module qcengine*), 22
get_program() (*in module qcengine.programs*), 30
get_provenance_augments() (*in module qcengine.config*), 27
global_repr() (*in module qcengine.config*), 27

H

handle_output_metadata() (*in module qcengine.util*), 29

L

list_all_procedures() (*in module qcengine*), 22
list_all_programs() (*in module qcengine*), 22
list_all_programs() (*in module qcengine.programs*), 31
list_available_procedures() (*in module qcengine*), 22
list_available_programs() (*in module qcengine*), 22
list_available_programs() (*in module qcengine.programs*), 31

M

MDIServer (*class in qcengine*), 23
model_wrapper() (*in module qcengine.util*), 29

N

NodeDescriptor (*class in qcengine.config*), 15, 28

P

ProgramHarness (*class in qcengine.programs*), 31

Q

qcengine (*module*), 20
qcengine.compute (*module*), 26
qcengine.config (*module*), 27
qcengine.programs (*module*), 30
qcengine.util (*module*), 28

R

recv_coords() (*qcengine.MDIServer method*), 23
recv_elements() (*qcengine.MDIServer method*), 23
recv_masses() (*qcengine.MDIServer method*), 24
recv_multiplicity() (*qcengine.MDIServer method*), 24
recv_total_charge() (*qcengine.MDIServer method*), 24
register_program() (*in module qcengine*), 22
register_program() (*in module qcengine.programs*), 31
run_energy() (*qcengine.MDIServer method*), 24

S

send_commands() (*qcengine.MDIServer method*), 24
send_coords() (*qcengine.MDIServer method*), 24
send_elements() (*qcengine.MDIServer method*), 24
send_energy() (*qcengine.MDIServer method*), 24
send_forces() (*qcengine.MDIServer method*), 24
send_masses() (*qcengine.MDIServer method*), 24
send_multiplicity() (*qcengine.MDIServer method*), 24

send_natoms() (*qcengine.MDIServer method*), 25
send_ncommands() (*qcengine.MDIServer method*), 25
send_total_charge() (*qcengine.MDIServer method*), 25
start() (*qcengine.MDIServer method*), 25
stop() (*qcengine.MDIServer method*), 25

U

unregister_program() (*in module qcengine*), 22
unregister_program() (*in module qcengine.programs*), 31
update_molecule() (*qcengine.MDIServer method*), 25