QISquick Documentation

Contents

Module `qisquick`

Sub-modules

- qisquick.circuits
- qisquick.dbconfig
- qisquick.qis_logger
- qisquick.run_experiment
- qisquick.statblock
- qisquick.tools
- qisquick.transpilertools

Module `qisquick.circuits`

Classes

Class `Premades`

```
class Premades(size, truth_value, measure=True, seed=None)
```

QuantumCircuit subclass to store new attributes used to automate circuit generation. Used with TestCircuit

Creates a Premades object that wraps QuantumCircuits to carry additional information. Most important is that the Premades object stores the uniform interface parameters for generating new circuits.

Args

`size` : **`int`** Width of the desired circuit. i.e. the register size of the quantum register defining it.

`truth_value` : **`int`** An integer to encode in any oracles that the circuit uses. Usually used to define the “right” value for the circuit to return. E.g. the correct value for a grover’s search to find.

`measure` : `bool` Optional. If True, adds measurement operators to the end of the circuit.
`seed` : `int` Optional. If not None, the provided seed is used to set random state for reproducibility.

Ancestors (in MRO)

- qiskit.circuit.quantumcircuit.QuantumCircuit

Class variables

Variable `circ_lib` dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Instance variables

Variable `cr`

Variable `qr`

Methods

Method `bv`

```
def bv(self)
```

Implements a Bernstein-Vazirani algorithm circuit, where the oracle encodes self.truth_value.

Returns

`None`:

Method `grover`

```
def grover(self)
```

Create and return a quantum circuit implementation of Grover's search algorithm.

Method `islands`

```
def islands(self)
```

Represents hub-and-spoke constraint topologies. Selects hubs at random and assigns a random number of spokes to them. Selects new hubs after all qubits have been consumed, up to (size) layers

Method `m_to_m`

```
def m_to_m(self)
```

Method `qft`

```
def qft(self)
```

implementation stolen shamelessly from github.com/Qiskit/qiskit-terra/blob/master/examples/python/qft.p

Method `toff`

```
def toff(self)
```

Create a series of toffoli gates across the size of the circuit. If not 3 | size, then some qubits will not be used in each layer

Method `two_bell`

```
def two_bell(self)
```

Method `uniform_random`

```
def uniform_random(self)
```

Creates a circuit whose gates are uniformly chosen from {H, X, Y, Z, S, T, CX} and whose CX endpoints are chosen uniformly from available qubits

Class `TestCircuit`

```
class TestCircuit()
```

Primary object of qisquick. Associates a Premades object along with its statistics and execution environment

Constructor.

Args

`stats` : **`Statblock`** Stores all relevant statistics. Written at TestCircuit creation/association with a Premades, at job submission, and post-execution.

`compiled_circ` : **`Premades`** A transpiled version of self.circuit, using the transpilerConfig from stats

`backend` : **`str`** Identifier of IBM QX backend to use for transpilation and execution.

`job_id` : **`str`** Unique identifier returned by execute().

`transpiler_config` : **`TranspilerConfig`** Backend state used by PassManager to transpile circuit.

`circuit` : **`Premades`** Experimental object inherited from QuantumCircuit. Usually made from existing test set.

Instance variables

Variable `id`

Variable `job`

Static methods

`Method generate`

```
def generate(case, size, truth_value=None, measure=True, seed=None)
```

Static shortcut method to generate a TestCircuit and attach a Premades object to it. Same as calling add_circ() on an existing TestCircuit.

`Method run_all_tests`

```
def run_all_tests(tests, pass_manager=None, generate_compiled=True, be=None,
attempts=1)
```

Given a circuit or list of circuits to execute, it executes all of them and writes all results to the appropriate db. Depending on parameters, a custom PassManager can be used, and the circuits will also be compiled before execution.

Args

`tests` : `List[TestCircuit]` Circuits to be tested
`pass_manager` : `PassManager` Custom PassManager to use for transpilation, if desired. Default: IBM default
`generate_compiled` : `bool` If True, will transpile circuits prior to execution
`be` : `Backend` IBM backend to use for transpilation and execution. Default: _preferred_backend
`attempts` Number of times to transpile the circuits to generate average compile time
`Returns` : `None` (`but writes results to statistics database as a side effect`)

Methods

Method `add_circ`

```
def add_circ(self, case, size, truth_value=None, measure=True, seed=None)
```

Given an empty TestCircuit, add a circuit to it.

Args

`case` : `str` Dictionary key corresponding to circuit-generating function OR an existing circuit of type Premades
`size` : `int` width of circuit, in qubits. If case is a str, a circuit of size will be created. If case is a PreMades, then size should match the existing Premades circ_size attribute.
`truth_value` : `Union[int, None]` Integer corresponding to basis vector that should be returned by the circuit when executed on an ideal simulator. If case is a str, then truth_value will be encoded in any oracles created by Premades circuit functions. If case is a PreMades, then truth_value should match the existing Premades truth_value attribute.
`measure` : `bool` Optional. Add measurement operators to created circuit. Unused if case if of type Premades
`seed` : `int` Optional. Sets Random state for reproducibility. Unused if case is of type Premades.

Returns

`None`:

Method `get_circ_backend`

```
def get_circ_backend(self, hub='ibm-q-afrl', default_backend=None)
```

Helper function to map a backend's string ID to its object.

Args

`hub` : `str` Provider owning the backend
`default_backend` : `str` String identifier of backend

Returns

`qiskit.providers.ibmq.ibmqbackend.IBMQBackend`:

Method `get_ideal_result`

```
def get_ideal_result(self)
```

Returns the distribution associated with executing a TestCircuit.compiled_circuit on the QASM sim.

Method `get_post_stats`

```
def get_post_stats(self)
```

Retrieves, but does not store, results from the execution of this TestCircuit.

Method `get_status_done`

```
def get_status_done(self)
```

Method `run_job`

```
def run_job(self)
```

Executes self.compiled_circ on self.backend and register it as a running job in the Running table

Returns

`None`:

Method `transpile_test`

```
def transpile_test(self, pass_manager=None, default_be=None, ATTEMPTS=1)
```

Transpile TestCircuit with provided pass_manager and register statistics, but do not execute.

Args

`pass_manager` : `PassManager` Custom PassManager to use to transpile this circuit.
`default_be` : `str` Optional. Default backend to use for transpilation; defaults to _preferred_backend defined in run_experiment.py
`ATTEMPTS` : `int` Optional. Number of transpile tests to be run to generate averages.

Returns

`qiskit.circuit.quantumcircuit.QuantumCircuit`: `Returns the compiled circuit for chaining; also saves it t`
self.compiled_circ as a side-effect.

Module `qisquick.dbconfig`

Functions

Function `create_all_tables`

```
def create_all_tables(db='data/circuit_data.sqlite')
```

Conditionally creates db file at db_location if it does not exist, and conditionally creates all tables if they do not exist. If everything already exists, no changes are made.

Args

`db` : `str` Location to create the db at. If no path and/or name were passed when the experiment was run, defaults to ‘circuit_data.sqlite’
`Returns` : `None`

Function `drop_in_progress`

```
def drop_in_progress(db, done)
```

Function `insert_in_progress`

```
def insert_in_progress(db, jobs)
```

Function `insert_objects`

```
def insert_objects(db, tcs)
```

Function `is_empty`

```
def is_empty(db, table_name)
```

Function `load_in_progress`

```
def load_in_progress(db)
```

Function `partition_writes`

```
def partition_writes(db, tcs)
```

Function `record_exists`

```
def record_exists(db, uuid)
```

Function `retrieve_objects`

```
def retrieve_objects(db, ids)
```

Function `set_db_location`

```
def set_db_location(location)
```

Function `update_objects`

```
def update_objects(db, tcs)
```

Function `write_all_stats`

```
def write_all_stats(db)
```

Function `write_objects`

```
def write_objects(db, tcs)
```

Function `write_stats`

```
def write_stats(db, ids)
```

Module `qisquick.qis_logger`

Functions

Function `config_logger`

```
def config_logger(verbosity)
```

Provides initial setup of logging system by setting format, separating out Qiskit and other 3rd party logging systems, and defining the log location.

Args

`verbosity` : `int` Derived from the count of v's if passed as CLI param, or passed directly via verbosity param in run_experiment call. Should be in range(4).

Function `get_module_logger`

```
def get_module_logger(name, file='qls.log', lvl=0)
```

" Should be called by each module to ensure the logger can correctly trace message sources

Module `qisquick.run_experiment`

Functions

Function `check_running`

```
def check_running()
```

Checks the "Running" table of the linked db to see if any batches in progress have been executed by the IBM backend. If so, it retrieves the job details, saves them to the main stats db and deletes the record from Running.

Args:

Returns

`completed` : `List[str]` ids of jobs that have completed since last checked.

Function `create_all`

```
def create_all(size=4, truth_value=5, seed=None, filename=None)
```

Quick functionality test. Creates and returns one copy of each test set object, and generates diagrams of them.

Args

`size` : `int` Width of circuit to generate. Sizes > ~14 qubits are likely to take a long time.
`truth_value` : `int` Desired base state for the circuit to return under measurement, if executed on an ideal sim. This is not applicable for all test circuits (e.g. QFT)
`seed` : `int` Set random state for reproducibility.
`filename` : `str` If given, will cause each generated circuit to create a .png of its composer format.

Returns

`List[qls.circuits.TestCircuit]`: `A list containing one TestCircuit object for each Premades test circuit` in Premades.circ_lib

Function `get_batches`

```
def get_batches(tcs, batch_size=25)
```

IBM QX devices expect relatively small lists of circuits to run. This function takes a list of circuits of arbitrary size and returns a list of list of circuits, where each inner list contains batch_size circuits.

Args

`tcs` **:** `List` List to be batched

batch_size (int):

Returns

`List`**[**`List`**[**`Any`**]]:** `Original list` **of** `circuits`**,** `split into batch_size`

Function `get_cli_args`

```
def get_cli_args()
```

Uses argparse to parse arguments when this module is called directly. If being imported, these same flags can be passed as named parameters to the run_experinment() function call. Keys are described below:

Keys

`check_only` **:** `bool` If True the checking routine for recovering executed jobs from the IBM backend is run, but the experiment itself is not. Otherwise both are run

`run_only` **:** `bool` Opposite of check_only. If True the experiment is run but results are not checked for. otherwise, both are run

`verbosity` **:** `Union`**[**`str`**,** `int`**]** Called with -v to -vvv when done from the command line. Called with an integer in range(4) if called from import.

Function `main`

```
def main(argv)
```

Main function. Called automatically on execution as main module.

Args:

Returns

`None`**:**

Function `run_experiment`

```
def run_experiment(experiment, db_location=None, provider=None, backend=None,
**kwargs)
```

Function provided for run_experiment.py imports into other systems. This can be called with a user defined experiment script and will perform the same tasks as if main() was called on that function

Args

`experiment` **:** `Callable` The function defined by the user to run their experiment

`db_location` **:** `str` Optional. Should be of the form 'relative/path/dbName.sqlite' Defaults to data/circuit_data.sqlite

`provider` **:** `str` Optional. String reference used to retrieve provider object. Defaults to ibm-q

`backend` **:** `str` Optional. String reference used to retrieve backend. Defaults to ibmq_16_melbourne `kwargs` Dictionary of arguments corresponding to those defined by the get_cli_args function in this module.

Returns: None. But as a side-effect will write to the Circs and Stats tables at db_location.

Function `run_local_experiment`

```
def run_local_experiment()
```

By Brandon Kamaka, 30 Jan 2020. Reproducibility experiment to validate test bed Create a series of test circuits, and transpile each series with distinct options from various layout and SWAP optimizing papers. Compare success, SWAP efficiency, and time efficiency metrics

Args:

Returns

List[str]: List of ids of circuits created and tested by this experiment.

Module `qisquick.statblock`

Classes

Class `Statblock`

```
class Statblock(parent)
```

Associated with each TestCircuit on creation. Stores object properties relevant for analyzing experimental results or reproducing them.

Instance variables

Variable `backend`

Variable `job_id`

Methods

Method `to_dict`

```
def to_dict(self, writeable=False)
```

Module `qisquick.tools`

Functions

Function `cleanup_matrix`

```
def cleanup_matrix(mx, norm)
```

Function `norm_matrix`

```
def norm_matrix(mx, norm)
```

Function `real_matrix`

```
def real_matrix(mx)
```

Function `refresh`

```
def refresh(size, qreg_name='qr', creg_name='cr', circ_name='qc')
```

Function `trim_matrix`

```
def trim_matrix(mx, dim)
```

Takes an nxn matrix and returns a reduced matrix of dimension dim x dim corresponding to the upper-left square of the original matrix

Module `qisquick.transpilertools`

Functions

Function `get_basic_pm`

```
def get_basic_pm(transpiler_config, level=0)
```

Get a pre-populated PassManager from the native Qiskit implementation.

Args

`transpiler_config` **:** `TranspileConfig` Configuration used to generate the tailored PassManager.
`level` **:** `int` Optional. Qiskit Transpiler optimization level to target.

Returns

`PassManager` PassManager instance associated with the provided config.

Function `get_modified_pm`

```
def get_modified_pm(pass_manager, version, pass_type, new_pass)
```

Modifies a provided PassManager instance by exchanging swap or layout passes with others of the same basic type.

Args

`pass_manager` **:** `qiskit.transpiler.passmanager.PassManager` PassManager instance to modify.
`version` **:** `int` Which optimization level the original PassManager was targeted at.
`pass_type` **:** `str` Type of pass to exchange. Must be one of (‘swap’, ‘layout’)
`new_pass` **:** `BasePass` The pass to insert into pass_manager in place of that pass_manager’s pass of type (type).

Returns

qiskit.transpiler.passmanager.PassManager: Modified PassManager instance.

Function `get_passes_str`

```
def get_passes_str(pm)
```

Returns str of actual passes embedded in the provided PassManager object.

Function `get_transpiler_config`

```
def get_transpiler_config(circs, be='ibmq_16_melbourne', layout=None, optimization_level=None,
callback=None)
```

Given a list of circuits and a backend to execute them on, return a list of transpiler configs of the same length such that configs[i] is the config for circs[i]

Args

`circs` : `Union[List[TestCircuit], TestCircuit]` Single circuit or List of circuits to compile configurations for

`be` : `IBMQBackend` Backend object to execute the circuits on.

`layout` : `Layout` Optional. Initial layout to use.

`optimization_level` : `int` Optional. IBM transpiler optimization level to target [0, 3].

`callback` : `Callable` Optional. Function to call at the end of execution of each pass in the PassManager.

Returns

`List[TranspileConfig]`: `List` **of** `transpiler configurations associated with` circs. Also writes to self.transpiler_config as a side-effect.

Generated by *pdoc* 0.7.2 (https://pdoc3.github.io).