projectq Documentation

Release 0.1.3

a

May 15, 2017

Contents

1 Tutorial **3**

1.1 Getting started . . . 3

1.2 Detailed instructions and OS-specific hints . . . 4

1.3 Basic quantum program . . . 5

2 Examples **7**

2.1 Quantum Random Numbers . . . 7

2.2 Quantum Teleportation . . . 8

2.3 Shor’s algorithm for factoring . . . 11

3 Code Documentation **13**

3.1 backends . . . 13

3.2 cengines . . . 18

3.3 libs . . . 25

3.4 meta . . . 28

3.5 ops . . . 31

3.6 setups . . . 40

3.7 types . . . 44

Python Module Index **47**

ProjectQ is an open-source software framework for quantum computing. It aims at providing tools which facilitate **inventing**, **implementing**, **testing**, **debugging**, and **running** quantum algorithms using either classical hardware or actual quantum devices.

The **four core principles** of this open-source effort are

1. **Open & Free**: *ProjectQ is released under the* **Apache 2** *license*
2. **Simple learning curve**: *It is implemented in Python and has an intuitive syntax*
3. **Easily extensible**: *Anyone can contribute to the compiler, the embedded domain-specific language, and libraries*
4. **Code quality**: *Code reviews, continuous integration testing (unit and functional tests)*

Please cite

- Damian S. Steiger, Thomas Häner, and Matthias Troyer "ProjectQ: An Open Source Software Framework for Quantum Computing" [arxiv:1612.08091]
- Thomas Häner, Damian S. Steiger, Krysta M. Svore, and Matthias Troyer "A Software Methodology for Compiling Quantum Programs" [arxiv:1604.01401]

Contents

- *Tutorial*: Tutorial containing instructions on how to get started with ProjectQ.
- *Examples*: Example implementations of few quantum algorithms
- *Code Documentation*: The code documentation of ProjectQ.

CHAPTER 1

Tutorial

Getting started

To start using ProjectQ, simply run

```
python -m pip install --user projectq
```

or, alternatively, clone/download this repo (e.g., to your /home directory) and run

```
cd /home/projectq
python -m pip install --user .
```

Note: The setup will try to build a C++-Simulator, which is much faster than the Python implementation. If it fails, you may use the *–without-cppsimulator* parameter, i.e.,

```
python -m pip install --user --global-option=--without-cppsimulator .
```

and the framework will use the **slow Python simulator instead**. Note that this only works if the installation has been tried once without the *–without-cppsimulator* parameter and hence all requirements are now installed. See the instructions below if you want to run larger simulations. The Python simulator works perfectly fine for the small examples (e.g., running Shor's algorithm for factoring 15 or 21).

Note: If building the C++-Simulator does not work out of the box, consider specifying a different compiler. For example:

```
env CC=g++-5 python -m pip install --user projectq
```

Please note that the compiler you specify must support **C++11**!

Detailed instructions and OS-specific hints

Ubuntu: After having installed the build tools (for g++):

```
sudo apt-get install build-essential
```

You only need to install Python (and the package manager). For version 3, run

```
sudo apt-get install python3 python3-pip
```

When you then run

```
sudo pip3 install --user projectq
```

all dependencies (such as numpy and pybind11) should be installed automatically.

Windows: It is easiest to install a pre-compiled version of Python, including numpy and many more useful packages. One way to do so is using, e.g., the Python3.5 installers from python.org or ANACONDA. Installing ProjectQ right away will succeed for the (slow) Python simulator (i.e., with the *–without-cppsimulator* flag). For a compiled version of the simulator, install the Visual C++ Build Tools and the Microsoft Windows SDK prior to doing a pip install. The built simulator will not support multi-threading due to the limited OpenMP support of msvc.

Should you want to run multi-threaded simulations, you can install a compiler which supports newer OpenMP versions, such as MinGW GCC and then manually build the C++ simulator with OpenMP enabled.

macOS: These are the steps to install ProjectQ on a new Mac:

Install XCode:

```
xcode-select --install
```

Next, you need to install Python and pip. One way of doing so is using macports to install Python 3.5 and the corresponding version of pip. Visit macports.org and install the latest version (afterwards open a new terminal). Then, use macports to install Python 3.5 by

```
sudo port install python35
```

It might show a warning that if you intend to use python from the terminal, you should also install

```
sudo port install py35-readline
```

Install pip by

```
sudo port install py35-pip
```

Next, we can install ProjectQ with the high performance simulator written in C++. Therefore, we first need to install a suitable compiler which supports OpenMP and instrinsics. One option is to install clang 3.9 also using macports (note: gcc installed via macports does not work as the gcc compiler claims to support instrinsics, while the assembler of gcc does not support it)

```
sudo port install clang-3.9
```

ProjectQ is now installed by:

```
env CC=clang-mp-3.9 env CXX=clang++-mp-3.9 python3.5 -m pip install --user␣
↪projectq
```

If you don't want to install clang 3.9, you can try and specify your preferred compiler by changing CC and CXX in the above command (the compiler must support **C++11**).

Should something go wrong when compiling the C++ simulator extension in one of the above installation procedures, you can turn off this feature using the `--without-cppsimulator` parameter (note: this only works if one of the above installation methods has been tried and hence all requirements are now installed), i.e.,

```
python3.5 -m pip install --user --global-option=--without-cppsimulator projectq
```

While this simulator works fine for small examples, it is suggested to install the high performance simulator written in C++.

If you just want to install ProjectQ with the (slow) Python simulator and no compiler, then first try to install ProjectQ with the default compiler

```
python3.5 -m pip install --user projectq
```

which most likely will fail and then use the command above using the `--without-cppsimulator` parameter.

Basic quantum program

To check whether the installation worked, try running the following basic example:

```
from projectq import MainEngine  # import the main compiler engine
from projectq.ops import H, Measure  # import the operations we want to perform␣
↪(Hadamard and measurement)

eng = MainEngine()  # create a default compiler (the back-end is a simulator)
qubit = eng.allocate_qubit()  # allocate 1 qubit

H | qubit  # apply a Hadamard gate
Measure | qubit  # measure the qubit

eng.flush()  # flush all gates (and execute measurements)
print("Measured {}".format(int(qubit)))  # output measurement result
```

Which creates random bits (0 or 1).

CHAPTER 2

Examples

All example codes can be found on GitHub.

Quantum Random Numbers

The most basic example is a quantum random number generator (QRNG). It can be found in the examples-folder of ProjectQ. The code looks as follows

```
from projectq.ops import H, Measure
from projectq import MainEngine

# create a main compiler engine
eng = MainEngine()

# allocate one qubit
q1 = eng.allocate_qubit()

# put it in superposition
H | q1

# measure
Measure | q1

eng.flush()
# print the result:
print("Measured: {}".format(int(q1)))
```

Running this code three times may yield, e.g.,

```
$ python examples/quantum_random_numbers.py
Measured: 0
$ python examples/quantum_random_numbers.py
Measured: 0
```

```
$ python examples/quantum_random_numbers.py
Measured: 1
```

These values are obtained by simulating this quantum algorithm classically. By changing three lines of code, we can run an actual quantum random number generator using the IBM Quantum Experience back-end:

```
$ python examples/quantum_random_numbers_ibm.py
Measured: 1
$ python examples/quantum_random_numbers_ibm.py
Measured: 0
```

All you need to do is:

- Create an account for IBM's Quantum Experience
- And perform these minor changes:

```
--- ../examples/quantum_random_numbers.py
+++ ../examples/quantum_random_numbers_ibm.py
@@ -1,8 +1,10 @@
+import projectq.setups.ibm
 from projectq.ops import H, Measure
 from projectq import MainEngine
+from projectq.backends import IBMBackend

 # create a main compiler engine
-eng = MainEngine()
+eng = MainEngine(IBMBackend())

 # allocate one qubit
 q1 = eng.allocate_qubit()
```

Quantum Teleportation

Alice has a qubit in some interesting state $|\psi\rangle$, which she would like to show to Bob. This does not really make sense, since Bob would not be able to look at the qubit without collapsing the superposition; but let's just assume Alice wants to send her state to Bob for some reason. What she can do is use quantum teleportation to achieve this task. Yet, this only works if Alice and Bob share a Bell-pair (which luckily happens to be the case). A Bell-pair is a pair of qubits in the state

$$|A\rangle \otimes |B\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle \right)$$

They can create a Bell-pair using a very simple circuit which first applies a Hadamard gate to the first qubit, and then flips the second qubit conditional on the first qubit being in $|1\rangle$. The circuit diagram can be generated by calling the function

```
def create_bell_pair(eng):
  b1 = eng.allocate_qubit()
  b2 = eng.allocate_qubit()

  H | b1
  CNOT | (b1, b2)

  return b1, b2
```

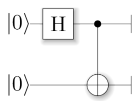
with a main compiler engine which has a CircuitDrawer back-end, i.e.,

```
from projectq import MainEngine
from projectq.backends import CircuitDrawer

from teleport import create_bell_pair

# create a main compiler engine
drawing_engine = CircuitDrawer()
eng = MainEngine(drawing_engine)

create_bell_pair(eng)

eng.flush()
print(drawing_engine.get_latex())
```

The resulting LaTeX code can be compiled to produce the circuit diagram:

```
$ python examples/bellpair_circuit.py > bellpair_circuit.tex
$ pdflatex bellpair_circuit.tex
```

The output looks as follows:



Now, this Bell-pair can be used to achieve the quantum teleportation: Alice entangles her qubit with her share of the Bell-pair. Then, she measures both qubits; one in the Z-basis (Measure) and one in the Hadamard basis (Hadamard, then Measure). She then sends her measurement results to Bob who, depending on these outcomes, applies a Pauli-X or -Z gate.

The complete example looks as follows:

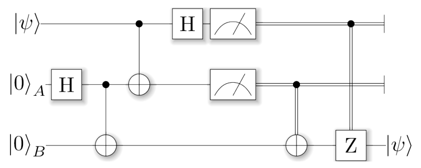
```
import projectq.setups.default
from projectq.ops import H, X, Z, Rz, CNOT, Measure
from projectq import MainEngine
from projectq.meta import Dagger, Control


def create_bell_pair(eng):
  b1 = eng.allocate_qubit()
  b2 = eng.allocate_qubit()

  H | b1
  CNOT | (b1, b2)

  return b1, b2


def run_teleport(eng, state_creation_function, verbose=False):
  # make a Bell-pair
  b1, b2 = create_bell_pair(eng)

  # Alice creates a nice state to send
  psi = eng.allocate_qubit()
  if verbose:
    print("Alice is creating her state from scratch, i.e., |0>.")
  state_creation_function(eng, psi)
```

```

    # entangle it with Alice's b1
    CNOT | (psi, b1)
    if verbose:
      print("Alice entangled her qubit with her share of the Bell-pair.")

    # measure two values (once in Hadamard basis) and send the bits to Bob
    H | psi
    Measure | (psi, b1)
    message_to_bob = [int(psi), int(b1)]
    if verbose:
      print("Alice is sending the message {} to Bob.".format(message_to_bob))

    # Bob may have to apply up to two operation depending on the message sent
    # by Alice:
    with Control(eng, b1):
      X | b2
    with Control(eng, psi):
      Z | b2

    # try to uncompute the psi state
    if verbose:
      print("Bob is trying to uncompute the state.")
    with Dagger(eng):
      state_creation_function(eng, b2)

    # check whether the uncompute was successful. The simulator only allows to
    # delete qubits which are in a computational basis state.
    del b2
    eng.flush()

    if verbose:
      print("Bob successfully arrived at |0>")


if __name__ == "__main__":
    # create a main compiler engine with a simulator backend:
    eng = MainEngine()

    # define our state-creation routine, which transforms a |0> to the state
    # we would like to send. Bob can then try to uncompute it and, if he arrives
    # back at |0>, we know that the teleportation worked.
    def create_state(eng, qb):
      H | qb
      Rz(1.21) | qb

    # run the teleport and then, let Bob try to uncompute his qubit:
    run_teleport(eng, create_state, verbose=True)
```

and the corresponding circuit can be generated using

```
$ python examples/teleport_circuit.py > teleport_circuit.tex
$ pdflatex teleport_circuit.tex
```

which produces (after renaming of the qubits inside the tex-file):



Shor's algorithm for factoring

As a third example, consider Shor's algorithm for factoring, which for a given (large) number N determines the two prime factor p_1 and p_2 such that $p_1 \cdot p_2 = N$ in polynomial time! This is a superpolynomial speed-up over the best known classical algorithm (which is the number field sieve) and enables the breaking of modern encryption schemes such as RSA on a future quantum computer.

A tiny bit of number theory There is a small amount of number theory involved, which reduces the problem of factoring to period-finding of the function

$$f(x) = a^x \bmod N$$

for some *a* (relative prime to N, otherwise we get a factor right away anyway by calling *gcd(a,N)*). The period *r* for a function *f(x)* is the number for which $f(x) = f(x+r)\forall x$ holds. In this case, this means that $a^x = a^{x+r} \pmod N \; \forall x$. Therefore, $a^r = 1 + qN$ for some integer q and hence, $a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1) = qN$. This suggests that using the gcd on *N* and $a^{r/2} \pm 1$ we may find a factor of *N*!

Factoring on a quantum computer: An example At the heart of Shor's algorithm lies modular exponentiation of a classically known constant (denoted by *a* in the code) by a quantum superposition of numbers x, i.e.,

$$|x\rangle|0\rangle \mapsto |x\rangle|a^x \bmod N\rangle$$

Using $N = 15$ and $a = 2$, and applying this operation to the uniform superposition over all x leads to the superposition (modulo renormalization)

$$|0\rangle|1\rangle + |1\rangle|2\rangle + |2\rangle|4\rangle + |3\rangle|8\rangle + |4\rangle|1\rangle + |5\rangle|2\rangle + |6\rangle|4\rangle + \cdots$$

In Shor's algorithm, the second register will not be touched again before the end of the quantum program, which means it might as well be measured now. Let's assume we measure 2; this collapses the state above to

$$|1\rangle|2\rangle + |5\rangle|2\rangle + |9\rangle|2\rangle + \cdots$$

The period of *a* modulo *N* can now be read off. On a quantum computer, this information can be accessed by applying an inverse quantum Fourier transform to the x-register, followed by a measurement of x.

Implementation There is an implementation of Shor's algorithm in the examples folder. It uses the implementation by Beauregard, arxiv:0205095 to factor an n-bit number using 2n+3 qubits. In this implementation, the modular exponentiation is carried out using modular multiplication and shift. Furthermore it uses the semi-classical quantum Fourier transform [see arxiv:9511007]: Pulling the final measurement of the *x*-register through the final inverse quantum Fourier transform allows to run the 2n modular multiplications serially, which keeps one from having to store the 2n qubits of x.

Let's run it using the ProjectQ simulator:

```
$ python3 examples/shor.py

projectq
--------
Implementation of Shor's algorithm.
```

```
Number to factor: 15

Factoring N = 15: 00000001

Factors found :-) : 3 * 5 = 15
```

Simulating Shor's algorithm at the level of single-qubit gates and CNOTs already takes quite a bit of time for larger numbers than 15. To turn on our **emulation feature**, which does not decompose the modular arithmetic to low-level gates, but carries it out directly instead, we can change the line

```

# Filter function, which defines the gate set for the first optimization
# (don't decompose QFTs and iQFTs to make cancellation easier)
def high_level_gates(eng, cmd):
  g = cmd.gate
  if g == QFT or get_inverse(g) == QFT or g == Swap:
    return True
  if isinstance(g, BasicMathGate):
    return False
    if isinstance(g, AddConstant):
      return True
    elif isinstance(g, AddConstantModN):
```

in examples/shor.py to *return True*. This allows to factor, e.g. $N = 4,028,033$ in under 3 minutes on a regular laptop!

The most important part of the code is

```
for k in range(2 * n):
  current_a = pow(a, 1 << (2 * n - 1 - k), N)
  # one iteration of 1-qubit QPE
  H | ctrl_qubit
  with Control(eng, ctrl_qubit):
    MultiplyByConstantModN(current_a, N) | x

  # perform inverse QFT --> Rotations conditioned on previous outcomes
  for i in range(k):
    if measurements[i]:
      R(-math.pi/(1 << (k - i))) | ctrl_qubit
  H | ctrl_qubit

  # and measure
  Measure | ctrl_qubit
  eng.flush()
  measurements[k] = int(ctrl_qubit)
  if measurements[k]:
    X | ctrl_qubit
```

which executes the 2n modular multiplications conditioned on a control qubit *ctrl_qubit* in a uniform superposition of 0 and 1. The control qubit is then measured after performing the semi-classical inverse quantum Fourier transform and the measurement outcome is saved in the list *measurements*, followed by a reset of the control qubit to state 0.

CHAPTER 3

Code Documentation

Welcome to the package documentation of ProjectQ. You may now browse through the entire documentation and discover the capabilities of the ProjectQ framework.

For a detailed documentation of a subpackage or module, click on its name below:

backends

Module contents

Contains back-ends for ProjectQ.

This includes:

- a debugging tool to print all received commands (CommandPrinter)
- a circuit drawing engine (which can be used anywhere within the compilation chain)
- a simulator with emulation capabilities
- a resource counter (counts gates and keeps track of the maximal width of the circuit)
- an interface to the IBM Quantum Experience chip (and simulator).

class `projectq.backends.`**`CircuitDrawer`**`(`*accept_input=False*, *default_measure=0*`)`

CircuitDrawer is a compiler engine which generates TikZ code for drawing quantum circuits.

The circuit can be modified by editing the settings.json file which is generated upon first execution. This includes adjusting the gate width, height, shadowing, line thickness, and many more options.

After initializing the CircuitDrawer, it can also be given the mapping from qubit IDs to wire location (via the *`set_qubit_locations()`* function):

```
circuit_backend = CircuitDrawer()
circuit_backend.set_qubit_locations({0: 1, 1: 0}) # swaps lines 0 and 1
eng = MainEngine(circuit_backend)
```

```
... # run quantum algorithm on this main engine

print(circuit_backend.get_latex()) # prints LaTeX code
```

To see the qubit IDs in the generated circuit, simply set the *draw_id* option in the settings.json file under "gates":"AllocateQubitGate" to True:

```
"gates": {
        "AllocateQubitGate": {
                "draw_id": True,
                "height": 0.15,
                "width": 0.2,
                "pre_offset": 0.1,
                "offset": 0.1
        },
        ...
```

The settings.json file has the following structure:

```
{
        "control": { # settings for control "circle"
                        "shadow": false,
                        "size": 0.1
        },
        "gate_shadow": true, # enable/disable shadows for all gates
        "gates": {
                        "GateClassString": {
                                GATE_PROPERTIES
                        }
                        "GateClassString2": {
                                ...
        },
        "lines": { # settings for qubit lines
                        "double_classical": true, # draw double-lines for␣
↪classical bits
                        "double_lines_sep": 0.04, # gap between the two lines for␣
↪double lines
                        "init_quantum": true, # start out with quantum bits
                        "style": "very thin" # line style
        }
}
```

All gates (except for the ones requiring special treatment) support the following properties:

```
"GateClassString": {
        "height": GATE_HEIGHT,
        "width": GATE_WIDTH
        "pre_offset": OFFSET_BEFORE_PLACEMENT,
        "offset": OFFSET_AFTER_PLACEMENT,
},
```

__init__(*accept_input=False*, *default_measure=0*)

Initialize a circuit drawing engine.

The TikZ code generator uses a settings file (settings.json), which can be altered by the user. It contains gate widths, heights, offsets, etc.

Parameters

- **`accept_input`** (*`bool`*) – If accept_input is true, the printer queries the user to input measurement results if the CircuitDrawer is the last engine. Otherwise, all measurements yield the result default_measure (0 or 1).
- **`default_measure`** (*`bool`*) – Default value to use as measurement results if accept_input is False and there is no underlying backend to register real measurement results.

`get_latex`()

Return the latex document string representing the circuit.

Simply write this string into a tex-file or, alternatively, pipe the output directly to, e.g., pdflatex:

```
python3 my_circuit.py | pdflatex
```

where my_circuit.py calls this function and prints it to the terminal.

`is_available`(*cmd*)

Specialized implementation of is_available: Returns True if the CircuitDrawer is the last engine (since it can print any command).

Parameters **`cmd`** (`Command`) – Command of which to check availability (all Commands can be printed).

Returns True, unless the next engine cannot handle the Command (if there is a next engine).

Return type availability (bool)

`receive`(*command_list*)

Receive a list of commands from the previous engine, print the commands, and then send them on to the next engine.

Parameters **`command_list`** (*`list<Command>`*) – List of Commands to print (and potentially send on to the next engine).

`set_qubit_locations`(*id_to_loc*)

Sets the qubit lines to use for the qubits explicitly.

To figure out the qubit IDs, simply use the setting *draw_id* in the settings file. It is located in "gates":"AllocateQubitGate". If draw_id is True, the qubit IDs are drawn in red.

Parameters **`id_to_loc`** (*`dict`*) – Dictionary mapping qubit ids to qubit line numbers.

Raises `RuntimeError` – If the mapping has already begun (this function needs be called before any gates have been received).

class `projectq.backends.`**`CommandPrinter`**(*accept_input=True*, *default_measure=False*, *in_place=False*)

CommandPrinter is a compiler engine which prints commands to stdout prior to sending them on to the next compiler engine.

`__init__`(*accept_input=True*, *default_measure=False*, *in_place=False*)

Initialize a CommandPrinter.

Parameters

- **`accept_input`** (*`bool`*) – If accept_input is true, the printer queries the user to input measurement results if the CommandPrinter is the last engine. Otherwise, all measurements yield 0.
- **`default_measure`** (*`bool`*) – Default measurement result (if accept_input is False).
- **`in_place`** (*`bool`*) – If in_place is true, all output is written on the same line of the terminal.

`is_available`(*cmd*)

Specialized implementation of is_available: Returns True if the CommandPrinter is the last engine (since it can print any command).

Parameters **`cmd`** (`Command`) – Command of which to check availability (all Commands can be printed).

Returns True, unless the next engine cannot handle the Command (if there is a next engine).

Return type availability (bool)

`receive`(*command_list*)

Receive a list of commands from the previous engine, print the commands, and then send them on to the next engine.

Parameters **`command_list`** (*`list<Command>`*) – List of Commands to print (and potentially send on to the next engine).

class `projectq.backends.`**`IBMBackend`**(*use_hardware=False*, *num_runs=1024*, *verbose=False*, *user=None*, *password=None*)

The IBM Backend class, which stores the circuit, transforms it to JSON QASM, and sends the circuit through the IBM API.

`__init__`(*use_hardware=False*, *num_runs=1024*, *verbose=False*, *user=None*, *password=None*)

Initialize the Backend object.

Parameters

- **`use_hardware`** (*`bool`*) – If True, the code is run on the IBM quantum chip (instead of using the IBM simulator)
- **`num_runs`** (*`int`*) – Number of runs to collect statistics. (default is 1024)
- **`verbose`** (*`bool`*) – If True, statistics are printed, in addition to the measurement result being registered (at the end of the circuit).
- **`user`** (*`string`*) – IBM Quantum Experience user name
- **`password`** (*`string`*) – IBM Quantum Experience password

`get_probabilities`(*qureg*)

Return the list of basis states with corresponding probabilities.

The measured bits are ordered according to the supplied quantum register, i.e., the left-most bit in the state-string corresponds to the first qubit in the supplied quantum register.

Warning: Only call this function after the circuit has been executed!

Parameters **`qureg`** (*`list<Qubit>`*) – Quantum register determining the order of the qubits.

Returns Dictionary mapping n-bit strings to probabilities.

Return type probability_dict (dict)

Raises `Exception` – If no data is available (i.e., if the circuit has not been executed).

`is_available`(*cmd*)

Return true if the command can be executed.

The IBM quantum chip can do X, Y, Z, T, Tdag, S, Sdag, and CX / CNOT.

Parameters **`cmd`** (`Command`) – Command for which to check availability

receive(*command_list*)
Receives a command list and, for each command, stores it until completion.

Parameters **command_list** – List of commands to execute

class projectq.backends.**ResourceCounter**
ResourceCounter is a compiler engine which counts the number of gates and max. number of active qubits.

gate_counts
dict – Dictionary of gate counts. The keys are string representations of the gate.

max_width
int – Maximal width (=max. number of active qubits at any given point).

__init__()
Initialize a resource counter engine.

Sets all statistics to zero.

is_available(*cmd*)
Specialized implementation of is_available: Returns True if the ResourceCounter is the last engine (since it can count any command).

Parameters **cmd** (Command) – Command for which to check availability (all Commands can be counted).

Returns True, unless the next engine cannot handle the Command (if there is a next engine).

Return type availability (bool)

receive(*command_list*)
Receive a list of commands from the previous engine, print the commands, and then send them on to the next engine.

Parameters **command_list** (*list<Command>*) – List of commands to receive (and count).

class projectq.backends.**Simulator**(*gate_fusion=False*, *rnd_seed=None*)
Simulator is a compiler engine which simulates a quantum computer using C++- based kernels.

OpenMP is enabled and the number of threads can be controlled using the OMP_NUM_THREADS environment variable, i.e.

```
export OMP_NUM_THREADS=4 # use 4 threads
export OMP_PROC_BIND=spread # bind threads to processors by spreading
```

__init__(*gate_fusion=False*, *rnd_seed=None*)
Construct the C++/Python-simulator object and initialize it with a random seed.

Parameters

- **gate_fusion** (*bool*) – If True, gates are cached and only executed once a certain gate-size has been reached (only has an effect for the c++ simulator).
- **rnd_seed** (*int*) – Random seed (uses random.randint(0, 1024) by default).

Example of gate_fusion: Instead of applying a Hadamard gate to 5 qubits, the simulator calculates the kronecker product of the 1-qubit matrices and then applies 1 5-qubit gate. This increases operational intensity and keeps the simulator from having to iterate through the state vector multiple times. Depending on the system (and, especially, number of threads), this may or may not be beneficial.

Note: If the C++ Simulator extension was not built or cannot be found, the Simulator defaults to a Python implementation of the kernels. While this is much slower, it is still good enough to run basic quantum algorithms.

If you need to run large simulations, check out the tutorial in the docs which gives futher hints on how to build the C++ extension.

cheat()

Access the ordering of the qubits and the state vector directly.

This is a cheat function which enables, e.g., more efficient evaluation of expectation values and debugging.

Returns A tuple where the first entry is a dictionary mapping qubit indices to bit-locations and the second entry is the corresponding state vector

is_available(*cmd*)

Specialized implementation of is_available: The simulator can deal with all arbitrarily-controlled single-qubit gates which provide a gate-matrix (via gate.get_matrix()).

Parameters **cmd** (Command) – Command for which to check availability (single-qubit gate, arbitrary controls)

Returns True if it can be simulated and False otherwise.

receive(*command_list*)

Receive a list of commands from the previous engine and handle them (simulate them classically) prior to sending them on to the next engine.

Parameters **command_list** (*list<Command>*) – List of commands to execute on the simulator.

cengines

The ProjectQ compiler engines package.

Module contents

class projectq.cengines.**AutoReplacer**(*decomposition_chooser=<function AutoReplacer.<lambda>>*)

The AutoReplacer is a compiler engine which uses engine.is_available in order to determine which commands need to be replaced/decomposed/compiled further. The loaded setup is used to find decomposition rules appropriate for each command (e.g., setups.default).

__init__(*decomposition_chooser=<function AutoReplacer.<lambda>>*)

Initialize an AutoReplacer.

Parameters **decomposition_chooser** (*function*) – A function which, given the Command to decompose and a list of potential Decomposition objects, determines (and then returns) the 'best' decomposition.

The default decomposition chooser simply returns the first list element, i.e., calling

```
repl = AutoReplacer()
```

Amounts to

```
def decomposition_chooser(cmd, decomp_list):
        return decomp_list[0]
repl = AutoReplacer(decomposition_chooser)
```

receive(*command_list*)
Receive a list of commands from the previous compiler engine and, if necessary, replace/decompose the gates according to the decomposition rules in the loaded setup.

Parameters **command_list** (*list<Command>*) – List of commands to handle.

class projectq.cengines.**BasicEngine**
Basic compiler engine: All compiler engines are derived from this class. It provides basic functionality such as qubit allocation/deallocation and functions that provide information about the engine's position (e.g., next engine).

This information is provided by the MainEngine, which initializes all further engines.

next_engine
BasicEngine – Next compiler engine (or the back-end).

main_engine
MainEngine – Reference to the main compiler engine.

is_last_engine
bool – True for the last engine, which is the back-end.

__init__()
Initialize the basic engine.

Initializes local variables such as _next_engine, _main_engine, etc. to None.

allocate_qubit(*dirty=False*)
Return a new qubit as a list containing 1 qubit object (quantum register of size 1).

Allocates a new qubit by getting a (new) qubit id from the MainEngine, creating the qubit object, and then sending an AllocateQubit command down the pipeline. If dirty=True, the fresh qubit can be replaced by a pre-allocated one (in an unknown, dirty, initial state). Dirty qubits must be returned to their initial states before they are deallocated / freed.

All allocated qubits are added to the MainEngine's set of active qubits as weak references. This allows proper clean-up at the end of the Python program (using atexit), deallocating all qubits which are still alive. Qubit ids of dirty qubits are registered in MainEngine's dirty_qubits set.

Parameters **dirty** (*bool*) – If True, indicates that the allocated qubit may be dirty (i.e., in an arbitrary initial state).

Returns Qureg of length 1, where the first entry is the allocated qubit.

allocate_qureg(*n*)
Allocate n qubits and return them as a quantum register, which is a list of qubit objects.

Parameters **n** (*int*) – Number of qubits to allocate

Returns Qureg of length n, a list of n newly allocated qubits.

deallocate_qubit(*qubit*)
Deallocate a qubit (and sends the deallocation command down the pipeline). If the qubit was allocated as a dirty qubit, add DirtyQubitTag() to Deallocate command.

Parameters **qubit** (BasicQubit) – Qubit to deallocate.

is_available(*cmd*)
Default implementation of is_available: Ask the next engine whether a command is available, i.e., whether it can be executed by the next engine(s).

Parameters **cmd** (Command) – Command for which to check availability.

Returns True if the command can be executed.

Raises

- *LastEngineException* – If is_last_engine is True but is_available is not
- `implemented.`

is_meta_tag_supported(*meta_tag*)
Check if there is a compiler engine handling the meta tag

Parameters

- **engine** – First engine to check (then iteratively calls getNextEngine)
- **meta_tag** – Meta tag class for which to check support

Returns True if one of the further compiler engines is a meta tag handler, i.e., engine.is_meta_tag_handler(meta_tag) returns True.

Return type supported (bool)

send(*command_list*)
Forward the list of commands to the next engine in the pipeline.

class projectq.cengines.**CommandModifier**(*cmd_mod_fun*)
CommandModifier is a compiler engine which applies a function to all incoming commands, sending on the resulting command instead of the original one.

__init__(*cmd_mod_fun*)
Initialize the CommandModifier.

Parameters **cmd_mod_fun** (*function*) – Function which, given a command cmd, returns the command it should send instead.

Example

```
def cmd_mod_fun(cmd):
        cmd.tags += [MyOwnTag()]
compiler_engine = CommandModifier(cmd_mod_fun)
...
```

receive(*command_list*)
Receive a list of commands from the previous engine, modify all commands, and send them on to the next engine.

Parameters **command_list** (*list<Command>*) – List of commands to receive and then (after modification) send on.

class projectq.cengines.**CompareEngine**
CompareEngine is an engine which saves all commands. It is only intended for testing purposes. Two CompareEngine backends can be compared and return True if they contain the same commmands.

class projectq.cengines.**Decomposition**(*replacement_fun*, *recogn_fun*)
The Decomposition class can be used to register a decomposition rule (by calling register_decomposition)

__init__(*replacement_fun*, *recogn_fun*)
Construct the Decomposition object.

Parameters

- **replacement_fun** – Function that, when called with a *Command* object, decomposes this command.

- **recogn_fun** – Function that, when called with a *Command* object, returns True if and only if the replacement rule can handle this command.

Every Decomposition is registered with the gate class. The Decomposition rule is then potentially valid for all objects which are an instance of that same class (i.e., instance of gate_object.__class__). All other parameters have to be checked by the recogn_fun, i.e., it has to decide whether the decomposition rule can indeed be applied to replace the given Command.

As an example, consider recognizing the Toffoli gate, which is a Pauli-X gate with 2 control qubits. The recognizer function would then be:

```
def recogn_toffoli(cmd):
        # can be applied if the gate is an X-gate with 2 control qubits:
        return len(cmd.control_qubits) == 2
```

and, given a replacement function *replace_toffoli*, the decomposition rule can be registered as

```
register_decomposition(X.__class__, decompose_toffoli, recogn_toffoli)
```

Note: See projectq.setups.decompositions for more example codes.

get_inverse_decomposition()
Return the Decomposition object which handles the inverse of the original command.

This simulates the user having added a decomposition rule for the inverse as well. Since decomposing the inverse of a command can be achieved by running the original decomposition inside a *with Dagger(engine):* statement, this is not necessary (and will be done automatically by the framework).

Returns Decomposition handling the inverse of the original command.

class projectq.cengines.**DummyEngine**(*save_commands=False*)
DummyEngine used for testing.

The DummyEngine forwards all commands directly to next engine. If self.is_last_engine == True it just discards all gates. By setting save_commands == True all commands get saved as a list in self.received_commands. Elements are appended to this list so they are ordered according to when they are received.

__init__(*save_commands=False*)
Initialize DummyEngine

Parameters **save_commands** (*default = False*) – If True, commands are saved in self.received_commands.

class projectq.cengines.**ForwarderEngine**(*engine*, *cmd_mod_fun=None*)
A ForwarderEngine is a trivial engine which forwards all commands to the next engine.

It is mainly used as a substitute for the MainEngine at lower levels such that meta operations still work (e.g., with Compute).

__init__(*engine*, *cmd_mod_fun=None*)
Initialize a ForwarderEngine.

Parameters

- **engine** (BasicEngine) – Engine to forward all commands to.
- **cmd_mod_fun** (*function*) – Function which is called before sending a command. Each command cmd is replaced by the command it returns when getting called with cmd.

receive(*command_list*)
Forward all commands to the next engine.

class `projectq.cengines.`**`IBMCNOTMapper`**

CNOT mapper for the IBM backend.

Transforms CNOTs such that all CNOTs within the circuit have the same target qubit (required by IBM backend). If necessary, it will flip around the CNOT gate by first applying Hadamard gates to both qubits, then CNOT with swapped control and target qubit, and finally Hadamard gates to both qubits.

Note: The mapper has to be run once on the entire circuit. Else, an Exception will be raised (if, e.g., several measurements are performed without re- initializing the mapper).

Warning: If the provided circuit cannot be mapped to the hardware layout without performing Swaps, the mapping procedure **raises an Exception**.

`__init__``()`

Initialize an IBM CNOT Mapper compiler engine.

Resets the mapping.

`is_available``(`*cmd*`)`

Check if the IBM backend can perform the Command cmd and return True if so.

Parameters **`cmd`** (`Command`) – The command to check

`receive``(`*command_list*`)`

Receives a command list and, for each command, stores it until completion.

Parameters **`command_list`** (*`list of Command objects`*) – list of commands to receive.

Raises `Exception` – If mapping the CNOT gates to 1 qubit would require Swaps. The current version only supports remapping of CNOT gates without performing any Swaps due to the large costs associated with Swapping given the CNOT constraints.

class `projectq.cengines.`**`InstructionFilter`**`(`*filterfun*`)`

The InstructionFilter is a compiler engine which changes the behavior of is_available according to a filter function. All commands are passed to this function, which then returns whether this command can be executed (True) or needs replacement (False).

`__init__``(`*filterfun*`)`

Constructor: The provided filterfun returns True for all commands which do not need replacement and False for commands that do.

Parameters **`filterfun`** (*`function`*) – Filter function which returns True for available commands, and False otherwise. filterfun will be called as filterfun(self, cmd).

`is_available``(`*cmd*`)`

Specialized implementation of BasicBackend.is_available: Forwards this call to the filter function given to the constructor.

Parameters **`cmd`** (`Command`) – Command for which to check availability.

`receive``(`*command_list*`)`

Forward all commands to the next engine.

Parameters **`command_list`** (*`list<Command>`*) – List of commands to receive.

exception `projectq.cengines.`**`LastEngineException`**`(`*engine*`)`

Exception thrown when the last engine tries to access the next one. (Next engine does not exist)

The default implementation of isAvailable simply asks the next engine whether the command is available. An engine which legally may be the last engine, this behavior needs to be adapted (see BasicEngine.isAvailable).

class `projectq.cengines.`**`LocalOptimizer`**(*m=5*)

LocalOptimizer is a compiler engine which optimizes locally (merging rotations, cancelling gates with their inverse) in a local window of user- defined size.

It stores all commands in a list of lists, where each qubit has its own gate pipeline. After adding a gate, it tries to merge / cancel successive gates using the get_merged and get_inverse functions of the gate (if available). For examples, see BasicRotationGate. Once a list corresponding to a qubit contains >=m gates, the pipeline is sent on to the next engine.

`__init__`(*m=5*)

Initialize a LocalOptimizer object.

Parameters **`m`** (*`int`*) – Number of gates to cache per qubit, before sending on the first gate.

`receive`(*command_list*)

Receive commands from the previous engine and cache them. If a flush gate arrives, the entire buffer is sent on.

class `projectq.cengines.`**`MainEngine`**(*backend=None*, *engine_list=None*)

The MainEngine class provides all functionality of the main compiler engine.

It initializes all further compiler engines (calls, e.g., .next_engine=...) and keeps track of measurement results and active qubits (and their IDs).

`next_engine`

BasicEngine – Next compiler engine (or the back-end).

`main_engine`

MainEngine – Self.

`active_qubits`

WeakSet – WeakSet containing all active qubits

`dirty_qubits`

Set – Containing all dirty qubit ids

`backend`

BasicEngine – Access the back-end.

`__init__`(*backend=None*, *engine_list=None*)

Initialize the main compiler engine and all compiler engines.

Sets ‘next_engine’- and ‘main_engine’-attributes of all compiler engines and adds the back-end as the last engine.

Parameters

- **`backend`** (`BasicEngine`) – Backend to send the circuit to.
- **`engine_list`** (*`list<BasicEngine>`*) – List of engines / backends to use as compiler engines.

Example

```
from projectq import MainEngine
eng = MainEngine() # will load default setup using Simulator backend
```

Alternatively, one can specify all compiler engines explicitly, e.g.,

Example

```
from projectq.cengines import TagRemover,AutoReplacer,LocalOptimizer
from projectq.backends import Simulator
from projectq import MainEngine
engines = [AutoReplacer(), TagRemover(), LocalOptimizer(3)]
eng = MainEngine(Simulator(), engines)
```

flush(*deallocate_qubits=False*)
Flush the entire circuit down the pipeline, clearing potential buffers (of, e.g., optimizers).

Parameters **deallocate_qubits** (*bool*) – If True, deallocates all qubits that are still alive (invalidating references to them by setting their id to -1)

get_measurement_result(*qubit*)
Return the classical value of a measured qubit, given that an engine registered this result previously (see setMeasurementResult).

Parameters **qubit** (BasicQubit) – Qubit of which to get the measurement result.

Example

```
from projectq.ops import H, Measure
from projectq import MainEngine
eng = MainEngine()
qubit = eng.allocate_qubit() # quantum register of size 1
H | qubit
Measure | qubit
eng.get_measurement_result(qubit[0]) == int(qubit)
```

get_new_qubit_id()
Returns a unique qubit id to be used for the next qubit allocation.

Returns New unique qubit id.

Return type new_qubit_id (int)

receive(*command_list*)
Forward the list of commands to the first engine.

Parameters **command_list** (*list<Command>*) – List of commands to receive (and then send on)

set_measurement_result(*qubit*, *value*)
Register a measurement result

The engine being responsible for measurement results needs to register these results with the master engine such that they are available when the user calls an int() or bool() conversion operator on a measured qubit.

Parameters

- **qubit** (BasicQubit) – Qubit for which to register the measurement result.
- **value** (*bool*) – Boolean value of the measurement outcome (True / False = 1 / 0 respectively)

class projectq.cengines.**TagRemover**(*tags=[<class 'projectq.meta._compute.ComputeTag'>, <class 'projectq.meta._compute.UncomputeTag'>]*)
TagRemover is a compiler engine which removes temporary command tags (see the tag classes such as LoopTag in projectq.meta._loop).

Removing tags is important (after having handled them if necessary) in order to enable optimizations across meta-function boundaries (compute/action/ uncompute or loops after unrolling)

__init__(*tags=[<class 'projectq.meta._compute.ComputeTag'>, <class 'projectq.meta._compute.UncomputeTag'>]*)

Construct the TagRemover.

Parameters **tags** – A list of meta tag classes (e.g., [ComputeTag, UncomputeTag]) denoting the tags to remove

receive(*command_list*)

Receive a list of commands from the previous engine, remove all tags which are an instance of at least one of the meta tags provided in the constructor, and then send them on to the next compiler engine.

Parameters **command_list** (*list<Command>*) – List of commands to receive and then (after removing tags) send on.

projectq.cengines.**register_decomposition**(*gate_class*, *gate_decomposer*, *gate_recognizer=None*)

Add a decomposition rule for compiling 'gate' to the global setup.

The decomposition rule is a Decomposition object (see _decomposition.py) and consists of a function which recognizes a command (i.e., determines whether it can handle it) and a function which executes the decomposition. The gate_class parameter determines the gate class for which the decomposition is valid (keeps the number of calls to recognize functions lower).

Parameters

- **gate_class** – Gate class for which the decomposition should be applicable; this parameter is only used to enable binary search on *gate_object.__class__*. If your class is defined as

```
class MyGate(BasicGate):
        pass
```

  Then you supply gate_class=MyGate However, if MyGate is overridden as often is the case when

```
MyGate = MyGate() # Because it allows the syntax MyGate | qubit
```

  then gate_class = MyGate.__class__

- **gate_decomposer** (*function*) – Function which, given the command to decompose, applies a sequence of gates corresponding to the high-level function of a gate of type gate_class.
- **gate_recognizer** (*function*) – Optional function which, given the command to decompose, returns whether the decomposition supports the given command. E.g., rotation gates may be rewritten using one decomposition for some angles, and another one for other angles. If no such function is provided, the decomposition rule will be valid for all gates of type gate_class.

libs

The library collection of ProjectQ which, for now, only consists of a tiny math library. Soon, more libraries will be added.

Subpackages

math

A tiny math library which will be extended thoughout the next weeks. Right now, it only contains the math functions necessary to run Beauregard's implementation of Shor's algorithm.

Module contents

class `projectq.libs.math.`**`AddConstant`**(*a*)

Add a constant to a quantum number represented by a quantum register, stored from low- to high-bit.

Example

```
qunum = eng.allocate_qureg(5) # 5-qubit number
X | qunum[1] # qunum is now equal to 2
AddConstant(3) | qunum # qunum is now equal to 5
```

`__init__`(*a*)

Initializes the gate to the number to add.

Parameters **a** (*int*) – Number to add to a quantum register.

It also initializes its base class, BasicMathGate, with the corresponding function, so it can be emulated efficiently.

`get_inverse`()

Return the inverse gate (subtraction of the same constant).

class `projectq.libs.math.`**`AddConstantModN`**(*a*, *N*)

Add a constant to a quantum number represented by a quantum register modulo N.

The number is stored from low- to high-bit, i.e., qunum[0] is the LSB.

Example

```
qunum = eng.allocate_qureg(5) # 5-qubit number
X | qunum[1] # qunum is now equal to 2
AddConstantModN(3, 4) | qunum # qunum is now equal to 1
```

`__init__`(*a*, *N*)

Initializes the gate to the number to add modulo N.

Parameters

- **a** (*int*) – Number to add to a quantum register (0 <= a < N).
- **N** (*int*) – Number modulo which the addition is carried out.

It also initializes its base class, BasicMathGate, with the corresponding function, so it can be emulated efficiently.

`get_inverse`()

Return the inverse gate (subtraction of the same number a modulo the same number N).

class `projectq.libs.math.`**`MultiplyByConstantModN`**(*a*, *N*)

Multiply a quantum number represented by a quantum register by a constant modulo N.

The number is stored from low- to high-bit, i.e., qunum[0] is the LSB.

Example

```
qunum = eng.allocate_qureg(5) # 5-qubit number
X | qunum[2] # qunum is now equal to 4
MultiplyByConstantModN(3,5) | qunum # qunum is now equal to 2 (mod 5)
```

`__init__`(*a*, *N*)

Initializes the gate to the number to multiply with modulo N.

Parameters

- **a** (*int*) – Number by which to multiply a quantum register (0 <= a < N).
- **N** (*int*) – Number modulo which the multiplication is carried out.

It also initializes its base class, BasicMathGate, with the corresponding function, so it can be emulated efficiently.

`projectq.libs.math.`**`SubConstant`**(*a*)

Subtract a constant from a quantum number represented by a quantum register, stored from low- to high-bit.

Parameters **a** (*int*) – Constant to subtract

Example

```
qunum = eng.allocate_qureg(5) # 5-qubit number
X | qunum[2] # qunum is now equal to 4
SubConstant(3) | qunum # qunum is now equal to 1
```

`projectq.libs.math.`**`SubConstantModN`**(*a*, *N*)

Subtract a constant from a quantum number represented by a quantum register modulo N.

The number is stored from low- to high-bit, i.e., qunum[0] is the LSB.

Parameters

- **a** (*int*) – Constant to add
- **N** (*int*) – Constant modulo which the addition of a should be carried out.

Example

```
qunum = eng.allocate_qureg(3) # 3-qubit number
X | qunum[1] # qunum is now equal to 2
SubConstantModN(4,5) | qunum # qunum is now equal to -2 = 6 = 1 (mod 5)
```

Module contents

meta

Contains meta statements which allow more optimal code while making it easier for users to write their code. Examples are *with Compute*, followed by an automatic uncompute or *with Control*, which allows the user to condition an entire code block upon the state of a qubit.

Module contents

The projectq.meta package features meta instructions which help both the user and the compiler in writing/producing efficient code. It includes, e.g.,

- Loop (with Loop(eng): ...)
- Compute/Uncompute (with Compute(eng): ..., [...], Uncompute(eng))
- Control (with Control(eng, ctrl_qubits): ...)
- Dagger (with Dagger(eng): ...)

class `projectq.meta.`**`Compute`**`(`*engine*`)`

Start a compute-section.

Example

```
with Compute(eng):
        do_something(qubits)
action(qubits)
Uncompute(eng) # runs inverse of the compute section
```

Warning: If qubits are allocated within the compute section, they must either be uncomputed and deallocated within that section or, alternatively, uncomputed and deallocated in the following uncompute section.

This means that the following examples are valid:

```
with Compute(eng):
        anc = eng.allocate_qubit()
        do_something_with_ancilla(anc)
        ...
        uncompute_ancilla(anc)
        del anc

do_something_else(qubits)

Uncompute(eng)  # will allocate a new ancilla (with a different id)
                # and then deallocate it again
```

```
with Compute(eng):
        anc = eng.allocate_qubit()
        do_something_with_ancilla(anc)
        ...

do_something_else(qubits)

Uncompute(eng)  # will deallocate the ancilla!
```

After the uncompute section, ancilla qubits allocated within the compute section will be invalid (and deallocated). The same holds when using CustomUncompute.

Failure to comply with these rules results in an exception being thrown.

__init__(*engine*)

Initialize a Compute context.

Parameters **engine** (BasicEngine) – Engine which is the first to receive all commands (normally: MainEngine).

class projectq.meta.**ComputeTag**

Compute meta tag.

class projectq.meta.**Control**(*engine*, *qubits*)

Condition an entire code block on the value of qubits being 1.

Example

```
with Control(eng, ctrlqubits):
        do_something(otherqubits)
```

__init__(*engine*, *qubits*)

Enter a controlled section.

Parameters

- **engine** – Engine which handles the commands (usually MainEngine)
- **qubits** (*list of Qubit objects*) – Qubits to condition on

Enter the section using a with-statement:

```
with Control(eng, ctrlqubits):
        ...
```

class projectq.meta.**CustomUncompute**(*engine*)

Start a custom uncompute-section.

Example

```
with Compute(eng):
        do_something(qubits)
action(qubits)
with CustomUncompute(eng):
        do_something_inverse(qubits)
```

Raises QubitManagementError – If qubits are allocated within Compute or within CustomUncompute context but are not deallocated.

__init__(*engine*)

Initialize a CustomUncompute context.

Parameters **engine** (BasicEngine) – Engine which is the first to receive all commands (normally: MainEngine).

class `projectq.meta.`**`Dagger`**(*engine*)

Invert an entire code block.

Use it with a with-statement, i.e.,

```
with Dagger(eng):
        [code to invert]
```

Warning: If the code to invert contains allocation of qubits, those qubits have to be deleted prior to exiting the 'with Dagger()' context.

This code is **NOT VALID**:

```
with Dagger(eng):
        qb = eng.allocate_qubit()
        H | qb # qb is still available!!!
```

The **correct way** of handling qubit (de-)allocation is as follows:

```
with Dagger(eng):
        qb = eng.allocate_qubit()
        ...
        del qb # sends deallocate gate (which becomes an allocate)
```

`__init__`(*engine*)

Enter an inverted section.

Parameters **`engine`** – Engine which handles the commands (usually MainEngine)

Example (executes an inverse QFT):

```
with Dagger(eng):
        QFT | qubits
```

class `projectq.meta.`**`DirtyQubitTag`**

Dirty qubit meta tag

class `projectq.meta.`**`Loop`**(*engine*, *num*)

Loop n times over an entire code block.

Example

```
with Loop(eng, 4):
        # [quantum gates to be executed 4 times]
```

Warning: If the code in the loop contains allocation of qubits, those qubits have to be deleted prior to exiting the 'with Loop()' context.

This code is **NOT VALID**:

```
with Loop(eng, 4):
        qb = eng.allocate_qubit()
        H | qb # qb is still available!!!
```

The **correct way** of handling qubit (de-)allocation is as follows:

```
with Loop(eng, 4):
        qb = eng.allocate_qubit()
        ...
        del qb # sends deallocate gate
```

`__init__`(*engine*, *num*)
Enter a looped section.

Parameters

- **`engine`** – Engine handling the commands (usually MainEngine)
- **`num`** (`int`) – Number of loop iterations

Example

```
with Loop(eng, 4):
        H | qb
        Rz(M_PI/3.) | qb
```

class `projectq.meta.`**`LoopTag`**(*num*)
Loop meta tag

`__init__`(*num*)

`loop_tag_id` = **0**

`projectq.meta.`**`Uncompute`**(*engine*)
Uncompute automatically.

Example

```
with Compute(eng):
        do_something(qubits)
action(qubits)
Uncompute(eng) # runs inverse of the compute section
```

class `projectq.meta.`**`UncomputeTag`**
Uncompute meta tag.

`projectq.meta.`**`get_control_count`**(*cmd*)
Return the number of control qubits of the command object cmd

ops

The operations collection consists of various default gates and is a work-in-progress, as users start to work with ProjectQ.

Module contents

`projectq.ops.`**`All`**
alias of *`Tensor`*

class `projectq.ops.`**`AllocateDirtyQubitGate`**
Dirty qubit allocation gate class

class `projectq.ops.`**`AllocateQubitGate`**
Qubit allocation gate class

class `projectq.ops.`**`BasicGate`**
Base class of all gates.

`__init__``()`
Initialize a basic gate.

Note: Set interchangeable qubit indices! (gate.interchangeable_qubit_indices)

As an example, consider

```
ExampleGate | (a,b,c,d,e)
```

where a and b are interchangeable. Then, call this function as follows

```
self.set_interchangeable_qubit_indices([[0,1]])
```

As another example, consider

```
ExampleGate2 | (a,b,c,d,e)
```

where a and b are interchangeable and, in addition, c, d, and e are interchangeable among themselves. Then, call this function as

```
self.set_interchangeable_qubit_indices([[0,1],[2,3,4]])
```

`generate_command``(`*qubits*`)`
Return a Command object which represents the gate acting on qubits.

Parameters **`qubits`** – see BasicGate.make_tuple_of_qureg(qubits)

Returns A Command object which represents the gate acting on qubits.

`get_inverse``()`
Return the inverse gate.

Standard implementation of get_inverse:

Raises *`NotInvertible`* – inverse is not implemented

`get_merged``(`*other*`)`
Return this gate merged with another gate.

Standard implementation of get_merged:

Raises *`NotMergeable`* – merging is not implemented

static **`make_tuple_of_qureg`**`(`*qubits*`)`
Convert quantum input of “gate | quantum input” to internal formatting.

A Command object only accepts tuples of Quregs (list of Qubit objects) as qubits input parameter. However, with this function we allow the user to use a more flexible syntax:

1.Gate | qubit

2.Gate | [qubit0, qubit1]

3.Gate | qureg

4.Gate | (qubit,)

5.Gate | (qureg, qubit)

where qubit is a Qubit object and qureg is a Qureg object. This function takes the right hand side of | and transforms it to the correct input parameter of a Command object which is:

1.-> Gate | ([qubit],)

2.-> Gate | ([qubit0, qubit1],)

3.-> Gate | (qureg,)

4.-> Gate | ([qubit],)

5.-> Gate | (qureg, [qubit])

Parameters **`qubits`** – a Qubit object, a list of Qubit objects, a Qureg object, or a tuple of Qubit or Qureg objects (can be mixed).

Returns A tuple containing Qureg (or list of Qubits) objects.

Return type Canonical representation (tuple<qureg>)

class `projectq.ops.`**`BasicMathGate`**(*math_fun*)

Base class for all math gates.

It allows efficient emulation by providing a mathematical representation which is given by the concrete gate which derives from this base class. The AddConstant gate, for example, registers a function of the form

```
def add(x):
        return (x+a,)
```

upon initialization. More generally, the function takes integers as parameters and returns a tuple / list of outputs, each entry corresponding to the function input. As an example, consider out-of-place multiplication, which takes two input registers and adds the result into a third, i.e., (a,b,c) -> (a,b,c+a*b). The corresponding function then is

```
def multiply(a,b,c)
        return (a,b,c+a*b)
```

`__init__`(*math_fun*)

Initialize a BasicMathGate by providing the mathematical function that it implements.

Parameters **`math_fun`** (*`function`*) – Function which takes as many int values as input, as the gate takes registers. For each of these values, it then returns the output (i.e., it returns a list/tuple of output values).

Example

```
def add(a,b):
        return (a,a+b)
BasicMathGate.__init__(self, add)
```

If the gate acts on, e.g., fixed point numbers, the number of bits per register is also required in order to describe the action of such a mathematical gate. For this reason, there is

```
BasicMathGate.get_math_function(qubits)
```

which can be overwritten by the gate deriving from BasicMathGate.

Example

```
def get_math_function(self, qubits):
        n = len(qubits[0])
        scal = 2.**n
        def math_fun(a):
                return (int(scal * (math.sin(math.pi * a / scal))),)
        return math_fun
```

class `projectq.ops.`**`BasicRotationGate`**`(`*angle*`)`

Defines a base class of a rotation gate.

A rotation gate has a continuous parameter (the angle), labeled 'angle' / self._angle. Its inverse is the same gate with the negated argument. Rotation gates of the same class can be merged by adding the angles. The continuous parameter is modulo 4 * pi, self._angle is in the interval [0, 4 * pi).

`__init__``(`*angle*`)`

Initialize a basic rotation gate.

Parameters **`angle`** (*`float`*) – Angle of rotation (saved modulo 4 * pi)

`get_inverse``()`

Return the inverse of this rotation gate (negate the angle, return new object).

`get_merged``(`*other*`)`

Return self merged with another gate.

Default implementation handles rotation gate of the same type, where angles are simply added.

Parameters **`other`** – Rotation gate of same type.

Raises *`NotMergeable`* – For non-rotation gates or rotation gates of different type.

Returns New object representing the merged gates.

`tex_str``()`

Return the Latex string representation of a BasicRotationGate.

Returns the class name and the angle as a subscript, i.e.

```
[CLASSNAME]$_[ANGLE]$
```

`projectq.ops.`**`C`**`(`*gate*, *n=1*`)`

Return n-controlled version of the provided gate.

Parameters

- **`gate`** – Gate to turn into its controlled version

- **n** – Number of controls (default: 1)

Example

```
C(NOT) | (c, q) # equivalent to CNOT | (c, q)
```

class `projectq.ops.`**`ClassicalInstructionGate`**

Classical instruction gate.

Base class for all gates which are not quantum gates in the typical sense, e.g., measurement, allocation/deallocation, ...

class `projectq.ops.`**`Command`**(*engine*, *gate*, *qubits*)

Class used as a container to store commands. If a gate is applied to qubits, then the gate and qubits are saved in a command object. Qubits are copied into WeakQubitRefs in order to allow early deallocation (would be kept alive otherwise). WeakQubitRef qubits don't send deallocate gate when destructed.

`gate`

The gate to execute

`qubits`

Tuple of qubit lists (e.g. Quregs). Interchangeable qubits are stored in a unique order

`control_qubits`

The Qureg of control qubits in a unique order

`engine`

The engine (usually: MainEngine)

`tags`

The list of tag objects associated with this command (e.g., ComputeTag, UncomputeTag, LoopTag, ...). tag objects need to support ==, != (__eq__ and __ne__) for comparison as used in e.g. TagRemover. New tags should always be added to the end of the list. This means that if there are e.g. two LoopTags in a command, tag[0] is from the inner scope while tag[1] is from the other scope as the other scope receives the command after the inner scope LoopEngine and hence adds its LoopTag to the end.

`all_qubits`

A tuple of control_qubits + qubits

`__init__`(*engine*, *gate*, *qubits*)

Initialize a Command object.

Note: control qubits (Command.control_qubits) are stored as a list of qubits, and command tags (Command.tags) as a list of tag- objects. All functions within this class also work if WeakQubitRefs are supplied instead of normal Qubit objects (see WeakQubitRef).

Parameters

- **engine** – engine which created the qubit (mostly the MainEngine)
- **gate** – Gate to be executed
- **qubits** – Tuple of quantum registers (to which the gate is applied)

`add_control_qubits`(*qubits*)

Add (additional) control qubits to this command object.

They are sorted to ensure a canonical order. Also Qubit objects are converted to WeakQubitRef objects to allow garbage collection and thus early deallocation of qubits.

Parameters

- **qubits** (*list of Qubit objects*) – List of qubits which control this
- **i.e., the gate is only executed if all qubits are in state 1.** (gate,) –

all_qubits
Get all qubits (gate and control qubits).

Returns a tuple T where T[0] is a quantum register (a list of WeakQubitRef objects) containing the control qubits and T[1:] contains the quantum registers to which the gate is applied.

control_qubits
Returns Qureg of control qubits.

engine
Return engine to which the qubits belong / on which the gates are executed.

get_inverse()
Get the command object corresponding to the inverse of this command.

Inverts the gate (if possible) and creates a new command object from the result.

Raises

- *NotInvertible* – If the gate does not provide an inverse (see
- `BasicGate.get_inverse`)

get_merged(*other*)
Merge this command with another one and return the merged command object.

Parameters **other** – Other command to merge with this one (self)

Raises

- *NotMergeable* – if the gates don't supply a get_merged()-function or can't
- be merged for other reasons.

interchangeable_qubit_indices
Return nested list of qubit indices which are interchangeable.

Certain qubits can be interchanged (e.g., the qubit order for a Swap gate). To ensure that only those are sorted when determining the ordering (see _order_qubits), self.interchangeable_qubit_indices is used. .. rubric:: Example

If we can interchange qubits 0,1 and qubits 3,4,5, then this function returns [[0,1],[3,4,5]]

class `projectq.ops.`**ControlledGate**(*gate*, *n=1*)
Controlled version of a gate.

Note: Use the meta function *C()* to create a controlled gate

A wrapper class which enables (multi-) controlled gates. It overloads the __or__-operator, using the first qubits provided as control qubits. The n control-qubits need to be the first n qubits. They can be in separate quregs.

Example

```
ControlledGate(gate, 2) | (qb0, qb2, qb3) # qb0 and qb2 are controls
C(gate, 2) | (qb0, qb2, qb3) # This is much nicer.
C(gate, 2) | ([qb0,qb2], qb3) # Is equivalent
```

Note: Use `C()` rather than ControlledGate, i.e.,

```
C(X, 2) == Toffoli
```

`__init__`(*gate*, *n=1*)
Initialize a ControlledGate object.

Parameters

- **`gate`** – Gate to wrap.
- **`n`** (*`int`*) – Number of control qubits.

`get_inverse`()
Return inverse of a controlled gate, which is the controlled inverse gate.

class `projectq.ops.`**`DaggeredGate`**(*gate*)
Wrapper class allowing to execute the inverse of a gate, even when it does not define one.

If there is a replacement available, then there is also one for the inverse, namely the replacement function run in reverse, while inverting all gates. This class enables using this emulation automatically.

A DaggeredGate is returned automatically when employing the get_inverse- function on a gate which does not provide a get_inverse() member function.

Example

```
with Dagger(eng):
        MySpecialGate | qubits
```

will create a DaggeredGate if MySpecialGate does not implement get_inverse. If there is a decomposition function available, an auto-replacer engine can automatically replace the inverted gate by a call to the decomposition function inside a "with Dagger"-statement.

`__init__`(*gate*)
Initialize a DaggeredGate representing the inverse of the gate 'gate'.

Parameters **`gate`** – Any gate object of which to represent the inverse.

`get_inverse`()
Return the inverse gate (the inverse of the inverse of a gate is the gate itself).

class `projectq.ops.`**`DeallocateQubitGate`**
Qubit deallocation gate class

class `projectq.ops.`**`EntangleGate`**
Entangle gate (Hadamard on first qubit, followed by CNOTs applied to all other qubits).

class `projectq.ops.`**`FastForwardingGate`**
Base class for classical instruction gates which require a fast-forward through compiler engines that cache /

buffer gates. Examples include Measure and Deallocate, which both should be executed asap, such that Measurement results are available and resources are freed, respectively.

Note: The only requirement is that FlushGate commands run the entire circuit. FastForwardingGate objects can be used but the user cannot expect a measurement result to be available for all back-ends when calling only Measure. E.g., for the IBM Quantum Experience back-end, sending the circuit for each Measure-gate would be too inefficient, which is way a final

is required before the circuit gets sent through the API.

---

class `projectq.ops.`**`FlushGate`**

Flush gate (denotes the end of the circuit).

Note: All compiler engines (cengines) which cache/buffer gates are obligated to flush and send all gates to the next compiler engine (followed by the flush command).

Note: This gate is sent when calling

```
eng.flush()
```

on the MainEngine *eng*.

---

class `projectq.ops.`**`HGate`**

Hadamard gate class

class `projectq.ops.`**`MeasureGate`**

Measurement gate class

exception `projectq.ops.`**`NotInvertible`**

Exception thrown when trying to invert a gate which is not invertable (or where the inverse is not implemented (yet)).

exception `projectq.ops.`**`NotMergeable`**

Exception thrown when trying to merge two gates which are not mergeable (or where it is not implemented (yet)).

class `projectq.ops.`**`Ph`**(*angle*)

Phase gate (global phase)

class `projectq.ops.`**`R`**(*angle*)

Phase-shift gate (equivalent to Rz up to a global phase)

class `projectq.ops.`**`Rx`**(*angle*)

RotationX gate class

class `projectq.ops.`**`Ry`**(*angle*)

RotationX gate class

class `projectq.ops.`**`Rz`**(*angle*)

RotationZ gate class

class `projectq.ops.`**`SGate`**

S gate class

class `projectq.ops.`**`SelfInverseGate`**

Self-inverse basic gate class.

Automatic implementation of the get_inverse-member function for self-inverse gates.

Example

```
get_inverse(H) | qubit # get_inverse(H) == H; it is a self-inverse gate
```

class `projectq.ops.`**`SwapGate`**

Swap gate class (swaps 2 qubits)

class `projectq.ops.`**`TGate`**

T gate class

class `projectq.ops.`**`Tensor`**`(`*gate*`)`

Wrapper class allowing to apply a (single-qubit) gate to every qubit in a quantum register. Allowed syntax is to supply either a qureg or a tuple which contains only one qureg.

Example

```
Tensor(H) | x # applies H to every qubit in the list of qubits x
Tensor(H) | (x,) # alternative to be consistent with other syntax
```

`__init__``(`*gate*`)`

Initialize a Tensor object for the gate.

`get_inverse``()`

Return the inverse of this tensored gate (which is the tensored inverse of the gate).

class `projectq.ops.`**`XGate`**

Pauli-X gate class

class `projectq.ops.`**`YGate`**

Pauli-Y gate class

class `projectq.ops.`**`ZGate`**

Pauli-Z gate class

`projectq.ops.`**`apply_command`**`(`*cmd*`)`

Apply a command.

Extracts the qubits-owning (target) engine from the Command object and sends the Command to it.

Parameters **`cmd`** (`Command`) – Command to apply

`projectq.ops.`**`get_inverse`**`(`*gate*`)`

Return the inverse of a gate.

Tries to call gate.get_inverse and, upon failure, creates a DaggeredGate instead.

Parameters **`gate`** – Gate of which to get the inverse

Example

```
get_inverse(H) # returns a Hadamard gate (HGate object)
```

setups

The setups package contains a collection of setups which can be loaded using an import statement. Each setup then loads its own set of decomposition rules and default compiler engines.

Subpackages

decompositions

The decomposition package is a collection of gate decomposition / replacement rules which can be used by, e.g., the AutoReplacer engine.

Submodules

projectq.setups.decompositions.crz2cxandrz module

Registers a decomposition for controlled z-rotation gates.

It uses 2 z-rotations and 2 C^n NOT gates to achieve this gate.

projectq.setups.decompositions.entangle module

Registers a decomposition for the Entangle gate.

Applies a Hadamard gate to the first qubit and then, conditioned on this first qubit, CNOT gates to all others.

projectq.setups.decompositions.globalphase module

Registers a decomposition rule for global phases.

Deletes global phase gates (which can be ignored).

projectq.setups.decompositions.ph2r module

Registers a decomposition for the controlled global phase gate.

Turns the controlled global phase gate into a (controlled) phase-shift gate. Each time this rule is applied, one control can be shaved off.

projectq.setups.decompositions.qft2crandhadamard module

Registers a decomposition rule for the quantum Fourier transform.

Decomposes the QFT gate into Hadamard and controlled phase-shift gates (R).

Warning: The final Swaps are not included, as those are simply a re-indexing of quantum registers.

projectq.setups.decompositions.r2rzandph module

Registers a decomposition rule for the phase-shift gate.

Decomposes the (controlled) phase-shift gate using z-rotation and a global phase gate.

projectq.setups.decompositions.swap2cnot module

Registers a decomposition to achieve a Swap gate.

Decomposes a Swap gate using 3 CNOT gates, where the one in the middle features as many control qubits as the Swap gate has control qubits.

projectq.setups.decompositions.toffoli2cnotandtgate module

Registers a decomposition rule for the Toffoli gate.

Decomposes the Toffoli gate using Hadamard, T, Tdag, and CNOT gates.

Module contents

ibm

The IBM Setup imports the default IBM decomposition rules, compiler engines (featuring a CNOTMapper), and the IBM backend (IBMBackend).

Module contents

Registers a variety of useful gate decompositions, specifically for the IBM quantum experience backend. Among others it includes:

- Controlled z-rotations –> Controlled NOTs and single-qubit rotations
- Toffoli gate –> CNOT and single-qubit gates
- m-Controlled global phases –> (m-1)-controlled phase-shifts
- Global phases –> ignore
- (controlled) Swap gates –> CNOTs and Toffolis

class `projectq.setups.ibm.`**`AutoReplacer`**(*decomposition_chooser=<function AutoReplacer.<lambda>>*)

The AutoReplacer is a compiler engine which uses engine.is_available in order to determine which commands need to be replaced/decomposed/compiled further. The loaded setup is used to find decomposition rules appropriate for each command (e.g., setups.default).

`__init__`(*decomposition_chooser=<function AutoReplacer.<lambda>>*)

Initialize an AutoReplacer.

Parameters **`decomposition_chooser`** (*`function`*) – A function which, given the Command to decompose and a list of potential Decomposition objects, determines (and then returns) the 'best' decomposition.

The default decomposition chooser simply returns the first list element, i.e., calling

```
repl = AutoReplacer()
```

Amounts to

```
def decomposition_chooser(cmd, decomp_list):
        return decomp_list[0]
repl = AutoReplacer(decomposition_chooser)
```

receive (*command_list*)

Receive a list of commands from the previous compiler engine and, if necessary, replace/decompose the gates according to the decomposition rules in the loaded setup.

Parameters **command_list** (*list<Command>*) – List of commands to handle.

class projectq.setups.ibm.**IBMBackend** (*use_hardware=False*, *num_runs=1024*, *verbose=False*, *user=None*, *password=None*)

The IBM Backend class, which stores the circuit, transforms it to JSON QASM, and sends the circuit through the IBM API.

__init__ (*use_hardware=False*, *num_runs=1024*, *verbose=False*, *user=None*, *password=None*)

Initialize the Backend object.

Parameters

- **use_hardware** (*bool*) – If True, the code is run on the IBM quantum chip (instead of using the IBM simulator)
- **num_runs** (*int*) – Number of runs to collect statistics. (default is 1024)
- **verbose** (*bool*) – If True, statistics are printed, in addition to the measurement result being registered (at the end of the circuit).
- **user** (*string*) – IBM Quantum Experience user name
- **password** (*string*) – IBM Quantum Experience password

get_probabilities (*qureg*)

Return the list of basis states with corresponding probabilities.

The measured bits are ordered according to the supplied quantum register, i.e., the left-most bit in the state-string corresponds to the first qubit in the supplied quantum register.

Warning: Only call this function after the circuit has been executed!

Parameters **qureg** (*list<Qubit>*) – Quantum register determining the order of the qubits.

Returns Dictionary mapping n-bit strings to probabilities.

Return type probability_dict (dict)

Raises Exception – If no data is available (i.e., if the circuit has not been executed).

is_available (*cmd*)

Return true if the command can be executed.

The IBM quantum chip can do X, Y, Z, T, Tdag, S, Sdag, and CX / CNOT.

Parameters **cmd** (Command) – Command for which to check availability

receive (*command_list*)

Receives a command list and, for each command, stores it until completion.

Parameters **`command_list`** – List of commands to execute

class `projectq.setups.ibm.`**`IBMCNOTMapper`**

CNOT mapper for the IBM backend.

Transforms CNOTs such that all CNOTs within the circuit have the same target qubit (required by IBM backend). If necessary, it will flip around the CNOT gate by first applying Hadamard gates to both qubits, then CNOT with swapped control and target qubit, and finally Hadamard gates to both qubits.

Note: The mapper has to be run once on the entire circuit. Else, an Exception will be raised (if, e.g., several measurements are performed without re- initializing the mapper).

Warning: If the provided circuit cannot be mapped to the hardware layout without performing Swaps, the mapping procedure **raises an Exception**.

`__init__`()

Initialize an IBM CNOT Mapper compiler engine.

Resets the mapping.

`is_available`(*cmd*)

Check if the IBM backend can perform the Command cmd and return True if so.

Parameters **`cmd`** (`Command`) – The command to check

`receive`(*command_list*)

Receives a command list and, for each command, stores it until completion.

Parameters **`command_list`** (*`list of Command objects`*) – list of commands to receive.

Raises `Exception` – If mapping the CNOT gates to 1 qubit would require Swaps. The current version only supports remapping of CNOT gates without performing any Swaps due to the large costs associated with Swapping given the CNOT constraints.

class `projectq.setups.ibm.`**`LocalOptimizer`**(*m=5*)

LocalOptimizer is a compiler engine which optimizes locally (merging rotations, cancelling gates with their inverse) in a local window of user- defined size.

It stores all commands in a list of lists, where each qubit has its own gate pipeline. After adding a gate, it tries to merge / cancel successive gates using the get_merged and get_inverse functions of the gate (if available). For examples, see BasicRotationGate. Once a list corresponding to a qubit contains >=m gates, the pipeline is sent on to the next engine.

`__init__`(*m=5*)

Initialize a LocalOptimizer object.

Parameters **`m`** (*`int`*) – Number of gates to cache per qubit, before sending on the first gate.

`receive`(*command_list*)

Receive commands from the previous engine and cache them. If a flush gate arrives, the entire buffer is sent on.

class `projectq.setups.ibm.`**`TagRemover`**(*tags=[<class 'projectq.meta._compute.ComputeTag'>, <class 'projectq.meta._compute.UncomputeTag'>]*)

TagRemover is a compiler engine which removes temporary command tags (see the tag classes such as LoopTag in projectq.meta._loop).

Removing tags is important (after having handled them if necessary) in order to enable optimizations across meta-function boundaries (compute/action/ uncompute or loops after unrolling)

`__init__`(*tags=[<class 'projectq.meta._compute.ComputeTag'>, <class 'projectq.meta._compute.UncomputeTag'>]*)
Construct the TagRemover.

Parameters **`tags`** – A list of meta tag classes (e.g., [ComputeTag, UncomputeTag]) denoting the tags to remove

`receive`(*command_list*)
Receive a list of commands from the previous engine, remove all tags which are an instance of at least one of the meta tags provided in the constructor, and then send them on to the next compiler engine.

Parameters **`command_list`** (*`list<Command>`*) – List of commands to receive and then (after removing tags) send on.

Submodules

default module

The default module features a standard set of decomposition rules, compiler engines, and the simulator backend. Registers a variety of useful gate decompositions. Among others it includes

- Controlled z-rotations –> Controlled NOTs and single-qubit rotations
- Toffoli gate –> CNOT and single-qubit gates
- m-Controlled global phases –> (m-1)-controlled phase-shifts
- Global phases –> ignore
- (controlled) Swap gates –> CNOTs and Toffolis

Module contents

types

The types package contains quantum types such as Qubit, Qureg, and WeakQubitRef. With further development of the math library, also quantum integers, quantum fixed point numbers etc. will be added.

Module contents

class `projectq.types.`**`BasicQubit`**(*engine*, *idx*)
BasicQubit objects represent qubits.

They have an id and a reference to the owning engine.

`__bool__`()
Access the result of a previous measurement and return False / True (0/1)

`__eq__`(*other*)
Compare with other qubit (Returns True if equal id and engine).

Parameters **`other`** (`BasicQubit`) – BasicQubit to which to compare this one

`__hash__`()
Return the hash of this qubit.

Hash definition because of custom __eq__. Enables storing a qubit in, e.g., a set.

`__init__`(*engine*, *idx*)
Initialize a BasicQubit object.

Parameters

- **`engine`** – Owning engine / engine that created the qubit
- **`idx`** – Unique index of the qubit referenced by this qubit

`__int__`()
Access the result of a previous measurement and return as integer (0 / 1).

`__nonzero__`()
Access the result of a previous measurement for Python 2.7.

`__str__`()
Return string representation of this qubit.

`__weakref__`
list of weak references to the object (if defined)

class `projectq.types.`**`Qubit`**(*engine*, *idx*)
Qubit class.

Represents a (logical-level) qubit with a unique index provided by the MainEngine. Once the qubit goes out of scope (and is garbage-collected), it deallocates itself automatically, allowing automatic resource management.

Thus the qubit is not copyable; only returns a reference to the same object.

`__copy__`()
Non-copyable (returns reference to self).

Note: To prevent problems with automatic deallocation, qubits are not copyable!

`__deepcopy__`(*memo*)
Non-deepcopyable (returns reference to self).

Note: To prevent problems with automatic deallocation, qubits are not deepcopyable!

`__del__`()
Destroy the qubit and deallocate it (automatically).

class `projectq.types.`**`Qureg`**
Quantum register class.

Simplifies accessing measured values for single-qubit registers (no []- access necessary) and enables pretty-printing of general quantum registers (call Qureg.__str__(qureg)).

`__bool__`()
Return measured value if Qureg consists of 1 qubit only.

Raises

- Exception if more than 1 qubit resides in this register (then you
- need to specify which value to get using qureg[???])

`__int__``()`
Return measured value if Qureg consists of 1 qubit only.

Raises

- Exception if more than 1 qubit resides in this register (then you
- need to specify which value to get using qureg[???])

`__nonzero__``()`
Return measured value if Qureg consists of 1 qubit only for Python 2.7.

Raises

- Exception if more than 1 qubit resides in this register (then you
- need to specify which value to get using qureg[???])

`__str__``()`
Get string representation of a quantum register.

`__weakref__`
list of weak references to the object (if defined)

`engine`
Return owning engine.

class `projectq.types.`**`WeakQubitRef`**`(`*engine*, *idx*`)`
WeakQubitRef objects are used inside the Command object.

Qubits feature automatic deallocation when destroyed. WeakQubitRefs, on the other hand, do not share this feature, allowing to copy them and pass them along the compiler pipeline, while the actual qubit objects may be garbage- collected (and, thus, cleaned up early). Otherwise there is no difference between a WeakQubitRef and a Qubit object.

Python Module Index

p

projectq.backends, 13
projectq.cengines, 18
projectq.libs, 28
projectq.libs.math, 26
projectq.meta, 28
projectq.ops, 32
projectq.setups, 44
projectq.setups.decompositions, 41
projectq.setups.decompositions.crz2cxandrz, 40
projectq.setups.decompositions.entangle, 40
projectq.setups.decompositions.globalphase, 40
projectq.setups.decompositions.ph2r, 40
projectq.setups.decompositions.qft2crandhadamard, 40
projectq.setups.decompositions.r2rzandph, 41
projectq.setups.decompositions.swap2cnot, 41
projectq.setups.decompositions.toffoli2cnotandtgate, 41
projectq.setups.default, 44
projectq.setups.ibm, 41
projectq.types, 44

Index

Symbols

__bool__() (projectq.types.BasicQubit method), 44
__bool__() (projectq.types.Qureg method), 45
__copy__() (projectq.types.Qubit method), 45
__deepcopy__() (projectq.types.Qubit method), 45
__del__() (projectq.types.Qubit method), 45
__eq__() (projectq.types.BasicQubit method), 44
__hash__() (projectq.types.BasicQubit method), 44
__init__() (projectq.backends.CircuitDrawer method), 14
__init__() (projectq.backends.CommandPrinter method), 15
__init__() (projectq.backends.IBMBackend method), 16
__init__() (projectq.backends.ResourceCounter method), 17
__init__() (projectq.backends.Simulator method), 17
__init__() (projectq.cengines.AutoReplacer method), 18
__init__() (projectq.cengines.BasicEngine method), 19
__init__() (projectq.cengines.CommandModifier method), 20
__init__() (projectq.cengines.Decomposition method), 20
__init__() (projectq.cengines.DummyEngine method), 21
__init__() (projectq.cengines.ForwarderEngine method), 21
__init__() (projectq.cengines.IBMCNOTMapper method), 22
__init__() (projectq.cengines.InstructionFilter method), 22
__init__() (projectq.cengines.LocalOptimizer method), 23
__init__() (projectq.cengines.MainEngine method), 23
__init__() (projectq.cengines.TagRemover method), 25
__init__() (projectq.libs.math.AddConstant method), 26
__init__() (projectq.libs.math.AddConstantModN method), 26
__init__() (projectq.libs.math.MultiplyByConstantModN method), 27
__init__() (projectq.meta.Compute method), 29
__init__() (projectq.meta.Control method), 29
__init__() (projectq.meta.CustomUncompute method), 29
__init__() (projectq.meta.Dagger method), 30
__init__() (projectq.meta.Loop method), 31
__init__() (projectq.meta.LoopTag method), 31
__init__() (projectq.ops.BasicGate method), 32
__init__() (projectq.ops.BasicMathGate method), 33
__init__() (projectq.ops.BasicRotationGate method), 34
__init__() (projectq.ops.Command method), 35
__init__() (projectq.ops.ControlledGate method), 37
__init__() (projectq.ops.DaggeredGate method), 37
__init__() (projectq.ops.Tensor method), 39
__init__() (projectq.setups.ibm.AutoReplacer method), 41
__init__() (projectq.setups.ibm.IBMBackend method), 42
__init__() (projectq.setups.ibm.IBMCNOTMapper method), 43
__init__() (projectq.setups.ibm.LocalOptimizer method), 43
__init__() (projectq.setups.ibm.TagRemover method), 44
__init__() (projectq.types.BasicQubit method), 45
__int__() (projectq.types.BasicQubit method), 45
__int__() (projectq.types.Qureg method), 45
__nonzero__() (projectq.types.BasicQubit method), 45
__nonzero__() (projectq.types.Qureg method), 46
__str__() (projectq.types.BasicQubit method), 45
__str__() (projectq.types.Qureg method), 46
__weakref__ (projectq.types.BasicQubit attribute), 45
__weakref__ (projectq.types.Qureg attribute), 46

A

active_qubits (projectq.cengines.MainEngine attribute), 23
add_control_qubits() (projectq.ops.Command method), 35
AddConstant (class in projectq.libs.math), 26
AddConstantModN (class in projectq.libs.math), 26
All (in module projectq.ops), 32
all_qubits (projectq.ops.Command attribute), 35, 36
allocate_qubit() (projectq.cengines.BasicEngine method), 19

allocate_qureg() (projectq.cengines.BasicEngine method), 19
AllocateDirtyQubitGate (class in projectq.ops), 32
AllocateQubitGate (class in projectq.ops), 32
apply_command() (in module projectq.ops), 39
AutoReplacer (class in projectq.cengines), 18
AutoReplacer (class in projectq.setups.ibm), 41

B

backend (projectq.cengines.MainEngine attribute), 23
BasicEngine (class in projectq.cengines), 19
BasicGate (class in projectq.ops), 32
BasicMathGate (class in projectq.ops), 33
BasicQubit (class in projectq.types), 44
BasicRotationGate (class in projectq.ops), 34

C

C() (in module projectq.ops), 34
cheat() (projectq.backends.Simulator method), 18
CircuitDrawer (class in projectq.backends), 13
ClassicalInstructionGate (class in projectq.ops), 35
Command (class in projectq.ops), 35
CommandModifier (class in projectq.cengines), 20
CommandPrinter (class in projectq.backends), 15
CompareEngine (class in projectq.cengines), 20
Compute (class in projectq.meta), 28
ComputeTag (class in projectq.meta), 29
Control (class in projectq.meta), 29
control_qubits (projectq.ops.Command attribute), 35, 36
ControlledGate (class in projectq.ops), 36
CustomUncompute (class in projectq.meta), 29

D

Dagger (class in projectq.meta), 29
DaggeredGate (class in projectq.ops), 37
deallocate_qubit() (projectq.cengines.BasicEngine method), 19
DeallocateQubitGate (class in projectq.ops), 37
Decomposition (class in projectq.cengines), 20
dirty_qubits (projectq.cengines.MainEngine attribute), 23
DirtyQubitTag (class in projectq.meta), 30
DummyEngine (class in projectq.cengines), 21

E

engine (projectq.ops.Command attribute), 35, 36
engine (projectq.types.Qureg attribute), 46
EntangleGate (class in projectq.ops), 37

F

FastForwardingGate (class in projectq.ops), 37
flush() (projectq.cengines.MainEngine method), 24
FlushGate (class in projectq.ops), 38
ForwarderEngine (class in projectq.cengines), 21

G

gate (projectq.ops.Command attribute), 35
gate_counts (projectq.backends.ResourceCounter attribute), 17
generate_command() (projectq.ops.BasicGate method), 32
get_control_count() (in module projectq.meta), 31
get_inverse() (in module projectq.ops), 39
get_inverse() (projectq.libs.math.AddConstant method), 26
get_inverse() (projectq.libs.math.AddConstantModN method), 26
get_inverse() (projectq.ops.BasicGate method), 32
get_inverse() (projectq.ops.BasicRotationGate method), 34
get_inverse() (projectq.ops.Command method), 36
get_inverse() (projectq.ops.ControlledGate method), 37
get_inverse() (projectq.ops.DaggeredGate method), 37
get_inverse() (projectq.ops.Tensor method), 39
get_inverse_decomposition() (projectq.cengines.Decomposition method), 21
get_latex() (projectq.backends.CircuitDrawer method), 15
get_measurement_result() (projectq.cengines.MainEngine method), 24
get_merged() (projectq.ops.BasicGate method), 32
get_merged() (projectq.ops.BasicRotationGate method), 34
get_merged() (projectq.ops.Command method), 36
get_new_qubit_id() (projectq.cengines.MainEngine method), 24
get_probabilities() (projectq.backends.IBMBackend method), 16
get_probabilities() (projectq.setups.ibm.IBMBackend method), 42

H

HGate (class in projectq.ops), 38

I

IBMBackend (class in projectq.backends), 16
IBMBackend (class in projectq.setups.ibm), 42
IBMCNOTMapper (class in projectq.cengines), 22
IBMCNOTMapper (class in projectq.setups.ibm), 43
InstructionFilter (class in projectq.cengines), 22
interchangeable_qubit_indices (projectq.ops.Command attribute), 36
is_available() (projectq.backends.CircuitDrawer method), 15
is_available() (projectq.backends.CommandPrinter method), 15
is_available() (projectq.backends.IBMBackend method), 16

is_available() (projectq.backends.ResourceCounter method), 17
is_available() (projectq.backends.Simulator method), 18
is_available() (projectq.cengines.BasicEngine method), 19
is_available() (projectq.cengines.IBMCNOTMapper method), 22
is_available() (projectq.cengines.InstructionFilter method), 22
is_available() (projectq.setups.ibm.IBMBackend method), 42
is_available() (projectq.setups.ibm.IBMCNOTMapper method), 43
is_last_engine (projectq.cengines.BasicEngine attribute), 19
is_meta_tag_supported() (projectq.cengines.BasicEngine method), 20

L

LastEngineException, 22
LocalOptimizer (class in projectq.cengines), 23
LocalOptimizer (class in projectq.setups.ibm), 43
Loop (class in projectq.meta), 30
loop_tag_id (projectq.meta.LoopTag attribute), 31
LoopTag (class in projectq.meta), 31

M

main_engine (projectq.cengines.BasicEngine attribute), 19
main_engine (projectq.cengines.MainEngine attribute), 23
MainEngine (class in projectq.cengines), 23
make_tuple_of_qureg() (projectq.ops.BasicGate static method), 32
max_width (projectq.backends.ResourceCounter attribute), 17
MeasureGate (class in projectq.ops), 38
MultiplyByConstantModN (class in projectq.libs.math), 26

N

next_engine (projectq.cengines.BasicEngine attribute), 19
next_engine (projectq.cengines.MainEngine attribute), 23
NotInvertible, 38
NotMergeable, 38

P

Ph (class in projectq.ops), 38
projectq.backends (module), 13
projectq.cengines (module), 18
projectq.libs (module), 28
projectq.libs.math (module), 26
projectq.meta (module), 28
projectq.ops (module), 32
projectq.setups (module), 44
projectq.setups.decompositions (module), 41
projectq.setups.decompositions.crz2cxandrz (module), 40
projectq.setups.decompositions.entangle (module), 40
projectq.setups.decompositions.globalphase (module), 40
projectq.setups.decompositions.ph2r (module), 40
projectq.setups.decompositions.qft2crandhadamard (module), 40
projectq.setups.decompositions.r2rzandph (module), 41
projectq.setups.decompositions.swap2cnot (module), 41
projectq.setups.decompositions.toffoli2cnotandtgate (module), 41
projectq.setups.default (module), 44
projectq.setups.ibm (module), 41
projectq.types (module), 44

Q

Qubit (class in projectq.types), 45
qubits (projectq.ops.Command attribute), 35
Qureg (class in projectq.types), 45

R

R (class in projectq.ops), 38
receive() (projectq.backends.CircuitDrawer method), 15
receive() (projectq.backends.CommandPrinter method), 16
receive() (projectq.backends.IBMBackend method), 16
receive() (projectq.backends.ResourceCounter method), 17
receive() (projectq.backends.Simulator method), 18
receive() (projectq.cengines.AutoReplacer method), 18
receive() (projectq.cengines.CommandModifier method), 20
receive() (projectq.cengines.ForwarderEngine method), 21
receive() (projectq.cengines.IBMCNOTMapper method), 22
receive() (projectq.cengines.InstructionFilter method), 22
receive() (projectq.cengines.LocalOptimizer method), 23
receive() (projectq.cengines.MainEngine method), 24
receive() (projectq.cengines.TagRemover method), 25
receive() (projectq.setups.ibm.AutoReplacer method), 42
receive() (projectq.setups.ibm.IBMBackend method), 42
receive() (projectq.setups.ibm.IBMCNOTMapper method), 43
receive() (projectq.setups.ibm.LocalOptimizer method), 43
receive() (projectq.setups.ibm.TagRemover method), 44
register_decomposition() (in module projectq.cengines), 25
ResourceCounter (class in projectq.backends), 17
Rx (class in projectq.ops), 38

Ry (class in projectq.ops), 38
Rz (class in projectq.ops), 38

S

SelfInverseGate (class in projectq.ops), 38
send() (projectq.cengines.BasicEngine method), 20
set_measurement_result() (projectq.cengines.MainEngine method), 24
set_qubit_locations() (projectq.backends.CircuitDrawer method), 15
SGate (class in projectq.ops), 38
Simulator (class in projectq.backends), 17
SubConstant() (in module projectq.libs.math), 27
SubConstantModN() (in module projectq.libs.math), 27
SwapGate (class in projectq.ops), 39

T

TagRemover (class in projectq.cengines), 24
TagRemover (class in projectq.setups.ibm), 43
tags (projectq.ops.Command attribute), 35
Tensor (class in projectq.ops), 39
tex_str() (projectq.ops.BasicRotationGate method), 34
TGate (class in projectq.ops), 39

U

Uncompute() (in module projectq.meta), 31
UncomputeTag (class in projectq.meta), 31

W

WeakQubitRef (class in projectq.types), 46

X

XGate (class in projectq.ops), 39

Y

YGate (class in projectq.ops), 39

Z

ZGate (class in projectq.ops), 39