projectq Documentation

Release 0.4.1

a

Sep 18, 2018

Contents

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" Quantum 2, 49 (2018) (published on arXiv on 23 Dec 2016)
- Thomas Häner, Damian S. Steiger, Krysta M. Svore, and Matthias Troyer "A Software Methodology for Compiling Quantum Programs" Quantum Sci. Technol. 3 (2018) 020501 (published on arXiv on 5 Apr 2016)

Contents

CHAPTER 1

Tutorial

1.1 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 .
```

ProjectQ comes with a high-performance quantum simulator written in C++. Please see the detailed OS specific installation instructions below to make sure that you are installing the fastest version.

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**!

Note: Please use pip version v6.1.0 or higher as this ensures that dependencies are installed in the correct order.

Note: ProjectQ should be installed on each computer individually as the C++ simulator compilation creates binaries which are optimized for the specific hardware on which it is being installed (potentially using our AVX version and *-march=native*). Therefore, sharing the same ProjectQ installation across different hardware can cause problems.

1.2 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:

In order to install the fast C++ simulator, we require that your system has a C++ compiler (see option 3 below on how to only install the slower Python simulator via the *–without-cppsimulator* parameter)

Below you will find two options to install the fast C++ simulator. The first one is the easiest and requires only the standard compiler which Apple distributes with XCode. The second option uses macports to install the simulator with additional support for multi-threading by using OpenMP, which makes it slightly faster. We show how to install the required C++ compiler (clang) which supports OpenMP and additionally, we show how to install a newer python version.

Note: Depending on your system you might need to use *sudo* for the installation.

1. Installation using XCode and the default python:

Install XCode by opening a terminal and running the following command:

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

Next, you will need to install Python and pip. See option 2 for information on how to install a newer python version with macports. Here, we are using the standard python which is preinstalled with macOS. Pip can be installed by:

```
sudo easy_install pip
```

Now, you can install ProjectQ with the C++ simulator using the standard command:

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

2. Installation using macports:

   Either use the standard python and install pip as shown in option 1 or better use macports to install a newer python version, e.g., Python 3.5 and the corresponding 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++. First, we will need to install a suitable compiler with support for **C++11**, OpenMP, and instrinsics. The best option is to install clang 3.9 also using macports (note: gcc installed via macports does not work)

   ```
   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
   ```

3. Installation with only the slow Python simulator:

   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

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

   which most likely will fail. Then, try again with the flag `--without-cppsimulator`:

   ```
   python -m pip install --user --global-option=--without-cppsimulator␣
   ↪projectq
   ```

1.3 The ProjectQ syntax

Our goal is to have an intuitive syntax in order to enable an easy learning curve. Therefore, ProjectQ features a lean syntax which is close to the mathematical notation used in physics.

For example, consider applying an x-rotation by an angle *theta* to a qubit. In ProjectQ, this looks as follows:

```
Rx(theta) | qubit
```

whereas the corresponding notation in physics would be

$R_x(\theta)\ |\text{qubit}\rangle$

Moreover, the |-operator separates the classical arguments (on the left) from the quantum arguments (on the right). Next, you will see a basic quantum program using this syntax. Further examples can be found in the docs (*Examples* in the panel on the left) and in the ProjectQ examples folder on GitHub.

1.4 Basic quantum program

To check out the ProjectQ syntax in action and to see whether the installation worked, try to run 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 of these example codes **and more** can be found on GitHub.

2.1 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
```

(continues on next page)

(continued from previous page)

```
$ 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:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/projectq/checkouts/
↪latest/examples/quantum_random_numbers.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/projectq/checkouts/
↪latest/examples/quantum_random_numbers_ibm.py
@@ -1,8 +1,11 @@
+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(),
+                 engine_list=projectq.setups.ibm.get_engine_list())

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

2.2 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()
```

(continues on next page)

(continued from previous page)

```
    H | b1
    CNOT | (b1, b2)

    return b1, b2
```

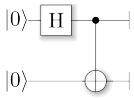
with a main compiler engine which has a CircuitDrawer back-end, i.e.,

```
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:

```
from projectq.ops import All, CNOT, H, Measure, Rz, X, Z
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
```

(continues on next page)

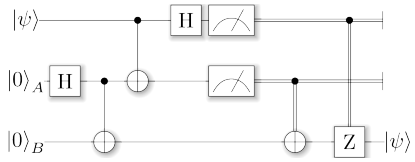
(continued from previous page)

```
    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
    Measure | b1
    msg_to_bob = [int(psi), int(b1)]
    if verbose:
        print("Alice is sending the message {} to Bob.".format(msg_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):



2.3 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):
            return True
        return False
    return eng.next_engine.is_available(cmd)
```

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:

3.1 backends

projectq.backends.CommandPrinter([...])	CommandPrinter is a compiler engine which prints commands to stdout prior to sending them on to the next compiler engine.
projectq.backends.CircuitDrawer([...])	CircuitDrawer is a compiler engine which generates TikZ code for drawing quantum circuits.
projectq.backends.Simulator([gate_fusion, ...])	Simulator is a compiler engine which simulates a quantum computer using C++-based kernels.
projectq.backends.ClassicalSimulator()	A simple introspective simulator that only permits classical operations.
projectq.backends.ResourceCounter()	ResourceCounter is a compiler engine which counts the number of gates and max.
projectq.backends.IBMBackend([use_hardware, ...])	The IBM Backend class, which stores the circuit, transforms it to JSON QASM, and sends the circuit through the IBM API.

3.1.1 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}) # swap 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 for 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.**ClassicalSimulator**

A simple introspective simulator that only permits classical operations.

Allows allocation, deallocation, measuring (no-op), flushing (no-op), controls, NOTs, and any BasicMathGate. Supports reading/writing directly from/to bits and registers of bits.

__init__()

Initialize the basic engine.

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

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.

read_bit(*qubit*)

Reads a bit.

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

Parameters **qubit** (projectq.types.Qubit) – The bit to read.

Returns 0 if the target bit is off, 1 if it's on.

Return type int

read_register(*qureg*)

Reads a group of bits as a little-endian integer.

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

Parameters **qureg** (projectq.types.Qureg) – The group of bits to read, in little-endian order.

Returns Little-endian register value.

Return type int

write_bit(*qubit*, *value*)

Resets/sets a bit to the given value.

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

Parameters

- **qubit** (projectq.types.Qubit) – The bit to write.

- **value** (*bool* | *int*) – Writes 1 if this value is truthy, else 0.

write_register(*qureg*, *value*)
Sets a group of bits to store a little-endian integer value.

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

Parameters

- **qureg** (projectq.types.Qureg) – The bits to write, in little-endian order.
- **value** (*int*) – The integer value to store. Must fit in the register.

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 default_measure.
- **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*, *device='ibmqx4'*, *retrieve_execution=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*, *device='ibmqx4'*, *retrieve_execution=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
- **device** (*string*) – Device to use ('ibmqx4', or 'ibmqx5') if use_hardware is set to True. Default is ibmqx4.
- **retrieve_execution** (*int*) – Job ID to retrieve instead of re- running the circuit (e.g., if previous run timed out).

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 RuntimeError – If no data is available (i.e., if the circuit has not been executed). Or if a qubit was supplied which was not present in the circuit (might have gotten optimized away).

is_available(*cmd*)

Return true if the command can be executed.

The IBM quantum chip can do X, Y, Z, T, Tdag, S, Sdag, rotation gates, barriers, 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 tuples of the form (cmd.gate, ctrl_cnt), where ctrl_cnt is the number of control qubits.

gate_class_counts
dict – Dictionary of gate class counts. The keys are tuples of the form (cmd.gate.__class__, ctrl_cnt), where ctrl_cnt is the number of control qubits.

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

Properties:

depth_of_dag (int): It is the longest path in the directed acyclic graph (DAG) of the program.

__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, increases the counters of the received 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, 4294967295) 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 gate matrices and then applies one 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.

---

apply_qubit_operator(*qubit_operator*, *qureg*)

Apply a (possibly non-unitary) qubit_operator to the current wave function represented by the supplied quantum register.

Parameters

- **qubit_operator** (projectq.ops.QubitOperator) – Operator to apply.
- **qureg** (*list[Qubit],Qureg*) – Quantum bits to which to apply the operator.

Raises Exception – If *qubit_operator* acts on more qubits than present in the *qureg* argument.

Warning: This function allows applying non-unitary gates and it will not re-normalize the wave function! It is for numerical experiments only and should not be used for other purposes.

Note: Make sure all previous commands (especially allocations) have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

---

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.

Note: Make sure all previous commands have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function DOES NOT automatically convert from logical qubits to mapped qubits.

---

collapse_wavefunction(*qureg*, *values*)

Collapse a quantum register onto a classical basis state.

Parameters

- **qureg** (*Qureg|list[Qubit]*) – Qubits to collapse.
- **values** (*list[bool|int]|string[0|1]*) – Measurement outcome for each of the qubits in *qureg*.

Raises RuntimeError – If an outcome has probability (approximately) 0 or if unknown qubits are provided (see note).

Note: Make sure all previous commands have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

get_amplitude(*bit_string*, *qureg*)

Return the probability amplitude of the supplied *bit_string*. The ordering is given by the quantum register *qureg*, which must contain all allocated qubits.

Parameters

- **bit_string** (*list[bool|int]|string[0|1]*) – Computational basis state
- **qureg** (*Qureg|list[Qubit]*) – Quantum register determining the ordering. Must contain all allocated qubits.

Returns Probability amplitude of the provided bit string.

Note: Make sure all previous commands (especially allocations) have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

get_expectation_value(*qubit_operator*, *qureg*)

Get the expectation value of qubit_operator w.r.t. the current wave function represented by the supplied quantum register.

Parameters

- **qubit_operator** (projectq.ops.QubitOperator) – Operator to measure.
- **qureg** (*list[Qubit],Qureg*) – Quantum bits to measure.

Returns Expectation value

Note: Make sure all previous commands (especially allocations) have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

Raises Exception – If *qubit_operator* acts on more qubits than present in the *qureg* argument.

get_probability(*bit_string*, *qureg*)

Return the probability of the outcome *bit_string* when measuring the quantum register *qureg*.

Parameters

- **bit_string** (*list[bool|int]|string[0|1]*) – Measurement outcome.
- **qureg** (*Qureg|list[Qubit]*) – Quantum register.

Returns Probability of measuring the provided bit string.

Note: Make sure all previous commands (especially allocations) have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

---

is_available(*cmd*)

Specialized implementation of is_available: The simulator can deal with all arbitrarily-controlled gates which provide a gate-matrix (via gate.matrix) and acts on 5 or less qubits (not counting the control qubits).

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.

set_wavefunction(*wavefunction*, *qureg*)

Set the wavefunction and the qubit ordering of the simulator.

The simulator will adopt the ordering of qureg (instead of reordering the wavefunction).

Parameters

- **wavefunction** (*list[complex]*) – Array of complex amplitudes describing the wavefunction (must be normalized).
- **qureg** (*Qureg|list[Qubit]*) – Quantum register determining the ordering. Must contain all allocated qubits.

Note: Make sure all previous commands (especially allocations) have passed through the compilation chain (call main_engine.flush() to make sure).

Note: If there is a mapper present in the compiler, this function automatically converts from logical qubits to mapped qubits for the qureg argument.

3.2 cengines

The ProjectQ compiler engines package.

projectq.cengines.AutoReplacer(...[, ...])	The AutoReplacer is a compiler engine which uses engine.is_available in order to determine which commands need to be replaced/decomposed/compiled further.
projectq.cengines.BasicEngine()	Basic compiler engine: All compiler engines are derived from this class.
projectq.cengines.BasicMapper	
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.
projectq.cengines.CompareEngine()	CompareEngine is an engine which saves all commands.
projectq.cengines.DecompositionRule(...[, ...])	A rule for breaking down specific gates into sequences of simpler gates.
projectq.cengines.DecompositionRuleSet([...])	A collection of indexed decomposition rules.
projectq.cengines.DummyEngine([save_commands])	DummyEngine used for testing.
projectq.cengines.ForwarderEngine(engine[, ...])	A ForwarderEngine is a trivial engine which forwards all commands to the next engine.
projectq.cengines.GridMapper(num_rows, ...)	Mapper to a 2-D grid graph.
projectq.cengines.InstructionFilter(filterfun)	The InstructionFilter is a compiler engine which changes the behavior of is_available according to a filter function.
projectq.cengines.IBM5QubitMapper()	Mapper for the 5-qubit IBM backend.
projectq.cengines.LinearMapper(num_qubits[, ...])	Maps a quantum circuit to a linear chain of nearest neighbour interactions.
projectq.cengines.LocalOptimizer([m])	LocalOptimizer is a compiler engine which optimizes locally (merging rotations, cancelling gates with their inverse) in a local window of user- defined size.
projectq.cengines.ManualMapper([map_fun])	Manual Mapper which adds QubitPlacementTags to Allocate gate commands according to a user-specified mapping.
projectq.cengines.MainEngine([backend, ...])	The MainEngine class provides all functionality of the main compiler engine.
projectq.cengines.SwapAndCNOTFlipper(...)	Flips CNOTs and translates Swaps to CNOTs where necessary.
projectq.cengines.TagRemover([tags])	TagRemover is a compiler engine which removes temporary command tags (see the tag classes such as LoopTag in projectq.meta._loop).

3.2.1 Module contents

class projectq.cengines.**AutoReplacer**(*decompositionRuleSet*, *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__(*decompositionRuleSet*, *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.

Raises ValueError – Qubit already deallocated. Caller likely has a bug.

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.**BasicMapperEngine**
Parent class for all Mappers.

self.**current_mapping**
dict – Keys are the logical qubit ids and values are the mapped qubit ids.

__init__()
Initialize the basic engine.

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

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.

__init__()

Initialize the basic engine.

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

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.

class projectq.cengines.**DecompositionRule**(*gate_class*, *gate_decomposer*, *gate_recognizer=<function DecompositionRule.<lambda>>*)

A rule for breaking down specific gates into sequences of simpler gates.

__init__(*gate_class*, *gate_decomposer*, *gate_recognizer=<function DecompositionRule.<lambda>>*)

Parameters

- **gate_class** (*type*) – The type of gate that this rule decomposes.

  The gate class is redundant information used to make lookups faster when iterating over a circuit and deciding "which rules apply to this gate?" again and again.

  Note that this parameter is a gate type, not a gate instance. You supply gate_class=MyGate or gate_class=MyGate().__class__, not gate_class=MyGate().

- **gate_decomposer** (*function[projectq.ops.Command]*) – 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.

- **(function[projectq.ops.Command]** (*gate_recognizer*) – boolean): A predicate that determines if the decomposition applies to the given command (on top of the filtering by gate_class).

  For example, a decomposition rule may only to apply rotation gates that rotate by a specific angle.

  If no gate_recognizer is given, the decomposition applies to all gates matching the gate_class.

class projectq.cengines.**DecompositionRuleSet**(*rules=None*, *modules=None*)
A collection of indexed decomposition rules.

__init__(*rules=None*, *modules=None*)

Parameters

- **list[DecompositionRule]** (*rules*) – Initial decomposition rules.
- **modules** (*iterable[ModuleWithDecompositionRuleSet]*) – A list of things with an "all_defined_decomposition_rules" property containing decomposition rules to add to the rule set.

add_decomposition_rule(*rule*)
Add a decomposition rule to the rule set.

Parameters **rule** (*DecompositionRuleGate*) – The decomposition rule to add.

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.

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.

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.**GridMapper**(*num_rows*, *num_columns*, *mapped_ids_to_backend_ids=None*, *storage=1000*, *optimization_function=<function GridMapper.<lambda>>*, *num_optimization_steps=50*)

Mapper to a 2-D grid graph.

Mapped qubits on the grid are numbered in row-major order. E.g. for 3 rows and 2 columns:

0 - 1 | | 2 - 3 | | 4 - 5

The numbers are the mapped qubit ids. The backend might number the qubits on the grid differently (e.g. not row-major), we call these backend qubit ids. If the backend qubit ids are not row-major, one can pass a dictionary translating from our row-major mapped ids to these backend ids.

Note: The algorithm sorts twice inside each column and once inside each row.

current_mapping
Stores the mapping: key is logical qubit id, value is backend qubit id.

storage
int – Number of gate it caches before mapping.

num_rows
int – Number of rows in the grid

num_columns
int – Number of columns in the grid

num_qubits
int – num_rows x num_columns = number of qubits

num_mappings
int – Number of times the mapper changed the mapping

depth_of_swaps
dict – Key are circuit depth of swaps, value is the number of such mappings which have been applied

num_of_swaps_per_mapping
dict – Key are the number of swaps per mapping, value is the number of such mappings which have been applied

__init__(*num_rows*, *num_columns*, *mapped_ids_to_backend_ids=None*, *storage=1000*, *optimization_function=<function GridMapper.<lambda>>*, *num_optimization_steps=50*)
Initialize a GridMapper compiler engine.

Parameters

- **num_rows** (*int*) – Number of rows in the grid
- **num_columns** (*int*) – Number of columns in the grid.
- **mapped_ids_to_backend_ids** (*dict*) – Stores a mapping from mapped ids which are 0,. . . ,self.num_qubits-1 in row-major order on the grid to the corresponding qubit ids of the backend. Key: mapped id. Value: corresponding backend id. Default is None which means backend ids are identical to mapped ids.
- **storage** – Number of gates to temporarily store
- **optimization_function** – Function which takes a list of swaps and returns a cost value. Mapper chooses a permutation which minimizes this cost. Default optimizes for circuit depth.
- **num_optimization_steps** (*int*) – Number of different permutations to of the matching to try and minimize the cost.

Raises `RuntimeError` – if incorrect *mapped_ids_to_backend_ids* parameter

is_available(*cmd*)
Only allows 1 or two qubit gates.

receive(*command_list*)
Receives a command list and, for each command, stores it until we do a mapping (FlushGate or Cache of stored commands is full).

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

return_swaps(*old_mapping*, *new_mapping*, *permutation=None*)
Returns the swap operation to change mapping

Parameters

- **old_mapping** – dict: keys are logical ids and values are mapped qubit ids
- **new_mapping** – dict: keys are logical ids and values are mapped qubit ids
- **permutation** – list of int from 0, 1, ..., self.num_rows-1. It is used to permute the found perfect matchings. Default is None which keeps the original order.

Returns List of tuples. Each tuple is a swap operation which needs to be applied. Tuple contains the two mapped qubit ids for the Swap.

class projectq.cengines.**IBM5QubitMapper**
Mapper for the 5-qubit IBM backend.

Maps a given circuit to the IBM Quantum Experience chip.

Note: The mapper has to be run once on the entire circuit.

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 5-qubit 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*)

Initializer: 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).

__init__(*engine*)

Initialize self. See help(type(self)) for accurate signature.

class projectq.cengines.**LinearMapper**(*num_qubits*, *cyclic=False*, *storage=1000*)

Maps a quantum circuit to a linear chain of nearest neighbour interactions.

Maps a quantum circuit to a linear chain of qubits with nearest neighbour interactions using Swap gates. It supports open or cyclic boundary conditions.

current_mapping

Stores the mapping: key is logical qubit id, value is mapped qubit id from 0,. . . ,self.num_qubits

cyclic

Bool – If chain is cyclic or not

storage

int – Number of gate it caches before mapping.

num_mappings

int – Number of times the mapper changed the mapping

depth_of_swaps

dict – Key are circuit depth of swaps, value is the number of such mappings which have been applied

num_of_swaps_per_mapping

dict – Key are the number of swaps per mapping, value is the number of such mappings which have been applied

---

Note:

1. Gates are cached and only mapped from time to time. A FastForwarding gate doesn't empty the cache, only a FlushGate does.
2. Only 1 and two qubit gates allowed.

3. Does not optimize for dirty qubits.

__init__(*num_qubits*, *cyclic=False*, *storage=1000*)
Initialize a LinearMapper compiler engine.

Parameters

- **num_qubits** (*int*) – Number of physical qubits in the linear chain
- **cyclic** (*bool*) – If 1D chain is a cycle. Default is False.
- **storage** (*int*) – Number of gates to temporarily store, default is 1000

is_available(*cmd*)
Only allows 1 or two qubit gates.

receive(*command_list*)
Receives a command list and, for each command, stores it until we do a mapping (FlushGate or Cache of stored commands is full).

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

static return_new_mapping(*num_qubits*, *cyclic*, *currently_allocated_ids*, *stored_commands*, *current_mapping*)
Builds a mapping of qubits to a linear chain.

It goes through stored_commands and tries to find a mapping to apply these gates on a first come first served basis. More compilicated scheme could try to optimize to apply as many gates as possible between the Swaps.

Parameters

- **num_qubits** (*int*) – Total number of qubits in the linear chain
- **cyclic** (*bool*) – If linear chain is a cycle.
- **currently_allocated_ids** (*set of int*) – Logical qubit ids for which the Allocate gate has already been processed and sent to the next engine but which are not yet deallocated and hence need to be included in the new mapping.
- **stored_commands** (*list of Command objects*) – Future commands which should be applied next.
- **current_mapping** – A current mapping as a dict. key is logical qubit id, value is placement id. If there are different possible maps, this current mapping is used to minimize the swaps to go to the new mapping by a heuristic.

Returns: A new mapping as a dict. key is logical qubit id, value is placement id

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 dict 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*, *verbose=False*)
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.

mapper
BasicMapperEngine – Access to the mapper if there is one.

__init__(*backend=None*, *engine_list=None*, *verbose=False*)
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 compiled circuit to.
- **engine_list** (*list<BasicEngine>*) – List of engines / backends to use as compiler engines. Note: The engine list must not contain multiple mappers (instances of BasicMapperEngine). Default: projectq.setups.default.get_engine_list()
- **verbose** (*bool*) – Either print full or compact error messages. Default: False (i.e. compact error messages).

Example

```
from projectq import MainEngine
eng = MainEngine() # uses default engine_list and the Simulator
```

Instead of the default *engine_list* one can use, e.g., one of the IBM setups which defines a custom *engine_list* useful for one of the IBM chips

Example

```
import projectq.setups.ibm as ibm_setup
from projectq import MainEngine
```

(continues on next page)

(continued from previous page)

```
eng = MainEngine(engine_list=ibm_setup.get_engine_list())
# eng uses the default Simulator backend
```

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

Example

```
from projectq.cengines import (TagRemover, AutoReplacer,
                               LocalOptimizer,
                               DecompositionRuleSet)
from projectq.backends import Simulator
from projectq import MainEngine
rule_set = DecompositionRuleSet()
engines = [AutoReplacer(rule_set), 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)

send(*command_list*)

Forward the list of commands to the next engine in the pipeline.

It also shortens exception stack traces if self.verbose is False.

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.**ManualMapper**(*map_fun=<function ManualMapper.<lambda>>*)
Manual Mapper which adds QubitPlacementTags to Allocate gate commands according to a user-specified mapping.

map
function – The function which maps a given qubit id to its location. It gets set when initializing the mapper.

__init__(*map_fun=<function ManualMapper.<lambda>>*)
Initialize the mapper to a given mapping. If no mapping function is provided, the qubit id is used as the location.

Parameters **map_fun** (*function*) – Function which, given the qubit id, returns an integer describing the physical location (must be constant).

receive(*command_list*)
Receives a command list and passes it to the next engine, adding qubit placement tags to allocate gates.

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

exception projectq.cengines.**NotYetMeasuredError**

class projectq.cengines.**SwapAndCNOTFlipper**(*connectivity*)
Flips CNOTs and translates Swaps to CNOTs where necessary.

Warning: This engine assumes that CNOT and Hadamard gates are supported by the following engines.

Warning: This engine cannot be used as a backend.

__init__(*connectivity*)
Initialize the engine.

Parameters **connectivity** (*set*) – Set of tuples (c, t) where if (c, t) is an element of the set means that a CNOT can be performed between the physical ids (c, t) with c being the control and t being the target qubit.

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 if the command is a CNOT gate, it flips it using Hadamard gates if necessary; if it is a Swap gate, it decomposes it using 3 CNOTs. All other gates are simply sent to the next engine.

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

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.

exception projectq.cengines.**UnsupportedEngineError**

projectq.cengines.**return_swap_depth**(*swaps*)

Returns the circuit depth to execute these swaps.

Parameters **swaps** (*list of tuples*) – Each tuple contains two integers representing the two IDs of the qubits involved in the Swap operation

Returns Circuit depth to execute these swaps.

3.3 libs

The library collection of ProjectQ which, for now, consists of a tiny math library and an interface library to RevKit. Soon, more libraries will be added.

3.3.1 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.

projectq.libs.math.all_defined_decomposition_rules	list() -> new empty list list(iterable) -> new list initialized from iterable's items
projectq.libs.math.AddConstant(a)	Add a constant to a quantum number represented by a quantum register, stored from low- to high-bit.
projectq.libs.math.SubConstant(a)	Subtract a constant from a quantum number represented by a quantum register, stored from low- to high-bit.
projectq.libs.math.AddConstantModN(a, N)	Add a constant to a quantum number represented by a quantum register modulo N.

Continued on next page

Table 3 – continued from previous page

projectq.libs.math.SubConstantModN(a, N)	Subtract a constant from a quantum number represented by a quantum register modulo N.
projectq.libs.math.MultiplyByConstantModN(a, N)	Multiply a quantum number represented by a quantum register by a constant modulo N.

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 2.
```

__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 -2 = 6 = 1 (mod 5)
```

revkit

This library integrates RevKit into ProjectQ to allow some automatic synthesis routines for reversible logic. The library adds the following operations that can be used to construct quantum circuits:

- *ControlFunctionOracle*: Synthesizes a reversible circuit from Boolean control function
- *PermutationOracle*: Synthesizes a reversible circuit for a permutation
- *PhaseOracle*: Synthesizes phase circuit from an arbitrary Boolean function

RevKit can be installed from PyPi with *pip install revkit*.

Note: The RevKit Python module must be installed in order to use this ProjectQ library.

There exist precompiled binaries in PyPi, as well as a source distribution. Note that a C++ compiler with C++17 support is required to build the RevKit python module from source. Examples for compatible compilers are Clang 6.0, GCC 7.3, and GCC 8.1.

The integration of RevKit into ProjectQ and other quantum programming languages is described in the paper

- Mathias Soeken, Thomas Haener, and Martin Roetteler "Programming Quantum Computers Using Design Automation," in: Design Automation and Test in Europe (2018) [arXiv:1803.01022]

Module contents

class projectq.libs.revkit.**ControlFunctionOracle**(*function*, ***kwargs*)

Synthesizes a negation controlled by an arbitrary control function.

This creates a circuit for a NOT gate which is controlled by an arbitrary Boolean control function. The control function is provided as integer representation of the function's truth table in binary notation. For example, for the majority-of-three function, which truth table 11101000, the value for function can be, e.g., `0b11101000`, `0xe8`, or `232`.

Example

This example creates a circuit that causes to invert qubit `d`, the majority-of-three function evaluates to true for the control qubits `a`, `b`, and `c`.

```
ControlFunctionOracle(0x8e) | ([a, b, c], d)
```

__init__(*function*, ***kwargs*)

Initializes a control function oracle.

Parameters **function** (*int*) – Function truth table.

Keyword Arguments **synth** – A RevKit synthesis command which creates a reversible circuit based on a truth table and requires no additional ancillae (e.g., `revkit.esopbs`). Can also be a nullary lambda that calls several RevKit commands. **Default:** `revkit.esopbs`

__or__(*qubits*)

Applies control function to qubits (and synthesizes circuit).

Parameters **qubits** (*tuple<Qureg>*) – Qubits to which the control function is being applied. The first *n* qubits are for the controls, the last qubit is for the target qubit.

class projectq.libs.revkit.**PermutationOracle**(*permutation*, ***kwargs*)

Synthesizes a permutation using RevKit.

Given a permutation over *2**q* elements (starting from 0), this class helps to automatically find a reversible circuit over *q* qubits that realizes that permutation.

Example

```
PermutationOracle([0, 2, 1, 3]) | (a, b)
```

__init__ (*permutation*, ***kwargs*)
Initializes a permutation oracle.

Parameters **permutation** (*list<int>*) – Permutation (starting from 0).

Keyword Arguments **synth** – A RevKit synthesis command which creates a reversible circuit based on a reversible truth table (e.g., `revkit.tbs` or `revkit.dbs`). Can also be a nullary lambda that calls several RevKit commands. **Default:** `revkit.tbs`

__or__ (*qubits*)
Applies permutation to qubits (and synthesizes circuit).

Parameters **qubits** (*tuple<Qureg>*) – Qubits to which the permutation is being applied.

class projectq.libs.revkit.**PhaseOracle** (*function*, ***kwargs*)
Synthesizes phase circuit from an arbitrary Boolean function.

This creates a phase circuit from a Boolean function. It inverts the phase of all amplitudes for which the function evaluates to 1. The Boolean function is provided as integer representation of the function's truth table in binary notation. For example, for the majority-of-three function, which truth table 11101000, the value for function can be, e.g., `0b11101000`, `0xe8`, or `232`.

Note that a phase circuit can only accurately be found for a normal function, i.e., a function that maps the input pattern 0, 0, . . . , 0 to 0. The circuits for a function and its inverse are the same.

Example

This example creates a phase circuit based on the majority-of-three function on qubits `a`, `b`, and `c`.

```
PhaseOracle(0x8e) | (a, b, c)
```

__init__ (*function*, ***kwargs*)
Initializes a phase oracle.

Parameters **function** (*int*) – Function truth table.

Keyword Arguments **synth** – A RevKit synthesis command which creates a reversible circuit based on a truth table and requires no additional ancillae (e.g., `revkit.esopps`). Can also be a nullary lambda that calls several RevKit commands. **Default:** `revkit.esopps`

__or__ (*qubits*)
Applies phase circuit to qubits (and synthesizes circuit).

Parameters **qubits** (*tuple<Qureg>*) – Qubits to which the phase circuit is being applied.

3.3.2 Module contents

3.4 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.

projectq.meta.DirtyQubitTag	Dirty qubit meta tag
projectq.meta.LogicalQubitIDTag(logical_qubit_id)	LogicalQubitIDTag for a mapped qubit to annotate a MeasureGate.
projectq.meta.LoopTag(num)	Loop meta tag
projectq.meta.Loop(engine, num)	Loop n times over an entire code block.
projectq.meta.Compute(engine)	Start a compute-section.
projectq.meta.Uncompute(engine)	Uncompute automatically.
projectq.meta.CustomUncompute(engine)	Start a custom uncompute-section.
projectq.meta.ComputeTag	Compute meta tag.
projectq.meta.UncomputeTag	Uncompute meta tag.
projectq.meta.Control(engine, qubits)	Condition an entire code block on the value of qubits being 1.
projectq.meta.get_control_count(cmd)	Return the number of control qubits of the command object cmd
projectq.meta.Dagger(engine)	Invert an entire code block.
projectq.meta.insert_engine(prev_engine, ...)	Inserts an engine into the singly-linked list of engines.
projectq.meta.drop_engine_after(prev_engine)	Removes an engine from the singly-linked list of engines.

3.4.1 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.**LogicalQubitIDTag**(*logical_qubit_id*)
LogicalQubitIDTag for a mapped qubit to annotate a MeasureGate.

logical_qubit_id
int – Logical qubit id

__init__(*logical_qubit_id*)
Initialize self. See help(type(self)) for accurate signature.

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
```

Raises

- TypeError – If number of iterations (num) is not an integer
- ValueError – If number of iterations (num) is not >= 0

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

__init__(*num*)
Initialize self. See help(type(self)) for accurate signature.

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.**drop_engine_after**(*prev_engine*)
Removes an engine from the singly-linked list of engines.

Parameters **prev_engine** (projectq.cengines.BasicEngine) – The engine just before the engine to drop.

Returns The dropped engine.

Return type Engine

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

projectq.meta.**insert_engine**(*prev_engine*, *engine_to_insert*)
Inserts an engine into the singly-linked list of engines.

It also sets the correct main_engine for engine_to_insert.

Parameters

- **prev_engine** (projectq.cengines.BasicEngine) – The engine just before the insertion point.
- **engine_to_insert** (projectq.cengines.BasicEngine) – The engine to insert at the insertion point.

3.5 ops

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

projectq.ops.BasicGate()	Base class of all gates.
projectq.ops.SelfInverseGate()	Self-inverse basic gate class.
projectq.ops.BasicRotationGate(angle)	Defines a base class of a rotation gate.
projectq.ops.BasicPhaseGate(angle)	Defines a base class of a phase gate.
projectq.ops.ClassicalInstructionGate()	Classical instruction gate.

Continued on next page

Table 5 – continued from previous page

projectq.ops.FastForwardingGate()	Base class for classical instruction gates which require a fast-forward through compiler engines that cache / buffer gates.
projectq.ops.BasicMathGate(math_fun)	Base class for all math gates.
projectq.ops.apply_command(cmd)	Apply a command.
projectq.ops.Command(engine, gate, qubits[, ...])	Class used as a container to store commands.
projectq.ops.H	Shortcut (instance of) *projectq.ops.HGate*
projectq.ops.X	Shortcut (instance of) *projectq.ops.XGate*
projectq.ops.Y	Shortcut (instance of) *projectq.ops.YGate*
projectq.ops.Z	Shortcut (instance of) *projectq.ops.ZGate*
projectq.ops.S	Shortcut (instance of) *projectq.ops.SGate*
projectq.ops.Sdag	Wrapper class allowing to execute the inverse of a gate, even when it does not define one.
projectq.ops.T	Shortcut (instance of) *projectq.ops.TGate*
projectq.ops.Tdag	Wrapper class allowing to execute the inverse of a gate, even when it does not define one.
projectq.ops.SqrtX	Shortcut (instance of) *projectq.ops.SqrtXGate*
projectq.ops.Swap	Shortcut (instance of) *projectq.ops.SwapGate*
projectq.ops.SqrtSwap	Shortcut (instance of) *projectq.ops.SqrtSwapGate*
projectq.ops.Entangle	Shortcut (instance of) *projectq.ops.EntangleGate*
projectq.ops.Ph(angle)	Phase gate (global phase)
projectq.ops.Rx(angle)	RotationX gate class
projectq.ops.Ry(angle)	RotationX gate class
projectq.ops.Rz(angle)	RotationZ gate class
projectq.ops.R(angle)	Phase-shift gate (equivalent to Rz up to a global phase)
projectq.ops.FlushGate()	Flush gate (denotes the end of the circuit).
projectq.ops.MeasureGate()	Measurement gate class (for single qubits).
projectq.ops.Allocate	Shortcut (instance of) *projectq.ops.AllocateQubitGate*
projectq.ops.Deallocate	Shortcut (instance of) *projectq.ops.DeallocateQubitGate*
projectq.ops.AllocateDirty	Shortcut (instance of) *projectq.ops.AllocateDirtyQubitGate*
projectq.ops.Barrier	Shortcut (instance of) *projectq.ops.BarrierGate*
projectq.ops.DaggeredGate(gate)	Wrapper class allowing to execute the inverse of a gate, even when it does not define one.
projectq.ops.ControlledGate(gate[, n])	Controlled version of a gate.
projectq.ops.C(gate[, n])	Return n-controlled version of the provided gate.
projectq.ops.All	Shortcut (instance of) *projectq.ops.Tensor*
projectq.ops.Tensor(gate)	Wrapper class allowing to apply a (single-qubit) gate to every qubit in a quantum register.
projectq.ops.QFT	Shortcut (instance of) *projectq.ops.QFTGate*
projectq.ops.QubitOperator([term, coefficient])	A sum of terms acting on qubits, e.g., 0.5 * 'X0 X5' + 0.3 * 'Z1 Z2'.
projectq.ops.CRz(angle)	Shortcut for C(Rz(angle), n=1).
projectq.ops.CNOT	Controlled version of a gate.
projectq.ops.CZ	Controlled version of a gate.

Continued on next page

Table 5 – continued from previous page

projectq.ops.Toffoli	Controlled version of a gate.
projectq.ops.TimeEvolution(time, hamiltonian)	Gate for time evolution under a Hamiltonian (QubitOperator object).
projectq.ops.UniformlyControlledRy(angles)	Uniformly controlled Ry gate as introduced in arXiv:quant-ph/0312218.
projectq.ops.UniformlyControlledRz(angles)	Uniformly controlled Rz gate as introduced in arXiv:quant-ph/0312218.
projectq.ops.StatePreparation(final_state)	Gate for transforming qubits in state |0> to any desired quantum state.

3.5.1 Module contents

projectq.ops.**All**
Shortcut (instance of) *projectq.ops.Tensor*

alias of projectq.ops._metagates.Tensor

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

get_inverse()
Return the inverse gate.

Standard implementation of get_inverse:

Raises *NotInvertible* – inverse is not implemented

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

get_inverse()
Return the inverse gate.

Standard implementation of get_inverse:

Raises *NotInvertible* – inverse is not implemented

class projectq.ops.**BarrierGate**
Barrier gate class

get_inverse()
Return the inverse gate.

Standard implementation of get_inverse:

Raises *NotInvertible* – inverse is not implemented

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]])
```

__or__(*qubits*)
Operator| overload which enables the syntax Gate | qubits.

Example

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).

generate_command(*qubits*)
Helper function to generate a command consisting of the gate and the qubits being acted upon.

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

Returns A Command object containing the gate and the 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
```

get_math_function(*qubits*)
Return the math function which corresponds to the action of this math gate, given the input to the gate (a tuple of quantum registers).

Parameters **qubits** (*tuple<Qureg>*) – Qubits to which the math gate is being applied.

Returns Python function describing the action of this gate. (See BasicMathGate.__init__ for an example).

Return type math_fun (function)

class projectq.ops.**BasicPhaseGate**(*angle*)
Defines a base class of a phase gate.

A phase gate has a continuous parameter (the angle), labeled 'angle' / self.angle. Its inverse is the same gate with the negated argument. Phase gates of the same class can be merged by adding the angles. The continuous parameter is modulo 2 * pi, self.angle is in the interval [0, 2 * pi).

__init__(*angle*)
Initialize a basic rotation gate.

Parameters **angle** (*float*) – Angle of rotation (saved modulo 2 * 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 BasicPhaseGate.

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

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

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

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

__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

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)
```

projectq.ops.**CRz**(*angle*)
Shortcut for C(Rz(angle), n=1).

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*, *controls=()*, *tags=()*)

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*, *controls=()*, *tags=()*)

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** (projectq.cengines.BasicEngine) – engine which created the qubit (mostly the MainEngine)
- **gate** (*projectq.ops.Gate*) – Gate to be executed
- **qubits** (*tuple[Qureg]*) – Tuple of quantum registers (to which the gate is applied)
- **controls** (*Qureg|list[Qubit]*) – Qubits that condition the command.
- **tags** (*list[object]*) – Tags associated with the command.

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 gate, i.e., the gate is only executed if all qubits are in state 1.

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 & 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.

__or__(*qubits*)
Apply the controlled gate to qubits, using the first n qubits as controls.

Note: The control qubits can be split across the first quregs. However, the n-th control qubit needs to be the last qubit in a qureg. The following quregs belong to the gate.

Parameters **qubits** (*tuple of lists of Qubit objects*) – qubits to which to apply the gate.

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).

tex_str()
Return the Latex string representation of a Daggered gate.

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

get_inverse()
Return the inverse gate.

Standard implementation of get_inverse:

Raises *NotInvertible* – inverse is not implemented

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 why 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 (for single qubits).

__or__(*qubits*)
Previously (ProjectQ <= v0.3.6) MeasureGate/Measure was allowed to be applied to any number of quantum registers. Now the MeasureGate/Measure is strictly a single qubit gate. In the coming releases the backward compatibility will be removed!

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.**QFTGate**
Quantum Fourier Transform gate.

class projectq.ops.**QubitOperator**(*term=None*, *coefficient=1.0*)
A sum of terms acting on qubits, e.g., 0.5 * 'X0 X5' + 0.3 * 'Z1 Z2'.

A term is an operator acting on n qubits and can be represented as:

coefficent * local_operator[0] x ... x local_operator[n-1]

where x is the tensor product. A local operator is a Pauli operator ('I', 'X', 'Y', or 'Z') which acts on one qubit. In math notation a term is, for example, 0.5 * 'X0 X5', which means that a Pauli X operator acts on qubit 0 and 5, while the identity operator acts on all other qubits.

A QubitOperator represents a sum of terms acting on qubits and overloads operations for easy manipulation of these objects by the user.

Note for a QubitOperator to be a Hamiltonian which is a hermitian operator, the coefficients of all terms must be real.

```
hamiltonian = 0.5 * QubitOperator('X0 X5') + 0.3 * QubitOperator('Z0')
```

Our Simulator takes a hermitian QubitOperator to directly calculate the expectation value (see Simulator.get_expectation_value) of this observable.

A hermitian QubitOperator can also be used as input for the TimeEvolution gate.

If the QubitOperator is unitary, i.e., it contains only one term with a coefficient, whose absolute value is 1, then one can apply it directly to qubits:

```
eng = projectq.MainEngine()
qureg = eng.allocate_qureg(6)
QubitOperator('X0 X5', 1.j) | qureg  # Applies X to qubit 0 and 5
                                     # with an additional global phase
                                     # of 1.j
```

terms

dict – **key**: A term represented by a tuple containing all non-trivial local Pauli operators ('X', 'Y', or 'Z'). A non-trivial local Pauli operator is specified by a tuple with the first element being an integer indicating the qubit on which a non-trivial local operator acts and the second element being a string, either 'X', 'Y', or 'Z', indicating which non-trivial Pauli operator acts on that qubit. Examples: ((1, 'X'),) or ((1, 'X'), (4,'Z')) or the identity (). The tuples representing the non-trivial local terms are sorted according to the qubit number they act on, starting from 0. **value**: Coefficient of this term as a (complex) float

__init__(*term=None*, *coefficient=1.0*)

Inits a QubitOperator.

The init function only allows to initialize one term. Additional terms have to be added using += (which is fast) or using + of two QubitOperator objects:

Example

```
ham = ((QubitOperator('X0 Y3', 0.5)
        + 0.6 * QubitOperator('X0 Y3')))
# Equivalently
ham2 = QubitOperator('X0 Y3', 0.5)
ham2 += 0.6 * QubitOperator('X0 Y3')
```

Note: Adding terms to QubitOperator is faster using += (as this is done by in-place addition). Specifying the coefficient in the __init__ is faster than by multiplying a QubitOperator with a scalar as calls an out-of-place multiplication.

Parameters

- **coefficient** (*complex float, optional*) – The coefficient of the first term of this QubitOperator. Default is 1.0.
- **term** (*optional, empy tuple, a tuple of tuples, or a string*) –

1. Default is None which means there are no terms in the QubitOperator hence it is the "zero" Operator
2. An empty tuple means there are no non-trivial Pauli operators acting on the qubits hence only identities with a coefficient (which by default is 1.0).
3. A sorted tuple of tuples. The first element of each tuple is an integer indicating the qubit on which a non-trivial local operator acts, starting from zero. The second element of each tuple is a string, either 'X', 'Y' or 'Z', indicating which local operator acts on that qubit.
4. A string of the form 'X0 Z2 Y5', indicating an X on qubit 0, Z on qubit 2, and Y on qubit 5. The string should be sorted by the qubit number. '' is the identity.

Raises `QubitOperatorError` – Invalid operators provided to QubitOperator.

__or__(*qubits*)

Operator| overload which enables the following syntax:

```
QubitOperator(...) | qureg
QubitOperator(...) | (qureg,)
QubitOperator(...) | qubit
QubitOperator(...) | (qubit,)
```

Unlike other gates, this gate is only allowed to be applied to one quantum register or one qubit and only if the QubitOperator is unitary, i.e., consists of one term with a coefficient whose absolute values is 1.

Example:

```
eng = projectq.MainEngine()
qureg = eng.allocate_qureg(6)
QubitOperator('X0 X5', 1.j) | qureg  # Applies X to qubit 0 and 5
                                     # with an additional global
                                     # phase of 1.j
```

While in the above example the QubitOperator gate is applied to 6 qubits, it only acts non-trivially on the two qubits qureg[0] and qureg[5]. Therefore, the operator| will create a new rescaled QubitOperator, i.e, it sends the equivalent of the following new gate to the MainEngine:

```
QubitOperator('X0 X1', 1.j) | [qureg[0], qureg[5]]
```

which is only a two qubit gate.

Parameters **qubits** – one Qubit object, one list of Qubit objects, one Qureg object, or a tuple of the former three cases.

Raises

- `TypeError` – If QubitOperator is not unitary or applied to more than one quantum register.
- `ValueError` – If quantum register does not have enough qubits

compress(*abs_tol=1e-12*)

Eliminates all terms with coefficients close to zero and removes imaginary parts of coefficients that are close to zero.

Parameters **abs_tol** (*float*) – Absolute tolerance, must be at least 0.0

get_inverse()

Return the inverse gate of a QubitOperator if applied as a gate.

Raises *NotInvertible* – Not implemented for QubitOperators which have multiple terms or a coefficient with absolute value not equal to 1.

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

Standard implementation of get_merged:

Raises *NotMergeable* – merging is not possible

isclose(*other*, *rel_tol=1e-12*, *abs_tol=1e-12*)
Returns True if other (QubitOperator) is close to self.

Comparison is done for each term individually. Return True if the difference between each term in self and other is less than the relative tolerance w.r.t. either other or self (symmetric test) or if the difference is less than the absolute tolerance.

Parameters

- **other** (*QubitOperator*) – QubitOperator to compare against.
- **rel_tol** (*float*) – Relative tolerance, must be greater than 0.0
- **abs_tol** (*float*) – Absolute tolerance, must be at least 0.0

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) == H, it is a self-inverse gate:
get_inverse(H) | qubit
```

get_inverse()
Return the inverse gate.

Standard implementation of get_inverse:

Raises *NotInvertible* – inverse is not implemented

class projectq.ops.**SqrtSwapGate**
Square-root Swap gate class

__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]])
```

class projectq.ops.**SqrtXGate**
Square-root X gate class

class projectq.ops.**StatePreparation**(*final_state*)
Gate for transforming qubits in state |0> to any desired quantum state.

__init__(*final_state*)
Initialize StatePreparation gate.

Example

```
qureg = eng.allocate_qureg(2)
StatePreparation([0.5, -0.5j, -0.5, 0.5]) | qureg
```

Note: The amplitude of state k is final_state[k]. When the state k is written in binary notation, then qureg[0] denotes the qubit whose state corresponds to the least significant bit of k.

Parameters **final_state** (*list[complex]*) – wavefunction of the desired quantum state. len(final_state) must be 2**len(qureg). Must be normalized!

class projectq.ops.**SwapGate**
Swap gate class (swaps 2 qubits)

__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]])
```

---

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.

__or__(*qubits*)
Applies the gate to every qubit in the quantum register qubits.

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

class projectq.ops.**TimeEvolution**(*time*, *hamiltonian*)
Gate for time evolution under a Hamiltonian (QubitOperator object).

This gate is the unitary time evolution propagator: exp(-i * H * t), where H is the Hamiltonian of the system and t is the time. Note that -i factor is stored implicitely.

Example

```
wavefunction = eng.allocate_qureg(5)
hamiltonian = 0.5 * QubitOperator("X0 Z1 Y5")
# Apply exp(-i * H * t) to the wavefunction:
TimeEvolution(time=2.0, hamiltonian=hamiltonian) | wavefunction
```

time
float, int – time t

hamiltonian
QubitOperator – hamiltonaian H

__init__(*time*, *hamiltonian*)
Initialize time evolution gate.

Note: The hamiltonian must be hermitian and therefore only terms with real coefficients are allowed. Coefficients are internally converted to float.

Parameters

- **time** (*float, or int*) – time to evolve under (can be negative).
- **hamiltonian** (QubitOperator) – hamiltonian to evolve under.

Raises

- TypeError – If time is not a numeric type and hamiltonian is not a QubitOperator.
- NotHermitianOperatorError – If the input hamiltonian is not hermitian (only real coefficients).

__or__(*qubits*)

Operator| overload which enables the following syntax:

```
TimeEvolution(...) | qureg
TimeEvolution(...) | (qureg,)
TimeEvolution(...) | qubit
TimeEvolution(...) | (qubit,)
```

Unlike other gates, this gate is only allowed to be applied to one quantum register or one qubit.

Example:

```
wavefunction = eng.allocate_qureg(5)
hamiltonian = QubitOperator("X1 Y3", 0.5)
TimeEvolution(time=2.0, hamiltonian=hamiltonian) | wavefunction
```

While in the above example the TimeEvolution gate is applied to 5 qubits, the hamiltonian of this TimeEvolution gate acts only non-trivially on the two qubits wavefunction[1] and wavefunction[3]. Therefore, the operator| will rescale the indices in the hamiltonian and sends the equivalent of the following new gate to the MainEngine:

```
h = QubitOperator("X0 Y1", 0.5)
TimeEvolution(2.0, h) | [wavefunction[1], wavefunction[3]]
```

which is only a two qubit gate.

Parameters **qubits** – one Qubit object, one list of Qubit objects, one Qureg object, or a tuple of the former three cases.

get_inverse()

Return the inverse gate.

get_merged(*other*)

Return self merged with another TimeEvolution gate if possible.

Two TimeEvolution gates are merged if:

1. both have the same terms
2. the proportionality factor for each of the terms must have relative error <= 1e-9 compared to the proportionality factors of the other terms.

Note: While one could merge gates for which both hamiltonians commute, we are not doing this as in general the resulting gate would have to be decomposed again.

Note: We are not comparing if terms are proportional to each other with an absolute tolerance. It is up to the user to remove terms close to zero because we cannot choose a suitable absolute error which works for everyone. Use, e.g., a decomposition rule for that.

Parameters **other** – TimeEvolution gate

Raises *NotMergeable* – If the other gate is not a TimeEvolution gate or hamiltonians are not suitable for merging.

Returns New TimeEvolution gate equivalent to the two merged gates.

class projectq.ops.**UniformlyControlledRy**(*angles*)

Uniformly controlled Ry gate as introduced in arXiv:quant-ph/0312218.

This is an n-qubit gate. There are n-1 control qubits and one target qubit. This gate applies Ry(angles(k)) to the target qubit if the n-1 control qubits are in the classical state k. As there are 2^(n-1) classical states for the control qubits, this gate requires 2^(n-1) (potentially different) angle parameters.

Example

```
```

controls = eng.allocate_qureg(2) target = eng.allocate_qubit() UniformlyControlledRy(angles=[0.1, 0.2, 0.3, 0.4]) | (controls, target)

Note: The first quantum register contains the control qubits. When converting the classical state k of the control qubits to an integer, we define controls[0] to be the least significant (qu)bit. controls can also be an empty list in which case the gate corresponds to an Ry.

Parameters **angles** (*list[float]*) – Rotation angles. Ry(angles[k]) is applied conditioned on the control qubits being in state k.

__init__(*angles*)

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]])
```

---

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

class projectq.ops.**UniformlyControlledRz**(*angles*)
Uniformly controlled Rz gate as introduced in arXiv:quant-ph/0312218.

This is an n-qubit gate. There are n-1 control qubits and one target qubit. This gate applies Rz(angles(k)) to the target qubit if the n-1 control qubits are in the classical state k. As there are 2^(n-1) classical states for the control qubits, this gate requires 2^(n-1) (potentially different) angle parameters.

Example

```

```

controls = eng.allocate_qureg(2) target = eng.allocate_qubit() UniformlyControlledRz(angles=[0.1, 0.2, 0.3, 0.4]) | (controls, target)

Note: The first quantum register are the contains qubits. When converting the classical state k of the control qubits to an integer, we define controls[0] to be the least significant (qu)bit. controls can also be an empty list in which case the gate corresponds to an Rz.

Parameters **angles** (*list[float]*) – Rotation angles. Rz(angles[k]) is applied conditioned on the control qubits being in state k.

__init__(*angles*)
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]])
```

---

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

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)
```

3.6 setups

The setups package contains a collection of setups which can be loaded by the *MainEngine*. Each setup contains a *get_engine_list* function which returns a list of compiler engines:

Example:

```
import projectq.setups.ibm as ibm_setup
from projectq import MainEngine
eng = MainEngine(engine_list=ibm_setup.get_engine_list())
# eng uses the default Simulator backend
```

The subpackage decompositions contains all the individual decomposition rules which can be given to, e.g., an *AutoReplacer*.

3.6.1 Subpackages

decompositions

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

projectq.setups.decompositions.arb1qubit2rzandry	Registers the Z-Y decomposition for an arbitrary one qubit gate.
projectq.setups.decompositions.barrier	Registers a decomposition rule for barriers.
projectq.setups.decompositions.carb1qubit2cnotrzandry	Registers the decomposition of an controlled arbitary single qubit gate.
projectq.setups.decompositions.cnu2toffoliandcu	Registers a decomposition rule for multi-controlled gates.
projectq.setups.decompositions.crz2cxandrz	Registers a decomposition for controlled z-rotation gates.
projectq.setups.decompositions.entangle	Registers a decomposition for the Entangle gate.
projectq.setups.decompositions.globalphase	Registers a decomposition rule for global phases.
projectq.setups.decompositions.ph2r	Registers a decomposition for the controlled global phase gate.
projectq.setups.decompositions.qft2crandhadamard	Registers a decomposition rule for the quantum Fourier transform.
projectq.setups.decompositions.r2rzandph	Registers a decomposition rule for the phase-shift gate.
projectq.setups.decompositions.rx2rz	Registers a decomposition for the Rx gate into an Rz gate and Hadamard.
projectq.setups.decompositions.ry2rz	Registers a decomposition for the Ry gate into an Rz and Rx(pi/2) gate.
projectq.setups.decompositions.swap2cnot	Registers a decomposition to achieve a Swap gate.
projectq.setups.decompositions.time_evolution	Registers decomposition for the TimeEvolution gates.
projectq.setups.decompositions.toffoli2cnotandtgate	Registers a decomposition rule for the Toffoli gate.

Submodules

projectq.setups.decompositions.arb1qubit2rzandry module

Registers the Z-Y decomposition for an arbitrary one qubit gate.

See paper "Elementary gates for quantum computing" by Adriano Barenco et al., arXiv:quant-ph/9503016v1. (Note: They use different gate definitions!) Or see theorem 4.1 in Nielsen and Chuang.

Decompose an arbitrary one qubit gate U into U = e^(i alpha) Rz(beta) Ry(gamma) Rz(delta). If a gate V is element of SU(2), i.e., determinant == 1, then V = Rz(beta) Ry(gamma) Rz(delta)

projectq.setups.decompositions.arb1qubit2rzandry.**all_defined_decomposition_rules = [<proje**
Decomposition rules

projectq.setups.decompositions.barrier module

Registers a decomposition rule for barriers.

Deletes all barriers if they are not supported.

projectq.setups.decompositions.barrier.**all_defined_decomposition_rules = [<projectq.cengine**
Decomposition rules

projectq.setups.decompositions.carb1qubit2cnotrzandry module

Registers the decomposition of an controlled arbitary single qubit gate.

See paper "Elementary gates for quantum computing" by Adriano Barenco et al., arXiv:quant-ph/9503016v1. (Note: They use different gate definitions!) or Nielsen and Chuang chapter 4.3.

projectq.setups.decompositions.carb1qubit2cnotrzandry.**all_defined_decomposition_rules = [<p**
Decomposition rules

projectq.setups.decompositions.cnu2toffoliandcu module

Registers a decomposition rule for multi-controlled gates.

Implements the decomposition of Nielsen and Chuang (Fig. 4.10) which decomposes a C^n(U) gate into a sequence of 2 * (n-1) Toffoli gates and one C(U) gate by using (n-1) ancilla qubits and circuit depth of 2n-1.

projectq.setups.decompositions.cnu2toffoliandcu.**all_defined_decomposition_rules = [<project**
Decomposition rules

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.crz2cxandrz.**all_defined_decomposition_rules = [<projectq.cen**
Decomposition rules

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.entangle.**all_defined_decomposition_rules = [<projectq.cengi**
Decomposition rules

projectq.setups.decompositions.globalphase module

Registers a decomposition rule for global phases.

Deletes global phase gates (which can be ignored).

projectq.setups.decompositions.globalphase.**all_defined_decomposition_rules = [<projectq.ce**
Decomposition rules

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.ph2r.**all_defined_decomposition_rules = [<projectq.cengines._**
Decomposition rules

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.qft2crandhadamard.**all_defined_decomposition_rules = [<projec**
Decomposition rules

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.r2rzandph.**all_defined_decomposition_rules = [<projectq.cengi**
Decomposition rules

projectq.setups.decompositions.rx2rz module

Registers a decomposition for the Rx gate into an Rz gate and Hadamard.

projectq.setups.decompositions.rx2rz.**all_defined_decomposition_rules = [<projectq.cengines**
Decomposition rules

projectq.setups.decompositions.ry2rz module

Registers a decomposition for the Ry gate into an Rz and Rx(pi/2) gate.

projectq.setups.decompositions.ry2rz.**all_defined_decomposition_rules = [<projectq.cengines**
Decomposition rules

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.swap2cnot.**all_defined_decomposition_rules = [<projectq.cengi**
Decomposition rules

projectq.setups.decompositions.time_evolution module

Registers decomposition for the TimeEvolution gates.

An exact straight forward decomposition of a TimeEvolution gate is possible if the hamiltonian has only one term or if all the terms commute with each other in which case one can implement each term individually.

projectq.setups.decompositions.time_evolution.**all_defined_decomposition_rules = [<projectq**
Decomposition rules

projectq.setups.decompositions.toffoli2cnotandtgate module

Registers a decomposition rule for the Toffoli gate.

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

projectq.setups.decompositions.toffoli2cnotandtgate.**all_defined_decomposition_rules = [<pro**
Decomposition rules

Module contents

3.6.2 Submodules

Each of the submodules contains a setup which can be used to specify the *engine_list* used by the *MainEngine* :

projectq.setups.default	Defines the default setup which provides an *engine_list* for the *MainEngine*

Continued on next page

Table 7 – continued from previous page

projectq.setups.grid	Defines a setup to compile to qubits placed in 2-D grid.
projectq.setups.ibm	Defines a setup useful for the IBM QE chip with 5 qubits.
projectq.setups.ibm16	Defines a setup useful for the IBM QE chip with 16 qubits.
projectq.setups.linear	Defines a setup to compile to qubits placed in a linear chain or a circle.
projectq.setups.restrictedgateset	Defines a setup to compile to a restricted gate set.

3.6.3 default

Defines the default setup which provides an *engine_list* for the *MainEngine*

It contains *LocalOptimizers* and an *AutoReplacer* which uses most of the decompositions rules defined in projectq.setups.decompositions

projectq.setups.default.**get_engine_list**()

3.6.4 grid

Defines a setup to compile to qubits placed in 2-D grid.

It provides the *engine_list* for the *MainEngine*. This engine list contains an AutoReplacer with most of the gate decompositions of ProjectQ, which are used to decompose a circuit into only two qubit gates and arbitrary single qubit gates. ProjectQ's GridMapper is then used to introduce the necessary Swap operations to route interacting qubits next to each other. This setup allows to choose the final gate set (with some limitations).

projectq.setups.grid.**get_engine_list**(*num_rows*, *num_columns*, *one_qubit_gates='any'*, *two_qubit_gates=(<projectq.ops._metagates.ControlledGate object>, <projectq.ops._gates.SwapGate object>)*)

Returns an engine list to compile to a 2-D grid of qubits.

Note: If you choose a new gate set for which the compiler does not yet have standard rules, it raises an *NoGateDecompositionError* or a *RuntimeError: maximum recursion depth exceeded...*. Also note that even the gate sets which work might not yet be optimized. So make sure to double check and potentially extend the decomposition rules. This implemention currently requires that the one qubit gates must contain Rz and at least one of {Ry(best), Rx, H} and the two qubit gate must contain CNOT (recommended) or CZ.

Note: Classical instructions gates such as e.g. Flush and Measure are automatically allowed.

Example

get_engine_list(num_rows=2, num_columns=3, one_qubit_gates=(Rz, Ry, Rx, H), two_qubit_gates=(CNOT,))

Parameters

- **num_rows** (*int*) – Number of rows in the grid
- **num_columns** (*int*) – Number of columns in the grid.

- **one_qubit_gates** – "any" allows any one qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. X), it allows all gates which are equal to it. If the gate is a class (Rz), it allows all instances of this class. Default is "any"
- **two_qubit_gates** – "any" allows any two qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. CNOT), it allows all gates which are equal to it. If the gate is a class, it allows all instances of this class. Default is (CNOT, Swap).

Raises `TypeError` – If input is for the gates is not "any" or a tuple.

Returns A list of suitable compiler engines.

`projectq.setups.grid.`**`high_level_gates`**(*eng*, *cmd*)
Remove any MathGates.

`projectq.setups.grid.`**`one_and_two_qubit_gates`**(*eng*, *cmd*)

3.6.5 ibm

Defines a setup useful for the IBM QE chip with 5 qubits.

It provides the *engine_list* for the *MainEngine*, and contains an AutoReplacer with most of the gate decompositions of ProjectQ, 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
- Arbitrary single qubit gates –> Rz and Ry
- Controlled arbitrary single qubit gates –> Rz, Ry, and CNOT gates

Moreover, it contains *LocalOptimizers* and a custom mapper for the CNOT gates.

`projectq.setups.ibm.`**`get_engine_list`**()

3.6.6 ibm16

Defines a setup useful for the IBM QE chip with 16 qubits.

It provides the *engine_list* for the *MainEngine*, and contains an AutoReplacer with most of the gate decompositions of ProjectQ, 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
- Arbitrary single qubit gates –> Rz and Ry
- Controlled arbitrary single qubit gates –> Rz, Ry, and CNOT gates

Moreover, it contains *LocalOptimizers*.

projectq.setups.ibm16.**get_engine_list**()

3.6.7 linear

Defines a setup to compile to qubits placed in a linear chain or a circle.

It provides the *engine_list* for the *MainEngine*. This engine list contains an AutoReplacer with most of the gate decompositions of ProjectQ, which are used to decompose a circuit into only two qubit gates and arbitrary single qubit gates. ProjectQ's LinearMapper is then used to introduce the necessary Swap operations to route interacting qubits next to each other. This setup allows to choose the final gate set (with some limitations).

projectq.setups.linear.**get_engine_list**(*num_qubits*, *cyclic=False*, *one_qubit_gates='any'*, *two_qubit_gates=(<projectq.ops._metagates.ControlledGate object>, <projectq.ops._gates.SwapGate object>)*)

Returns an engine list to compile to a linear chain of qubits.

Note: If you choose a new gate set for which the compiler does not yet have standard rules, it raises an *NoGateDecompositionError* or a *RuntimeError: maximum recursion depth exceeded...*. Also note that even the gate sets which work might not yet be optimized. So make sure to double check and potentially extend the decomposition rules. This implemention currently requires that the one qubit gates must contain Rz and at least one of {Ry(best), Rx, H} and the two qubit gate must contain CNOT (recommended) or CZ.

Note: Classical instructions gates such as e.g. Flush and Measure are automatically allowed.

Example

get_engine_list(num_qubits=10, cyclic=False, one_qubit_gates=(Rz, Ry, Rx, H), two_qubit_gates=(CNOT,))

Parameters

- **num_qubits** (*int*) – Number of qubits in the chain
- **cyclic** (*bool*) – If a circle or not. Default is False
- **one_qubit_gates** – "any" allows any one qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. X), it allows all gates which are equal to it. If the gate is a class (Rz), it allows all instances of this class. Default is "any"
- **two_qubit_gates** – "any" allows any two qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. CNOT), it allows all gates which are equal to it. If the gate is a class, it allows all instances of this class. Default is (CNOT, Swap).

Raises TypeError – If input is for the gates is not "any" or a tuple.

Returns A list of suitable compiler engines.

projectq.setups.linear.**high_level_gates**(*eng*, *cmd*)

Remove any MathGates.

projectq.setups.linear.**one_and_two_qubit_gates**(*eng*, *cmd*)

3.6.8 restrictedgateset

Defines a setup to compile to a restricted gate set.

It provides the *engine_list* for the *MainEngine*. This engine list contains an AutoReplacer with most of the gate decompositions of ProjectQ, which are used to decompose a circuit into a restricted gate set (with some limitions on the choice of gates).

projectq.setups.restrictedgateset.**get_engine_list**(*one_qubit_gates='any'*, *two_qubit_gates=(<projectq.ops._metagates.ControlledGat object>,)*, *other_gates=()*)

Returns an engine list to compile to a restricted gate set.

Note: If you choose a new gate set for which the compiler does not yet have standard rules, it raises an *NoGateDecompositionError* or a *RuntimeError: maximum recursion depth exceeded...*. Also note that even the gate sets which work might not yet be optimized. So make sure to double check and potentially extend the decomposition rules. This implemention currently requires that the one qubit gates must contain Rz and at least one of {Ry(best), Rx, H} and the two qubit gate must contain CNOT (recommended) or CZ.

Note: Classical instructions gates such as e.g. Flush and Measure are automatically allowed.

Example

get_engine_list(one_qubit_gates=(Rz, Ry, Rx, H), two_qubit_gates=(CNOT,), other_gates=(TimeEvolution,))

Parameters

- **one_qubit_gates** – "any" allows any one qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. X), it allows all gates which are equal to it. If the gate is a class (Rz), it allows all instances of this class. Default is "any"
- **two_qubit_gates** – "any" allows any two qubit gate, otherwise provide a tuple of the allowed gates. If the gates are instances of a class (e.g. CNOT), it allows all gates which are equal to it. If the gate is a class, it allows all instances of this class. Default is (CNOT,).
- **other_gates** – A tuple of the allowed gates. If the gates are instances of a class (e.g. QFT), it allows all gates which are equal to it. If the gate is a class, it allows all instances of this class.

Raises `TypeError` – If input is for the gates is not "any" or a tuple.

Returns A list of suitable compiler engines.

projectq.setups.restrictedgateset.**high_level_gates**(*eng*, *cmd*)

Remove any MathGates.

projectq.setups.restrictedgateset.**one_and_two_qubit_gates**(*eng*, *cmd*)

3.6.9 Module contents

3.7 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.

projectq.types.BasicQubit(engine, idx)	BasicQubit objects represent qubits.
projectq.types.Qubit(engine, idx)	Qubit class.
projectq.types.Qureg	Quantum register class.
projectq.types.WeakQubitRef(engine, idx)	WeakQubitRef objects are used inside the Command object.

3.7.1 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).

__ne__(*other*)
Return self!=value.

__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

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

T

U

W

X

Y

Z