projectq Documentation

Release 0.8.1.dev19

a

Mar 09, 2023

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

ONE

TUTORIAL

1.1 Getting started

To start using ProjectQ, simply run

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

Since version 0.6.0, ProjectQ is available as pre-compiled binary wheels in addition to the traditional source package. These wheels should work on most platforms, provided that your processor supports AVX2 instructions. Should you encounter any troubles while installation ProjectQ in binary form, you can always try tom compile the project manually as described below. You may want to pass the *–no-binary projectq* flag to Pip during the installation to make sure that you are downloading the source package.

Alternatively, you can also clone/download this repository (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 the C++ compilation were to fail, the setup will install a pure Python implementation of the simulator instead. The Python simulator should work fine for small examples (e.g., running Shor's algorithm for factoring 15 or 21).

If you want to skip the installation of the C++-Simulator altogether, you can define the `PROJECTQ_DISABLE_CEXT` environment variable to avoid any compilation steps.

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

```
env CC=g++-10 python -m pip install --user projectq
```

Please note that the compiler you specify must support at least **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 may cause some problems.

Install AWS Braket Backend requirement

AWS Braket Backend requires the use of the official AWS SDK for Python, Boto3. This is an extra requirement only needed if you plan to use the AWS Braket Backend. To install ProjectQ including this requirement you can include it in the installation instruction as

```
python -m pip install --user projectq[braket]
```

Install Azure Quantum Backend requirement

Azure Quantum Backend requires the use of the official Azure Quantum SDK for Python. This is an extra requirement only needed if you plan to use the Azure Quantum Backend. To install ProjectQ including this requirement you can include it in the installation instruction as

```
python -m pip install --user projectq[azure-quantum]
```

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

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

ArchLinux/Manjaro:

Make sure that you have a C/C++ compiler installed:

```
sudo pacman -Syu gcc
```

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

```
sudo pacman -Syu python python-pip
```

When you then run

```
sudo python3 -m pip 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 Python 3.7 installers from python.org or ANACONDA. Installing ProjectQ right away will succeed for the (slow) Python simulator. 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 the Visual Studio compiler.

If the Python executable is added to your PATH (option normally suggested at the end of the Python installation procedure), you can then open a cmdline window (WIN + R, type "cmd" and click *OK*) and enter the following in order to install ProjectQ:

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

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:

Similarly to the other platforms, installing ProjectQ without the C++ simulator is really easy:

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

In order to install the fast C++ simulator, we require that a C++ compiler is installed on your system. There are essentially three options you can choose from:

1. Using the compiler provided by Apple through the XCode command line tools.
2. Using Homebrew
3. Using MacPorts

For both options 2 and 3, you will be required to first install the XCode command line tools

Apple XCode command line tool

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

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

Next, you will need to install Python and pip. See options 2 and 3 for information on how to install a newer python version with either Homebrew or 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:

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

Note that the compiler provided by Apple is currently not able to compile ProjectQ's multi-threaded code.

Homebrew

First install the XCode command line tools. Then install Homebrew with the following command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
↪install/master/install)"
```

Then proceed to install Python as well as a C/C++ compiler (note: gcc installed via Homebrew may lead to some issues):

```
brew install python llvm
```

You should now be able to install ProjectQ with the C++ simulator using the following command:

```
env P=/usr/local/opt/llvm/bin CC=$P/clang CXX=$P/clang++ python3 -m pip␣
↪install --user projectq
```

MacPorts

Visit macports.org and install the latest version that corresponds to your operating system's version. Afterwards, open a new terminal window.

Then, use macports to install Python 3.7 by entering the following command

```
sudo port install python37
```

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

```
sudo port install py37-gnureadline
```

Install pip by

```
sudo port install py37-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 9.0 also using macports (note: gcc installed via macports does not work).

```
sudo port install clang-9.0
```

ProjectQ is now installed by:

```
env CC=clang-mp-9.0 env CXX=clang++-mp-9.0 /opt/local/bin/python3.7 -m pip␣
↪install --user 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(f"Measured {int(qubit)}")  # output measurement result
```

Which creates random bits (0 or 1).

CHAPTER

TWO

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

```
# pylint: skip-file

"""Example of a simple quantum random number generator."""

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

# 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(f"Measured: {int(q1)}")
```

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

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

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

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

All you need to do is:

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

```
--- /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
@@ -1,12 +1,14 @@
 # pylint: skip-file

-"""Example of a simple quantum random number generator."""
+"""Example of a simple quantum random number generator using IBM's API."""

+import projectq.setups.ibm
 from projectq import MainEngine
+from projectq.backends import IBMBackend
 from projectq.ops import H, Measure

 # 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

```
from projectq.meta import Control, Dagger
        eng (MainEngine): MainEngine from which to allocate the qubits.

    Returns:
        bell_pair (tuple<Qubits>): The Bell-pair.
    """
    b1 = eng.allocate_qubit()
    b2 = eng.allocate_qubit()
```

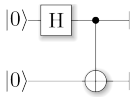
with a main compiler engine which has a CircuitDrawer back-end, i.e.,

```
# pylint: skip-file

"""Example implementation of a quantum circuit generating a Bell pair state."""

import matplotlib.pyplot as plt
from teleport import create_bell_pair

from projectq import MainEngine
from projectq.backends import CircuitDrawer
from projectq.libs.hist import histogram
from projectq.setups.default import get_engine_list

# create a main compiler engine
drawing_engine = CircuitDrawer()
eng = MainEngine(engine_list=get_engine_list() + [drawing_engine])

qb0, qb1 = create_bell_pair(eng)

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

histogram(eng.backend, [qb0, qb1])
plt.show()
```

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:

```
# pylint: skip-file

"""Example of a quantum teleportation circuit."""

from projectq import MainEngine
from projectq.meta import Control, Dagger
        eng (MainEngine): MainEngine from which to allocate the qubits.

    Returns:
        bell_pair (tuple<Qubits>): The Bell-pair.
    """
    b1 = eng.allocate_qubit()
    b2 = eng.allocate_qubit()

    H | b1
    CNOT | (b1, b2)
        verbose (bool): If True, info messages will be printed.

    """
    # make a Bell-pair
    b1, b2 = create_bell_pair(eng)

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

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

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

(continues on next page)

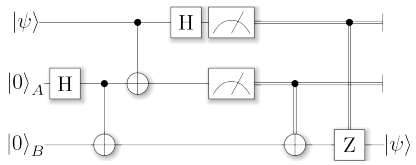
(continued from previous page)

```
        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):
        """Create a quantum state."""
        H | qb
```

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

```
86      return r
87
88
89  # Filter function, which defines the gate set for the first optimization
90  # (don't decompose QFTs and iQFTs to make cancellation easier)
91  def high_level_gates(eng, cmd):
92      """Filter high-level gates."""
93      g = cmd.gate
94      if g == QFT or get_inverse(g) == QFT or g == Swap:
95          return True
96      if isinstance(g, BasicMathGate):
```

(continues on next page)

(continued from previous page)

```
97          return False
98          if isinstance(g, AddConstant):
99              return True
```

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

```
50      measurements = [0] * (2 * n)  # will hold the 2n measurement results
51
52      ctrl_qubit = eng.allocate_qubit()
53
54      for k in range(2 * n):
55          current_a = pow(a, 1 << (2 * n - 1 - k), N)
56          # one iteration of 1-qubit QPE
57          H | ctrl_qubit
58          with Control(eng, ctrl_qubit):
59              MultiplyByConstantModN(current_a, N) | x
60
61          # perform inverse QFT --> Rotations conditioned on previous outcomes
62          for i in range(k):
63              if measurements[i]:
64                  R(-math.pi / (1 << (k - i))) | ctrl_qubit
65          H | ctrl_qubit
66
67          # and measure
68          Measure | ctrl_qubit
69          eng.flush()
```

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

THREE

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._aqt	ProjectQ module for supporting the AQT platform.
projectq.backends._awsbraket	ProjectQ module for supporting the AWS Braket platform.
projectq.backends._azure	ProjectQ module for supporting the Azure Quantum platform.
projectq.backends._circuits	ProjectQ module for exporting/printing quantum circuits.
projectq.backends._exceptions	Exception classes for projectq.backends.
projectq.backends._ibm	ProjectQ module for supporting the IBM QE platform.
projectq.backends._ionq	ProjectQ module for supporting the IonQ platform.
projectq.backends._printer	Contains a compiler engine which prints commands to stdout prior to sending them on to the next engines.
projectq.backends._resource	Contain a compiler engine to calculate resource count used by a quantum circuit.
projectq.backends._sim	ProjectQ module dedicated to simulation.
projectq.backends._unitary	Contain a backend that saves the unitary of a quantum circuit.
projectq.backends._utils	Module containing some utility functions.
projectq.backends.AQTBackend([use_hardware, ...])	Backend for building circuits and submitting them to the AQT API.
projectq.backends.AWSBraketBackend(*args, ...)	Dummy class.
projectq.backends.AzureQuantumBackend(*args, ...)	Dummy class.
projectq.backends.CircuitDrawer([...])	CircuitDrawer is a compiler engine which generates TikZ code for drawing quantum circuits.
projectq.backends.CircuitDrawerMatplotlib([...])	CircuitDrawerMatplotlib is a compiler engine which using Matplotlib library for drawing quantum circuits.
projectq.backends.ClassicalSimulator()	A simple introspective simulator that only permits classical operations.
projectq.backends.CommandPrinter([...])	Compiler engine that prints command to the standard output.
projectq.backends.DeviceNotHandledError	Exception raised if a selected device cannot handle the circuit or is not supported by ProjectQ.
projectq.backends.DeviceOfflineError	Raised when a device is required but is currently offline.
projectq.backends.DeviceTooSmall	Raised when a device does not have enough qubits for a desired job.
projectq.backends.IBMBackend([use_hardware, ...])	Define the compiler engine class that handles interactions with the IBM API.
projectq.backends.IonQBackend([...])	Backend for building circuits and submitting them to the IonQ API.
projectq.backends.ResourceCounter()	ResourceCounter is a compiler engine which counts the number of gates and max.
projectq.backends.Simulator([gate_fusion, ...])	Simulator is a compiler engine which simulates a quantum computer using C++-based kernels.
projectq.backends.UnitarySimulator()	Simulator engine aimed at calculating the unitary transformation that represents the current quantum circuit.

3.1.1 Submodules

_aqt

ProjectQ module for supporting the AQT platform.

_awsbraket

ProjectQ module for supporting the AWS Braket platform.

class projectq.backends._awsbraket.**AWSBraketBackend**(**args*, ***kwargs*)

Dummy class.

_azure

ProjectQ module for supporting the Azure Quantum platform.

class projectq.backends._azure.**AzureQuantumBackend**(**args*, ***kwargs*)

Dummy class.

_circuits

ProjectQ module for exporting/printing quantum circuits.

_exceptions

Exception classes for projectq.backends.

exception projectq.backends._exceptions.**DeviceNotHandledError**

Exception raised if a selected device cannot handle the circuit or is not supported by ProjectQ.

exception projectq.backends._exceptions.**DeviceOfflineError**

Raised when a device is required but is currently offline.

exception projectq.backends._exceptions.**DeviceTooSmall**

Raised when a device does not have enough qubits for a desired job.

exception projectq.backends._exceptions.**InvalidCommandError**

Raised if the backend encounters an invalid command.

exception projectq.backends._exceptions.**JobSubmissionError**

Raised when the job creation API contains an error of some kind.

exception projectq.backends._exceptions.**MidCircuitMeasurementError**

Raised when a mid-circuit measurement is detected on a qubit.

exception projectq.backends._exceptions.**RequestTimeoutError**

Raised if a request to the job creation API times out.

_ibm

ProjectQ module for supporting the IBM QE platform.

_ionq

ProjectQ module for supporting the IonQ platform.

class projectq.backends._ionq.**IonQBackend**(*use_hardware=False*, *num_runs=100*, *verbose=False*, *token=None*, *device='ionq_simulator'*, *num_retries=3000*, *interval=1*, *retrieve_execution=None*)

Backend for building circuits and submitting them to the IonQ API.

get_probabilities(*qureg*)

Given the provided qubit register, determine the probability of each possible outcome.

Note: This method should only be called *after* a circuit has been run and its results are available.

Parameters
qureg (Qureg) – A ProjectQ Qureg object.

Returns
A dict mapping of states -> probability.

Return type
dict

get_probability(*state*, *qureg*)

Shortcut to get a specific state's probability.

Parameters

- **state** (*str*) – A state in bit-string format.
- **qureg** (Qureg) – A ProjectQ Qureg object.

Returns
The probability for the provided state.

Return type
float

is_available(*cmd*)

Test if this backend is available to process the provided command.

Parameters
cmd (Command) – A command to process.

Returns
If this backend can process the command.

Return type
bool

receive(*command_list*)

Receive a command list from the ProjectQ engine pipeline.

If a given command is a "flush" operation, the pending circuit will be submitted to IonQ's API for processing.

Parameters
command_list (*list[Command]*) – A list of ProjectQ Command objects.

_printer

Contains a compiler engine which prints commands to stdout prior to sending them on to the next engines.

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

Compiler engine that prints command to the standard output.

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

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

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.

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).

_resource

Contain a compiler engine to calculate resource count used by a quantum circuit.

A resource counter compiler engine counts the number of calls for each type of gate used in a circuit, in addition to the max. number of active qubits.

class projectq.backends._resource.**ResourceCounter**

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

gate_counts

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

Type
dict

`gate_class_counts`

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.

Type
dict

`max_width`

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

Type
int

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

`property depth_of_dag`

Return the depth of the DAG.

`is_available`(*cmd*)

Test whether a Command is supported by a compiler engine.

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.

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).

_sim

ProjectQ module dedicated to simulation.

_unitary

Contain a backend that saves the unitary of a quantum circuit.

class projectq.backends._unitary.**UnitarySimulator**

Simulator engine aimed at calculating the unitary transformation that represents the current quantum circuit.

unitary

Current unitary representing the quantum circuit being processed so far.

Type
np.ndarray

history

List of previous quantum circuit unitaries.

Type
list<np.ndarray>

Note: The current implementation of this backend resets the unitary after the first gate that is neither a qubit deallocation nor a measurement occurs after one of those two aforementioned gates.

The old unitary call be accessed at anytime after such a situation occurs via the *history* property.

```
eng = MainEngine(backend=UnitarySimulator(), engine_list=[])
qureg = eng.allocate_qureg(3)
All(X) | qureg

eng.flush()
All(Measure) | qureg
eng.deallocate_qubit(qureg[1])

X | qureg[0]  # WARNING: appending gate after measurements or deallocations resets␣
↪the unitary
```

---

property history

Access all previous unitary matrices.

The current unitary matrix is appended to this list once a gate is received after either a measurement or a qubit deallocation has occurred.

Returns
A list where the elements are all previous unitary matrices representing the circuit, separated by measurement/deallocate gates.

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

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

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

Returns
True if it can be simulated and False otherwise.

measure_qubits(*ids*)

Measure the qubits with IDs ids and return a list of measurement outcomes (True/False).

Parameters
ids (*list<int>*) – List of qubit IDs to measure.

Returns
List of measurement results (containing either True or False).

receive(*command_list*)

Receive a list of commands.

Receive a list of commands from the previous engine and handle them:

- update the unitary of the quantum circuit
- update the internal quantum state if a measurement or a qubit deallocation occurs

prior to sending them on to the next engine.

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

property unitary

Access the last unitary matrix directly.

Returns
A numpy array which is the unitary matrix of the circuit.

_utils

Module containing some utility functions.

3.1.2 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).
- an interface to the AQT trapped ion system (and simulator).
- an interface to the AWS Braket service decives (and simulators)
- an interface to the Azure Quantum service devices (and simulators)
- an interface to the IonQ trapped ionq hardware (and simulator).

class projectq.backends.**AQTBackend**(*use_hardware=False*, *num_runs=100*, *verbose=False*, *token=''*, *device='simulator'*, *num_retries=3000*, *interval=1*, *retrieve_execution=None*)

Backend for building circuits and submitting them to the AQT API.

The AQT Backend class, which stores the circuit, transforms it to the appropriate data format, and sends the circuit through the AQT API.

__init__(*use_hardware=False*, *num_runs=100*, *verbose=False*, *token=''*, *device='simulator'*, *num_retries=3000*, *interval=1*, *retrieve_execution=None*)

Initialize the Backend object.

Parameters

- **use_hardware** (*bool*) – If True, the code is run on the AQT quantum chip (instead of using the AQT simulator)
- **num_runs** (*int*) – Number of runs to collect statistics. (default is 100, max is usually around 200)
- **verbose** (*bool*) – If True, statistics are printed, in addition to the measurement result being registered (at the end of the circuit).
- **token** (*str*) – AQT user API token.
- **device** (*str*) – name of the AQT device to use. simulator By default
- **num_retries** (*int*) – Number of times to retry to obtain results from the AQT API. (default is 3000)
- **interval** (*float, int*) – Number of seconds between successive attempts to obtain results from the AQT API. (default is 1)
- **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 probability of the outcome *bit_string* when measuring the quantum register *qureg*.

Return the list of basis states with corresponding probabilities. If input qureg is a subset of the register used for the experiment, then returns the projected probabilities over the other states. 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 AQT ion trap can only do Rx,Ry and Rxx.

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

receive(*command_list*)

Receive a list of commands.

Receive a command list and, for each command, stores it until completion. Upon flush, send the data to the AQT API.

Parameters
command_list – List of commands to execute

class projectq.backends.**AWSBraketBackend**(**args*, ***kwargs*)

Dummy class.

__init__(**args*, ***kwargs*)

Initialize dummy class.

class projectq.backends.**AzureQuantumBackend**(**args*, ***kwargs*)

Dummy class.

__init__(**args*, ***kwargs*)

Initialize dummy class.

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,
        },
```

(continues on next page)

(continued from previous page)

```
        # ...
    }
}
```

The settings.json file has the following structure:

```
{
    "control": {"shadow": false, "size": 0.1},  # settings for control "circle"
    "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(*ordered=False*, *draw_gates_in_parallel=True*)

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.

Parameters

- **ordered** (*bool*) – flag if the gates should be drawn in the order they were added to the circuit
- **draw_gates_in_parallel** (*bool*) – flag if parallel gates should be drawn parallel (True), or not (False)

`is_available`(*cmd*)

Test whether a Command is supported by a compiler engine.

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.

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*)

Set 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.CircuitDrawerMatplotlib`(*accept_input=False*, *default_measure=0*)

CircuitDrawerMatplotlib is a compiler engine which using Matplotlib library for drawing quantum circuits.

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

Initialize a circuit drawing engine(mpl).

Parameters

- **accept_input** (*bool*) – If accept_input is true, the printer queries the user to input measurement results if the CircuitDrawerMPL 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.

draw(*qubit_labels=None*, *drawing_order=None*, ***kwargs*)

Generate and returns the plot of the quantum circuit stored so far.

Parameters

- **qubit_labels** (*dict*) – label for each wire in the output figure. Keys: qubit IDs, Values: string to print out as label for that particular qubit wire.
- **drawing_order** (*dict*) – position of each qubit in the output graphic. Keys: qubit IDs, Values: position of qubit on the qubit line in the graphic.
- ****kwargs** (*dict*) – additional parameters are used to update the default plot parameters

Returns

A tuple containing the matplotlib figure and axes objects

Note: Additional keyword arguments can be passed to this function in order to further customize the figure output by matplotlib (default value in parentheses):

- fontsize (14): Font size in pt
- column_spacing (.5): Vertical spacing between two neighbouring gates (roughly in inches)
- control_radius (.015): Radius of the circle for controls
- labels_margin (1): Margin between labels and begin of wire (roughly in inches)
- linewidth (1): Width of line
- not_radius (.03): Radius of the circle for X/NOT gates
- gate_offset (.05): Inner margins for gates with a text representation
- mgate_width (.1): Width of the measurement gate
- swap_delta (.02): Half-size of the SWAP gate
- x_offset (.05): Absolute X-offset for drawing within the axes
- wire_height (1): Vertical spacing between two qubit wires (roughly in inches)

---

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

Specialized implementation of is_available: Returns True if the CircuitDrawerMatplotlib 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.

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.**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 a ClassicalSimulator object.

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

read_bit(*qubit*)

Read 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*)

Read 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

receive(*command_list*)

Receive a list of commands.

This implementation simply forwards all commands to the next engine.

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*)

Set 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*)

Compiler engine that prints command to the standard output.

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*)

Test whether a Command is supported by a compiler engine.

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.

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).

exception projectq.backends.**DeviceNotHandledError**

Exception raised if a selected device cannot handle the circuit or is not supported by ProjectQ.

exception projectq.backends.**DeviceOfflineError**

Raised when a device is required but is currently offline.

exception projectq.backends.**DeviceTooSmall**

Raised when a device does not have enough qubits for a desired job.

class projectq.backends.**IBMBackend**(*use_hardware=False*, *num_runs=1024*, *verbose=False*, *token=''*, *device='ibmq_essex'*, *num_retries=3000*, *interval=1*, *retrieve_execution=None*)

Define the compiler engine class that handles interactions with the IBM API.

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

__init__(*use_hardware=False*, *num_runs=1024*, *verbose=False*, *token=''*, *device='ibmq_essex'*, *num_retries=3000*, *interval=1*, *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).
- **token** (*str*) – IBM quantum experience user password.
- **device** (*str*) – name of the IBM device to use. ibmq_essex By default
- **num_retries** (*int*) – Number of times to retry to obtain results from the IBM API. (default is 3000)
- **interval** (*float, int*) – Number of seconds between successive attempts to obtain results from the IBM API. (default is 1)
- **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 probability of the outcome *bit_string* when measuring the quantum register *qureg*.

Return the list of basis states with corresponding probabilities. If input qureg is a subset of the register used for the experiment, then returns the projected probabilities over the other states.

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).

`get_qasm()`

Return the QASM representation of the circuit sent to the backend.

Should be called AFTER calling the ibm device.

`is_available(cmd)`

Return true if the command can be executed.

The IBM quantum chip can only do U1,U2,U3,barriers, and CX / CNOT. Conversion implemented for Rotation gates and H gates.

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

`receive(command_list)`

Receive a list of commands.

Receive a command list and, for each command, stores it until completion. Upon flush, send the data to the IBM QE API.

Parameters
command_list – List of commands to execute

`class projectq.backends.IonQBackend(use_hardware=False, num_runs=100, verbose=False, token=None, device='ionq_simulator', num_retries=3000, interval=1, retrieve_execution=None)`

Backend for building circuits and submitting them to the IonQ API.

`__init__(use_hardware=False, num_runs=100, verbose=False, token=None, device='ionq_simulator', num_retries=3000, interval=1, retrieve_execution=None)`

Initialize an IonQBackend object.

Parameters

- **use_hardware** (*bool, optional*) – Whether or not to use real IonQ hardware or just a simulator. If False, the ionq_simulator is used regardless of the value of `device`. Defaults to False.
- **num_runs** (*int, optional*) – Number of times to run circuits. Defaults to 100.
- **verbose** (*bool, optional*) – If True, print statistics after job results have been collected. Defaults to False.
- **token** (*str, optional*) – An IonQ API token. Defaults to None.

- **device** (*str, optional*) – Device to run jobs on. Supported devices are `'ionq_qpu'` or `'ionq_simulator'`. Defaults to `'ionq_simulator'`.
- **num_retries** (*int, optional*) – Number of times to retry fetching a job after it has been submitted. Defaults to 3000.
- **interval** (*int, optional*) – Number of seconds to wait in between result fetch retries. Defaults to 1.
- **retrieve_execution** (*str, optional*) – An IonQ API Job ID. If provided, a job with this ID will be fetched. Defaults to None.

`get_probabilities`(*qureg*)

Given the provided qubit register, determine the probability of each possible outcome.

Note: This method should only be called *after* a circuit has been run and its results are available.

Parameters
qureg (Qureg) – A ProjectQ Qureg object.

Returns
A dict mapping of states -> probability.

Return type
dict

`get_probability`(*state*, *qureg*)

Shortcut to get a specific state's probability.

Parameters

- **state** (*str*) – A state in bit-string format.
- **qureg** (Qureg) – A ProjectQ Qureg object.

Returns
The probability for the provided state.

Return type
float

`is_available`(*cmd*)

Test if this backend is available to process the provided command.

Parameters
cmd (Command) – A command to process.

Returns
If this backend can process the command.

Return type
bool

`receive`(*command_list*)

Receive a command list from the ProjectQ engine pipeline.

If a given command is a "flush" operation, the pending circuit will be submitted to IonQ's API for processing.

Parameters
command_list (*list[*Command*]*) – A list of ProjectQ Command objects.

class projectq.backends.**ResourceCounter**

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

gate_counts

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

Type
dict

gate_class_counts

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.

Type
dict

max_width

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

Type
int

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.

property depth_of_dag

Return the depth of the DAG.

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

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.

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 further 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 a 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*)

Return the expectation value of a qubit operator.

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*)

Test whether a Command is supported by a compiler engine.

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.

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.

`class` projectq.backends.**UnitarySimulator**

Simulator engine aimed at calculating the unitary transformation that represents the current quantum circuit.

`unitary`

Current unitary representing the quantum circuit being processed so far.

Type
np.ndarray

`history`

List of previous quantum circuit unitaries.

Type
list<np.ndarray>

Note: The current implementation of this backend resets the unitary after the first gate that is neither a qubit deallocation nor a measurement occurs after one of those two aforementioned gates.

The old unitary call be accessed at anytime after such a situation occurs via the *history* property.

```
eng = MainEngine(backend=UnitarySimulator(), engine_list=[])
qureg = eng.allocate_qureg(3)
All(X) | qureg

eng.flush()
All(Measure) | qureg
eng.deallocate_qubit(qureg[1])

X | qureg[0]  # WARNING: appending gate after measurements or deallocations resets␣
↪the unitary
```

__init__()

Initialize a UnitarySimulator object.

property history

Access all previous unitary matrices.

The current unitary matrix is appended to this list once a gate is received after either a measurement or a qubit deallocation has occurred.

Returns

A list where the elements are all previous unitary matrices representing the circuit, separated by measurement/deallocate gates.

is_available(*cmd*)

Test whether a Command is supported by a compiler engine.

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

Parameters

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

Returns

True if it can be simulated and False otherwise.

measure_qubits(*ids*)

Measure the qubits with IDs ids and return a list of measurement outcomes (True/False).

Parameters

ids (*list<int>*) – List of qubit IDs to measure.

Returns

List of measurement results (containing either True or False).

receive(*command_list*)

Receive a list of commands.

Receive a list of commands from the previous engine and handle them:

- update the unitary of the quantum circuit
- update the internal quantum state if a measurement or a qubit deallocation occurs

prior to sending them on to the next engine.

Parameters

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

property unitary

Access the last unitary matrix directly.

Returns

A numpy array which is the unitary matrix of the circuit.

3.2 cengines

The ProjectQ compiler engines package.

projectq.cengines._basicmapper	The parent class from which all mappers should be derived.
projectq.cengines._basics	Module containing the basic definition of a compiler engine.
projectq.cengines._cmdmodifier	A CommandModifier engine that can be used to apply a user-defined transformation to all incoming commands.
projectq.cengines._ibm5qubitmapper	Contains a compiler engine to map to the 5-qubit IBM chip.
projectq.cengines._linearmapper	Mapper for a quantum circuit to a linear chain of qubits.
projectq.cengines._main	The main engine of every compiler engine pipeline, called MainEngine.
projectq.cengines._manualmapper	A compiler engine to add mapping information.
projectq.cengines._optimize	A local optimizer engine.
projectq.cengines._replacer	
projectq.cengines._swapandcnotflipper	A compiler engine which flips the directionality of CNOTs according to the given connectivity graph.
projectq.cengines._tagremover	The TagRemover compiler engine.
projectq.cengines._testengine	TestEngine and DummyEngine.
projectq.cengines._twodmapper	Mapper for a quantum circuit to a 2D square grid.
projectq.cengines.AutoReplacer(...[, ...])	A compiler engine to automatically replace certain commands.
projectq.cengines.BasicEngine()	Basic compiler engine: All compiler engines are derived from this class.
projectq.cengines.BasicMapperEngine()	Parent class for all Mappers.
projectq.cengines.CommandModifier(cmd_mod_fun)	Compiler engine applying a user-defined transformation to all incoming commands.
projectq.cengines.CompareEngine()	Command list comparison compiler engine for testing purposes.
projectq.cengines.contextmanager(func)	@contextmanager decorator.
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.flushing(engine)	Context manager to flush the given engine at the end of the 'with' context block.
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.

continues on next page

Table 1 – continued from previous page

projectq.cengines.IBM5QubitMapper([connections])	Mapper for the 5-qubit IBM backend.
projectq.cengines.InstructionFilter(filterfun)	A compiler engine that implements a user-defined is_available() method.
projectq.cengines.LastEngineException(engine)	Exception thrown when the last engine tries to access the next one.
projectq.cengines.LinearMapper(num_qubits[, ...])	Map a quantum circuit to a linear chain of nearest neighbour interactions.
projectq.cengines.LocalOptimizer([cache_size, m])	Circuit optimization compiler engine.
projectq.cengines.MainEngine([backend, ...])	The MainEngine class provides all functionality of the main compiler engine.
projectq.cengines.ManualMapper([map_fun])	Manual Mapper which adds QubitPlacementTags to Allocate gate commands according to a user-specified mapping.
projectq.cengines.NotYetMeasuredError	Exception raised when trying to access the measurement value of a qubit that has not yet been measured.
projectq.cengines.return_swap_depth(swaps)	Return the circuit depth to execute these swaps.
projectq.cengines.SwapAndCNOTFlipper(...)	Flip CNOTs and translates Swaps to CNOTs where necessary.
projectq.cengines.TagRemover([tags])	Compiler engine that remove temporary command tags.
projectq.cengines.UnsupportedEngineError	Exception raised when a non-supported compiler engine is encountered.

3.2.1 Submodules

_basicmapper

The parent class from which all mappers should be derived.

There is only one engine currently allowed to be derived from BasicMapperEngine. This allows the simulator to automatically translate logical qubit ids to mapped ids.

class projectq.cengines._basicmapper.**BasicMapperEngine**

Parent class for all Mappers.

self.**current_mapping**

Keys are the logical qubit ids and values are the mapped qubit ids.

Type
dict

property current_mapping

Access the current mapping.

receive(*command_list*)

Receive a list of commands.

This implementation simply forwards all commands to the next compiler engine while adjusting the qubit IDs of measurement gates.

_basics

Module containing the basic definition of a compiler engine.

class projectq.cengines._basics.**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

Next compiler engine (or the back-end).

Type
BasicEngine

main_engine

Reference to the main compiler engine.

Type
MainEngine

is_last_engine

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

Type
bool

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_qubits*)

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*)

Test whether a Command is supported by a compiler engine.

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._basics.**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).

receive(*command_list*)

Forward all commands to the next engine.

exception projectq.cengines._basics.**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).

_cmdmodifier

A CommandModifier engine that can be used to apply a user-defined transformation to all incoming commands.

A CommandModifier engine can be used to, e.g., modify the tags of all commands which pass by (see the AutoReplacer for an example).

class projectq.cengines._cmdmodifier.**CommandModifier**(*cmd_mod_fun*)

Compiler engine applying a user-defined transformation to all incoming commands.

CommandModifier is a compiler engine which applies a function to all incoming commands, sending on the resulting command instead of the original one.

receive(*command_list*)

Receive a list of commands.

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.

_ibm5qubitmapper

Contains a compiler engine to map to the 5-qubit IBM chip.

class projectq.cengines._ibm5qubitmapper.**IBM5QubitMapper**(*connections=None*)

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**.

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*)

Receive a list of commands.

Receive 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.

_linearmapper

Mapper for a quantum circuit to a linear chain of qubits.

Input: Quantum circuit with 1 and 2 qubit gates on n qubits. Gates are assumed to be applied in parallel if they act
on disjoint qubit(s) and any pair of qubits can perform a 2 qubit gate (all-to-all connectivity)

Output: Quantum circuit in which qubits are placed in 1-D chain in which only nearest neighbour qubits can perform a 2
qubit gate. The mapper uses Swap gates in order to move qubits next to each other.

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

Map 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`

If chain is cyclic or not

Type
Bool

`storage`

Number of gate it caches before mapping.

Type
int

`num_mappings`

Number of times the mapper changed the mapping

Type
int

`depth_of_swaps`

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

Type
dict

`num_of_swaps_per_mapping`

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

Type
dict

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.

---

is_available(*cmd*)

Only allows 1 or two qubit gates.

receive(*command_list*)

Receive a list of commands.

Receive 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*)

Build 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

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

Return 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.

_main

The main engine of every compiler engine pipeline, called MainEngine.

class projectq.cengines._main.**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

Next compiler engine (or the back-end).

Type
BasicEngine

main_engine

Self.

Type
MainEngine

active_qubits

WeakSet containing all active qubits

Type
WeakSet

dirty_qubits

Containing all dirty qubit ids

Type
Set

backend

Access the back-end.

Type
BasicEngine

mapper

Access to the mapper if there is one.

Type
BasicMapperEngine

n_engines

Current number of compiler engines in the engine list

Type
int

n_engines_max

Maximum number of compiler engines allowed in the engine list. Defaults to 100.

Type
int

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 also 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()

Return 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).

exception projectq.cengines._main.**NotYetMeasuredError**

Exception raised when trying to access the measurement value of a qubit that has not yet been measured.

exception projectq.cengines._main.**UnsupportedEngineError**

Exception raised when a non-supported compiler engine is encountered.

_manualmapper

A compiler engine to add mapping information.

class projectq.cengines._manualmapper.**ManualMapper**(*map_fun=<function ManualMapper.<lambda>>*)

Manual Mapper which adds QubitPlacementTags to Allocate gate commands according to a user-specified mapping.

map

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

Type
function

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.

_optimize

A local optimizer engine.

class projectq.cengines._optimize.**LocalOptimizer**(*cache_size=5*, *m=None*)

Circuit optimization compiler engine.

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.

receive(*command_list*)

Receive a list of commands.

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

_replacer

_swapandcnotflipper

A compiler engine which flips the directionality of CNOTs according to the given connectivity graph.

It also translates Swap gates to CNOTs if necessary.

class projectq.cengines._swapandcnotflipper.**SwapAndCNOTFlipper**(*connectivity*)

Flip 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.

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*)

Receive a list of commands.

Receive 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.

_tagremover

The TagRemover compiler engine.

A TagRemover engine removes temporary command tags (such as Compute/Uncompute), thus enabling optimization across meta statements (loops after unrolling, compute/uncompute, . . .)

class projectq.cengines._tagremover.**TagRemover**(*tags=None*)

Compiler engine that remove temporary command tags.

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)

receive(*command_list*)

Receive a list of commands.

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.

_testengine

TestEngine and DummyEngine.

class projectq.cengines._testengine.**CompareEngine**

Command list comparison compiler engine for testing purposes.

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 commands.

cache_cmd(*cmd*)

Cache a command.

is_available(*cmd*)

All commands are accepted by this compiler engine.

receive(*command_list*)

Receive a list of commands.

Receive a command list and, for each command, stores it inside the cache before sending it to the next compiler engine.

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

class projectq.cengines._testengine.**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.

is_available(*cmd*)

All commands are accepted by this compiler engine.

receive(*command_list*)

Receive a list of commands.

Receive a command list and, for each command, stores it internally if requested before sending it to the next compiler engine.

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

_twodmapper

Mapper for a quantum circuit to a 2D square grid.

Input: Quantum circuit with 1 and 2 qubit gates on n qubits. Gates are assumed to be applied in parallel if they act
on disjoint qubit(s) and any pair of qubits can perform a 2 qubit gate (all-to-all connectivity)

Output: Quantum circuit in which qubits are placed in 2-D square grid in which only nearest neighbour qubits can
perform a 2 qubit gate. The mapper uses Swap gates in order to move qubits next to each other.

class projectq.cengines._twodmapper.**GridMapper**(*num_rows*, *num_columns*, *mapped_ids_to_backend_ids=None*, *storage=1000*, *optimization_function=<function return_swap_depth>*, *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`

Number of gate it caches before mapping.

Type
int

`num_rows`

Number of rows in the grid

Type
int

`num_columns`

Number of columns in the grid

Type
int

`num_qubits`

num_rows x num_columns = number of qubits

Type
int

`num_mappings`

Number of times the mapper changed the mapping

Type
int

`depth_of_swaps`

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

Type
dict

`num_of_swaps_per_mapping`

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

Type
dict

`property current_mapping`

Access to the mapping stored inside the mapper engine.

is_available(*cmd*)

Only allow 1 or two qubit gates.

receive(*command_list*)

Receive a list of commands.

Receive 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*)

Return 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.

3.2.2 Module contents

ProjectQ module containing all compiler engines.

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

A compiler engine to automatically replace certain commands.

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

__init__(*decomposition_rule_se*, *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.

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

Next compiler engine (or the back-end).

Type
BasicEngine

main_engine

Reference to the main compiler engine.

Type
MainEngine

is_last_engine

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

Type
bool

__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_qubits*)

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*)

Test whether a Command is supported by a compiler engine.

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**

Keys are the logical qubit ids and values are the mapped qubit ids.

Type
dict

__init__()

Initialize a BasicMapperEngine object.

property current_mapping

Access the current mapping.

receive(*command_list*)

Receive a list of commands.

This implementation simply forwards all commands to the next compiler engine while adjusting the qubit IDs of measurement gates.

class projectq.cengines.**CommandModifier**(*cmd_mod_fun*)

Compiler engine applying a user-defined transformation to all incoming commands.

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.

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**

Command list comparison compiler engine for testing purposes.

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 commands.

__init__()

Initialize a CompareEngine object.

cache_cmd(*cmd*)

Cache a command.

is_available(*cmd*)

All commands are accepted by this compiler engine.

receive(*command_list*)

Receive a list of commands.

Receive a command list and, for each command, stores it inside the cache before sending it to the next compiler engine.

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

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>>*)

Initialize a DecompositionRule object.

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*)

Initialize a DecompositionRuleSet object.

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.

add_decomposition_rules(*rules*)

Add some decomposition rules to a decomposition rule set.

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 a DummyEngine.

Parameters

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

is_available(*cmd*)

All commands are accepted by this compiler engine.

receive(*command_list*)

Receive a list of commands.

Receive a command list and, for each command, stores it internally if requested before sending it to the next compiler engine.

Parameters

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

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 return_swap_depth>*, *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`

Number of gate it caches before mapping.

Type
int

`num_rows`

Number of rows in the grid

Type
int

`num_columns`

Number of columns in the grid

Type
int

`num_qubits`

num_rows x num_columns = number of qubits

Type
int

`num_mappings`

Number of times the mapper changed the mapping

Type
int

`depth_of_swaps`

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

Type
dict

`num_of_swaps_per_mapping`

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

Type
dict

`__init__`(*num_rows*, *num_columns*, *mapped_ids_to_backend_ids=None*, *storage=1000*, *optimization_function=<function return_swap_depth>*, *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

property current_mapping

Access to the mapping stored inside the mapper engine.

is_available(*cmd*)

Only allow 1 or two qubit gates.

receive(*command_list*)

Receive a list of commands.

Receive 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*)

Return 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**(*connections=None*)

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__(*connections=None*)

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*)

Receive a list of commands.

Receive 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*)

A compiler engine that implements a user-defined is_available() method.

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*)

Initialize an InstructionFilter object.

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*)

Test whether a Command is supported by a compiler engine.

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*)

Receive a list of commands.

This implementation simply forwards 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 the exception.

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

Map 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

If chain is cyclic or not

Type
Bool

storage

Number of gate it caches before mapping.

Type
int

num_mappings

Number of times the mapper changed the mapping

Type
int

depth_of_swaps

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

Type
dict

num_of_swaps_per_mapping

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

Type
dict

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*)

Receive a list of commands.

Receive 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*)

Build 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**(*cache_size=5*, *m=None*)

Circuit optimization compiler engine.

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__(*cache_size=5*, *m=None*)

Initialize a LocalOptimizer object.

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

receive(*command_list*)

Receive a list of commands.

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

Next compiler engine (or the back-end).

Type
BasicEngine

main_engine

Self.

Type
MainEngine

active_qubits

WeakSet containing all active qubits

Type
WeakSet

dirty_qubits

Containing all dirty qubit ids

Type
Set

backend

Access the back-end.

Type
BasicEngine

mapper

Access to the mapper if there is one.

Type
BasicMapperEngine

n_engines

Current number of compiler engines in the engine list

Type
int

n_engines_max

Maximum number of compiler engines allowed in the engine list. Defaults to 100.

Type
int

__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

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 also 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()

Return 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

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

Type
function

__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**

Exception raised when trying to access the measurement value of a qubit that has not yet been measured.

class projectq.cengines.**SwapAndCNOTFlipper**(*connectivity*)

Flip 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*)

Receive a list of commands.

Receive 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=None*)

Compiler engine that remove temporary command tags.

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=None*)

Initialize a TagRemover object.

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.

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**

Exception raised when a non-supported compiler engine is encountered.

projectq.cengines.**contextmanager**(*func*)

@contextmanager decorator.

Typical usage:

@contextmanager def some_generator(<arguments>):

<setup> try:

yield <value>

finally:
<cleanup>

This makes this:

with some_generator(<arguments>) as <variable>:
<body>

equivalent to this:

<setup> try:

<variable> = <value> <body>

finally:
<cleanup>

projectq.cengines.**flushing**(*engine*)

Context manager to flush the given engine at the end of the 'with' context block.

Example

with flushing(MainEngine()) as eng:
qubit = eng.allocate_qubit() ...

Calling 'eng.flush()' is no longer needed because the engine will be flushed at the end of the 'with' block even if an exception has been raised within that block.

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

Return 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

libs.math

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

projectq.libs.math._constantmath	Module containing constant math quantum operations.
projectq.libs.math._default_rules	Registers a few default replacement rules for Shor's algorithm to work (see Examples).
projectq.libs.math._gates	Quantum number math gates for ProjectQ.
projectq.libs.math._quantummath	Definition of some mathematical quantum operations.
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.AddConstantModN(a, N)	Add a constant to a quantum number represented by a quantum register modulo N.
projectq.libs.math. all_defined_decomposition_rules	Built-in mutable sequence.
projectq.libs.math. *MultiplyByConstantModN*(a, N)	Multiply a quantum number represented by a quantum register by a constant modulo N.
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.SubConstantModN(a, N)	Subtract a constant from a quantum number represented by a quantum register modulo N.

Submodules

_constantmath

Module containing constant math quantum operations.

projectq.libs.math._constantmath.**add_constant**(*eng*, *constant*, *quint*)

Add a classical constant c to the quantum integer (qureg) quint using Draper addition.

Note: Uses the Fourier-transform adder from https://arxiv.org/abs/quant-ph/0008033.

---

projectq.libs.math._constantmath.**add_constant_modN**(*eng*, *constant*, *N*, *quint*)

Add a classical constant c to a quantum integer (qureg) quint modulo N using Draper addition.

This function uses Draper addition and the construction from https://arxiv.org/abs/quant-ph/0205095.

projectq.libs.math._constantmath.**inv_mod_N**(*a*, *N*)

Calculate the inverse of a modulo N.

projectq.libs.math._constantmath.**mul_by_constant_modN**(*eng*, *constant*, *N*, *quint_in*)

Multiply a quantum integer by a classical number a modulo N.

i.e.,

|x> -> |a*x mod N>

(only works if a and N are relative primes, otherwise the modular inverse does not exist).

_default_rules

Registers a few default replacement rules for Shor's algorithm to work (see Examples).

_gates

Quantum number math gates for ProjectQ.

class projectq.libs.math._gates.**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
```

Important: if you run with conditional and carry, carry needs to be a quantum register for the compiler/decomposition to work.

get_inverse()

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

class projectq.libs.math._gates.**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
```

Note: Pre-conditions:

- c < N
- c >= 0
- The value stored in the quantum register must be lower than N

---

get_inverse()

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

class projectq.libs.math._gates.**AddQuantumGate**

Add up two quantum numbers represented by quantum registers.

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

Example

```
qunum_a = eng.allocate_qureg(5)  # 5-qubit number
qunum_b = eng.allocate_qureg(5)  # 5-qubit number
carry_bit = eng.allocate_qubit()

X | qunum_a[2]  # qunum_a is now equal to 4
X | qunum_b[3]  # qunum_b is now equal to 8
AddQuantum | (qunum_a, qunum_b, carry)
# qunum_a remains 4, qunum_b is now 12 and carry_bit is 0
```

get_inverse()

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

get_math_function(*qubits*)

Get the math function associated with an AddQuantumGate.

class projectq.libs.math._gates.**ComparatorQuantumGate**

Flip a compare qubit if the binary value of first input is higher than the second input.

The numbers are stored from low- to high-bit, i.e., qunum[0] is the LSB. .. rubric:: Example

```
qunum_a = eng.allocate_qureg(5)  # 5-qubit number
qunum_b = eng.allocate_qureg(5)  # 5-qubit number
compare_bit = eng.allocate_qubit()
X | qunum_a[4]  # qunum_a is now equal to 16
```

(continues on next page)

(continued from previous page)

```
X | qunum_b[3]  # qunum_b is now equal to 8
ComparatorQuantum | (qunum_a, qunum_b, compare_bit)
# qunum_a and qunum_b remain 16 and 8, qunum_b is now 12 and compare bit is now 1
```

get_inverse()

Return the inverse of this gate.

class projectq.libs.math._gates.**DivideQuantumGate**

Divide one quantum number from another.

Takes three inputs which should be quantum registers of equal size; a dividend, a remainder and a divisor. The remainder should be in the state |0...0> and the dividend should be bigger than the divisor.The gate returns (in this order): the remainder, the quotient and the divisor.

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

Example: .. code-block:: python

qunum_a = eng.allocate_qureg(5) # 5-qubit number qunum_b = eng.allocate_qureg(5) # 5-qubit number qunum_c = eng.allocate_qureg(5) # 5-qubit number

All(X) | [qunum_a[0], qunum_a[3]] # qunum_a is now equal to 9 X | qunum_c[2] # qunum_c is now equal to 4

DivideQuantum | (qunum_a, qunum_b, qunum_c) # qunum_a is now equal to 1 (remainder), qunum_b is now # equal to 2 (quotient) and qunum_c remains 4 (divisor)

|dividend>|remainder>|divisor> -> |remainder>|quotient>|divisor>

get_inverse()

Return the inverse of this gate.

class projectq.libs.math._gates.**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.
```

Note: Pre-conditions:

- c < N
- c >= 0
- gcd(c, N) == 1
- The value stored in the quantum register must be lower than N

---

class projectq.libs.math._gates.**MultiplyQuantumGate**

Multiply two quantum numbers represented by a quantum registers.

Requires three quantum registers as inputs, the first two are the numbers to be multiplied and should have the same size (n qubits). The third register will hold the product and should be of size 2n+1. The numbers are stored from low- to high-bit, i.e., qunum[0] is the LSB.

Example

```
qunum_a = eng.allocate_qureg(4)
qunum_b = eng.allocate_qureg(4)
qunum_c = eng.allocate_qureg(9)
X | qunum_a[2]  # qunum_a is now 4
X | qunum_b[3]  # qunum_b is now 8
MultiplyQuantum() | (qunum_a, qunum_b, qunum_c)
# qunum_a remains 4 and qunum_b remains 8, qunum_c is now equal to 32
```

get_inverse()

Return the inverse of this gate.

projectq.libs.math._gates.**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._gates.**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)
```

Note: Pre-conditions:

- c < N

- c >= 0
- The value stored in the quantum register must be lower than N

---

class projectq.libs.math._gates.**SubtractQuantumGate**

Subtract one quantum number from another quantum number both represented by quantum registers.

Example: .. code-block:: python

qunum_a = eng.allocate_qureg(5) # 5-qubit number qunum_b = eng.allocate_qureg(5) # 5-qubit number X | qunum_a[2] # qunum_a is now equal to 4 X | qunum_b[3] # qunum_b is now equal to 8 SubtractQuantum | (qunum_a, qunum_b) # qunum_a remains 4, qunum_b is now 4

get_inverse()

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

_quantummath

Definition of some mathematical quantum operations.

projectq.libs.math._quantummath.**add_quantum**(*eng*, *quint_a*, *quint_b*, *carry=None*)

Add two quantum integers.

i.e.,

|a0...a(n-1)>|b(0)...b(n-1)>|c> -> |a0...a(n-1)>|b+a(0)...b+a(n)>

(only works if quint_a and quint_b are the same size and carry is a single qubit)

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **carry** (*list*) – Carry qubit

Notes

Ancilla: 0, size: 7n-6, toffoli: 2n-1, depth: 5n-3.

References

Quantum addition using ripple carry from: https://arxiv.org/pdf/0910.2530.pdf

projectq.libs.math._quantummath.**comparator**(*eng*, *quint_a*, *quint_b*, *comp*)

Compare the size of two quantum integers.

i.e,

if a>b: |a>|b>|c> -> |a>|b>|c+1>

else: |a>|b>|c> -> |a>|b>|c>

(only works if quint_a and quint_b are the same size and the comparator is 1 qubit)

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **comp** (Qubit) – Comparator qubit

Notes

Comparator flipping a compare qubit by computing the high bit of b-a, which is 1 if and only if a > b. The high bit is computed using the first half of circuit in AddQuantum (such that the high bit is written to the carry qubit) and then undoing the first half of the circuit. By complementing b at the start and b+a at the end the high bit of b-a is calculated.

Ancilla: 0, size: 8n-3, toffoli: 2n+1, depth: 4n+3.

projectq.libs.math._quantummath.**inverse_add_quantum_carry**(*eng*, *quint_a*, *quint_b*)

Inverse of quantum addition with carry.

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)

projectq.libs.math._quantummath.**inverse_quantum_division**(*eng*, *remainder*, *quotient*, *divisor*)

Perform the inverse of a restoring integer division.

i.e.,

|remainder>|quotient>|divisor> -> |dividend>|remainder(0)>|divisor>

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **dividend** (*list*) – Quantum register (or list of qubits)
- **remainder** (*list*) – Quantum register (or list of qubits)
- **divisor** (*list*) – Quantum register (or list of qubits)

projectq.libs.math._quantummath.**inverse_quantum_multiplication**(*eng*, *quint_a*, *quint_b*, *product*)

Inverse of the multiplication of two quantum integers.

i.e,

|a>|b>|a*b> -> |a>|b>|0>

(only works if quint_a and quint_b are of the same size, n qubits and product has size 2n+1)

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **product** (*list*) – Quantum register (or list of qubits) storing the result

projectq.libs.math._quantummath.**quantum_conditional_add**(*eng*, *quint_a*, *quint_b*, *conditional*)

Add up two quantum integers if conditional is high.

i.e.,

|a>|b>|c> -> |a>|b+a>|c> (without a carry out qubit)

if conditional is low, no operation is performed, i.e., |a>|b>|c> -> |a>|b>|c>

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **conditional** (*list*) – Conditional qubit

Notes

Ancilla: 0, Size: 7n-7, Toffoli: 3n-3, Depth: 5n-3.

References

Quantum Conditional Add from https://arxiv.org/pdf/1609.01241.pdf

projectq.libs.math._quantummath.**quantum_conditional_add_carry**(*eng*, *quint_a*, *quint_b*, *ctrl*, *z*)

Add up two quantum integers if the control qubit is |1>.

i.e.,

|a>|b>|ctrl>|z(0)z(1)> -> |a>|s(0)...s(n-1)>|ctrl>|s(n)z(1)> (where s denotes the sum of a and b)

If the control qubit is |0> no operation is performed:

|a>|b>|ctrl>|z(0)z(1)> -> |a>|b>|ctrl>|z(0)z(1)>

(only works if quint_a and quint_b are of the same size, ctrl is a single qubit and z is a quantum register with 2 qubits.

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **ctrl** (*list*) – Control qubit
- **z** (*list*) – Quantum register with 2 qubits

Notes

Ancilla: 2, size: 7n - 4, toffoli: 3n + 2, depth: 5n.

References

Quantum conditional add with no input carry from: https://arxiv.org/pdf/1706.05113.pdf

projectq.libs.math._quantummath.**quantum_division**(*eng*, *dividend*, *remainder*, *divisor*)

Perform restoring integer division.

i.e.,

|dividend>|remainder>|divisor> -> |remainder>|quotient>|divisor>

(only works if all three qubits are of equal length)

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **dividend** (*list*) – Quantum register (or list of qubits)
- **remainder** (*list*) – Quantum register (or list of qubits)
- **divisor** (*list*) – Quantum register (or list of qubits)

Notes

Ancilla: n, size 16n^2 - 13, toffoli: 5n^2 -5 , depth: 10n^2-6.

References

Quantum Restoring Integer Division from: https://arxiv.org/pdf/1609.01241.pdf.

projectq.libs.math._quantummath.**quantum_multiplication**(*eng*, *quint_a*, *quint_b*, *product*)

Multiplies two quantum integers.

i.e,

|a>|b>|0> -> |a>|b>|a*b>

(only works if quint_a and quint_b are of the same size, n qubits and product has size 2n+1).

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)
- **product** (*list*) – Quantum register (or list of qubits) storing the result

Notes

Ancilla: 2n + 1, size: 7n^2 - 9n + 4, toffoli: 5n^2 - 4n, depth: 3n^2 - 2.

References

Quantum multiplication from: https://arxiv.org/abs/1706.05113.

projectq.libs.math._quantummath.**subtract_quantum**(*eng*, *quint_a*, *quint_b*)

Subtract two quantum integers.

i.e.,

|a>|b> -> |a>|b-a>

(only works if quint_a and quint_b are the same size)

Parameters

- **eng** (MainEngine) – ProjectQ MainEngine
- **quint_a** (*list*) – Quantum register (or list of qubits)
- **quint_b** (*list*) – Quantum register (or list of qubits)

Notes

Quantum subtraction using bitwise complementation of quantum adder: b-a = (a + b')'. Same as the quantum addition circuit except that the steps involving the carry qubit are left out and complement b at the start and at the end of the circuit is added.

Ancilla: 0, size: 9n-8, toffoli: 2n-2, depth: 5n-5.

References

Quantum addition using ripple carry from: https://arxiv.org/pdf/0910.2530.pdf

Module contents

Math gate definitions.

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
```

Important: if you run with conditional and carry, carry needs to be a quantum register for the compiler/decomposition to work.

__init__(*a*)

Initialize 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
```

Note: Pre-conditions:

- c < N
- c >= 0
- The value stored in the quantum register must be lower than N

__init__(*a*, *N*)

Initialize 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.
```

Note: Pre-conditions:

- c < N
- c >= 0
- gcd(c, N) == 1
- The value stored in the quantum register must be lower than N

__init__(*a*, *N*)

Initialize 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)
```

Note: Pre-conditions:

- c < N
- c >= 0
- The value stored in the quantum register must be lower than N

libs.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]

projectq.libs.revkit._control_function	RevKit support for control function oracles.
projectq.libs.revkit._permutation	RevKit support for permutation oracles.
projectq.libs.revkit._phase	RevKit support for phase oracles.
projectq.libs.revkit._utils	Module containing some utility functions.
projectq.libs.revkit.ControlFunctionOracle(...)	Synthesize a negation controlled by an arbitrary control function.
projectq.libs.revkit.PermutationOracle(...)	Synthesize a permutation using RevKit.
projectq.libs.revkit.PhaseOracle(function, ...)	Synthesize phase circuit from an arbitrary Boolean function.

Submodules

_control_function

RevKit support for control function oracles.

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

Synthesize 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)
```

_permutation

RevKit support for permutation oracles.

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

Synthesize 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)
```

_phase

RevKit support for phase oracles.

class projectq.libs.revkit._phase.**PhaseOracle**(*function*, ***kwargs*)

Synthesize 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)
```

_utils

Module containing some utility functions.

Module contents

Module containing code to interface with RevKit.

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

Synthesize 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*)

Initialize 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*)

Apply 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*)

Synthesize 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*)

Initialize 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*)

Apply 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*)

Synthesize 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*)

Initialize 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*)

Apply phase circuit to qubits (and synthesizes circuit).

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

3.3.2 Submodules

projectq.libs.hist	Measurement histogram plot helper functions.

hist

Measurement histogram plot helper functions.

Contains a function to plot measurement outcome probabilities as a histogram for the simulator

3.3.3 Module contents

ProjectQ module containing libraries expanding the basic functionalities of ProjectQ.

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._compute	Definition of Compute, Uncompute and CustomUncompute.
projectq.meta._control	Contains the tools to make an entire section of operations controlled.
projectq.meta._dagger	Tools to easily invert a sequence of gates.
projectq.meta._dirtyqubit	Define the DirtyQubitTag meta tag.
projectq.meta._exceptions	Exception classes for projectq.meta.
projectq.meta._logicalqubit	Definition of LogicalQubitIDTag to annotate a MeasureGate for mapped qubits.
projectq.meta._loop	Tools to implement loops.
projectq.meta._util	Tools to add/remove compiler engines to the MainEngine list.
projectq.meta.canonical_ctrl_state(...)	Return canonical form for control state.
projectq.meta.Compute(engine)	Start a compute-section.
projectq.meta.ComputeTag()	Compute meta tag.
projectq.meta.Control(engine, qubits[, ...])	Condition an entire code block on the value of qubits being 1.
projectq.meta.CustomUncompute(engine)	Start a custom uncompute-section.
projectq.meta.Dagger(engine)	Invert an entire code block.
projectq.meta.DirtyQubitTag()	Dirty qubit meta tag.
projectq.meta.drop_engine_after(prev_engine)	Remove an engine from the singly-linked list of engines.
projectq.meta.get_control_count(cmd)	Return the number of control qubits of the command object cmd.
projectq.meta.has_negative_control(cmd)	Return whether a command has negatively controlled qubits.
projectq.meta.insert_engine(prev_engine, ...)	Insert an engine into the singly-linked list of engines.
projectq.meta.LogicalQubitIDTag(logical_qubit_i	LogicalQubitIDTag for a mapped qubit to annotate a MeasureGate.
projectq.meta.Loop(engine, num)	Loop n times over an entire code block.
projectq.meta.LoopTag(num)	Loop meta tag.
projectq.meta.Uncompute(engine)	Uncompute automatically.
projectq.meta.UncomputeTag()	Uncompute meta tag.

3.4.1 Submodules

_compute

Definition of Compute, Uncompute and CustomUncompute.

Contains Compute, Uncompute, and CustomUncompute classes which can be used to annotate Compute / Action / Uncompute sections, facilitating the conditioning of the entire operation on the value of a qubit / register (only Action needs controls). This file also defines the corresponding meta tags.

class projectq.meta._compute.**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.

class projectq.meta._compute.**ComputeEngine**

Add Compute-tags to all commands and stores them (to later uncompute them automatically).

end_compute()

End the compute step (exit the with Compute() - statement).

Will tell the Compute-engine to stop caching. It then waits for the uncompute instruction, which is when it sends all cached commands inverted and in reverse order down to the next compiler engine.

Raises

QubitManagementError – If qubit has been deallocated in Compute section which has not been allocated in Compute section

receive(*command_list*)

Receive a list of commands.

If in compute-mode, receive commands and store deepcopy of each cmd. Add ComputeTag to received cmd and send it on. Otherwise, send all received commands directly to next_engine.

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

run_uncompute()

Send uncomputing gates.

Sends the inverse of the stored commands in reverse order down to the next engine. And also deals with allocated qubits in Compute section. If a qubit has been allocated during compute, it will be deallocated during uncompute. If a qubit has been allocated and deallocated during compute, then a new qubit is allocated and deallocated during uncompute.

class projectq.meta._compute.**ComputeTag**

Compute meta tag.

class projectq.meta._compute.**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.

exception projectq.meta._compute.**NoComputeSectionError**

Exception raised if uncompute is called but no compute section found.

projectq.meta._compute.**Uncompute**(*engine*)

Uncompute automatically.

Example

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

class projectq.meta._compute.**UncomputeEngine**

Adds Uncompute-tags to all commands.

receive(*command_list*)

Receive a list of commands.

Receive commands and add an UncomputeTag to their tags.

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

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

_control

Contains the tools to make an entire section of operations controlled.

Example

```
with Control(eng, qubit1):
    H | qubit2
    X | qubit3
```

class projectq.meta._control.**Control**(*engine*, *qubits*, *ctrl_state=CtrlAll.One*)
Condition an entire code block on the value of qubits being 1.

Example

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

class projectq.meta._control.**ControlEngine**(*qubits*, *ctrl_state=CtrlAll.One*)
Add control qubits to all commands that have no compute / uncompute tags.

receive(*command_list*)
Receive a list of commands.

projectq.meta._control.**canonical_ctrl_state**(*ctrl_state*, *num_qubits*)
Return canonical form for control state.

Parameters
- **ctrl_state** (*int*,*str*,CtrlAll) – Initial control state representation
- **num_qubits** (*int*) – number of control qubits

Returns
Canonical form of control state (currently a string composed of '0' and '1')

Note: In case of integer values for *ctrl_state*, the least significant bit applies to the first qubit in the qubit register, e.g. if ctrl_state == 2, its binary representation if '10' with the least significan bit being 0.

This means in particular that the following are equivalent:

```
canonical_ctrl_state(6, 3) == canonical_ctrl_state(6, '110')
```

---

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

projectq.meta._control.**has_negative_control**(*cmd*)

Return whether a command has negatively controlled qubits.

_dagger

Tools to easily invert a sequence of gates.

```
with Dagger(eng):
    H | qubit1
    Rz(0.5) | qubit2
```

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

Invert an entire code block.

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

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

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)
```

class projectq.meta._dagger.**DaggerEngine**

Store all commands and, when done, inverts the circuit & runs it.

receive(*command_list*)

Receive a list of commands and store them for later inversion.

Parameters

command_list (*list<Command>*) – List of commands to temporarily store.

run()

Run the stored circuit in reverse and check that local qubits have been deallocated.

_dirtyqubit

Define the DirtyQubitTag meta tag.

class projectq.meta._dirtyqubit.**DirtyQubitTag**

Dirty qubit meta tag.

_exceptions

Exception classes for projectq.meta.

exception projectq.meta._exceptions.**QubitManagementError**

Exception raised when the lifetime of a qubit is problematic within a context manager.

This may occur within Loop, Dagger or Compute regions.

_logicalqubit

Definition of LogicalQubitIDTag to annotate a MeasureGate for mapped qubits.

class projectq.meta._logicalqubit.**LogicalQubitIDTag**(*logical_qubit_id*)

LogicalQubitIDTag for a mapped qubit to annotate a MeasureGate.

logical_qubit_id

Logical qubit id

Type

int

_loop

Tools to implement loops.

Example

```
with Loop(eng, 4):
    H | qb
Rz(M_PI / 3.0) | qb
```

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

Loop n times over an entire code block.

Example

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

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
```

class projectq.meta._loop.**LoopEngine**(*num*)

A compiler engine to represent executing part of the code multiple times.

Stores all commands and, when done, executes them num times if no loop tag handler engine is available. If there is one, it adds a loop_tag to the commands and sends them on.

receive(*command_list*)

Receive (and potentially temporarily store) all commands.

Add LoopTag to all receiving commands and send to the next engine if a further engine is a LoopTag-handling engine. Otherwise store all commands (to later unroll them). Check that within the loop body, all allocated qubits have also been deallocated. If loop needs to be unrolled and ancilla qubits have been allocated within the loop body, then store a reference all these qubit ids (to change them when unrolling the loop)

Parameters

command_list (*list<Command>*) – List of commands to store and later unroll or, if there is a LoopTag-handling engine, add the LoopTag.

run()

Apply the loop statements to all stored commands.

Unrolls the loop if LoopTag is not supported by any of the following engines, i.e., if

```
is_meta_tag_supported(next_engine, LoopTag) == False
```

class projectq.meta._loop.**LoopTag**(*num*)

Loop meta tag.

loop_tag_id = 0

_util

Tools to add/remove compiler engines to the MainEngine list.

projectq.meta._util.**drop_engine_after**(*prev_engine*)

Remove 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._util.**insert_engine**(*prev_engine*, *engine_to_insert*)

Insert 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.

3.4.2 Module contents

Provides 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*, *ctrl_state=CtrlAll.One*)

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

Example

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

__init__(*engine*, *qubits*, *ctrl_state=CtrlAll.One*)

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]
    pass
```

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

Logical qubit id

Type
int

__init__(*logical_qubit_id*)

Initialize a LogicalQubitIDTag object.

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]
    pass
```

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 / 3.0) | 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 a LoopTag object.

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.**canonical_ctrl_state**(*ctrl_state*, *num_qubits*)

Return canonical form for control state.

Parameters

- **ctrl_state** (*int*, *str*, CtrlAll) – Initial control state representation
- **num_qubits** (*int*) – number of control qubits

Returns

Canonical form of control state (currently a string composed of '0' and '1')

Note: In case of integer values for *ctrl_state*, the least significant bit applies to the first qubit in the qubit register, e.g. if ctrl_state == 2, its binary representation if '10' with the least significan bit being 0.

This means in particular that the following are equivalent:

```
canonical_ctrl_state(6, 3) == canonical_ctrl_state(6, '110')
```

---

projectq.meta.**drop_engine_after**(*prev_engine*)

Remove 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.**has_negative_control**(*cmd*)

Return whether a command has negatively controlled qubits.

projectq.meta.**insert_engine**(*prev_engine*, *engine_to_insert*)

Insert 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._basics	Definitions of some of the most basic quantum gates.
projectq.ops._command	The apply_command function and the Command class.
projectq.ops._gates	Definition of the basic set of quantum gates.
projectq.ops._metagates	Definition of some *meta* gates.
projectq.ops._qaagate	Definition of the quantum amplitude amplification gate.
projectq.ops._qftgate	Definition of the QFT gate.
projectq.ops._qpegate	Definition of the quantum phase estimation gate.
projectq.ops._qubit_operator	QubitOperator stores a sum of Pauli operators acting on qubits.
projectq.ops._shortcuts	A few shortcuts for certain gates.
projectq.ops._state_prep	Definition of the state preparation gate.
projectq.ops._time_evolution	Definition of the time evolution gate.
projectq.ops._uniformly_controlled_rotation	Definition of uniformly controlled Ry- and Rz-rotation gates.
projectq.ops.All	alias of *Tensor*
projectq.ops.AllocateDirtyQubitGate()	Dirty qubit allocation gate class.
projectq.ops.AllocateQubitGate()	Qubit allocation gate class.
projectq.ops.apply_command(cmd)	Apply a command.

continues on next page

Table 2 – continued from previous page

projectq.ops.BarrierGate()	Barrier gate class.
projectq.ops.BasicGate()	Base class of all gates.
projectq.ops.BasicMathGate(math_fun)	Base class for all math gates.
projectq.ops.BasicPhaseGate(angle)	Base class for all phase gates.
projectq.ops.BasicRotationGate(angle)	Base class of for all rotation gates.
projectq.ops.C(gate[, n_qubits])	Return n-controlled version of the provided gate.
projectq.ops.ClassicalInstructionGate()	Classical instruction gate.
projectq.ops.Command(engine, gate, qubits[, ...])	Class used as a container to store commands.
projectq.ops.ControlledGate(gate[, n])	Controlled version of a gate.
projectq.ops.CRz(angle)	Shortcut for C(Rz(angle), n_qubits=1).
projectq.ops.CtrlAll(value)	Enum type to initialise the control state of qubits.
projectq.ops.DaggeredGate(gate)	Wrapper class allowing to execute the inverse of a gate, even when it does not define one.
projectq.ops.DeallocateQubitGate()	Qubit deallocation gate class.
projectq.ops.EntangleGate()	Entangle gate class.
projectq.ops.FastForwardingGate()	Base class for fast-forward gates.
projectq.ops.FlipBits(bits_to_flip)	Gate for flipping qubits by means of XGates.
projectq.ops.FlushGate()	Flush gate (denotes the end of the circuit).
projectq.ops.get_inverse(gate)	Return the inverse of a gate.
projectq.ops.HGate()	Hadamard gate class.
projectq.ops.IncompatibleControlState	Exception thrown when trying to set two incompatible states for a control qubit.
projectq.ops.is_identity(gate)	Return True if the gate is an identity gate.
projectq.ops.MatrixGate([matrix])	A gate class whose instances are defined by a matrix.
projectq.ops.MeasureGate()	Measurement gate class (for single qubits).
projectq.ops.NotInvertible	Exception thrown when trying to invert a gate which is not invertable.
projectq.ops.NotMergeable	Exception thrown when trying to merge two gates which are not mergeable.
projectq.ops.Ph(angle)	Phase gate (global phase).
projectq.ops.QAA(algorithm, oracle)	Quantum Amplitude Amplification gate.
projectq.ops.QFTGate()	Quantum Fourier Transform gate.
projectq.ops.QPE(unitary)	Quantum Phase Estimation gate.
projectq.ops.QubitOperator([term, coefficient])	A sum of terms acting on qubits, e.g., 0.5 * 'X0 X5' + 0.3 * 'Z1 Z2'.
projectq.ops.R(angle)	Phase-shift gate (equivalent to Rz up to a global phase).
projectq.ops.Rx(angle)	RotationX gate class.
projectq.ops.Rxx(angle)	RotationXX gate class.
projectq.ops.Ry(angle)	RotationY gate class.
projectq.ops.Ryy(angle)	RotationYY gate class.
projectq.ops.Rz(angle)	RotationZ gate class.
projectq.ops.Rzz(angle)	RotationZZ gate class.
projectq.ops.SelfInverseGate()	Self-inverse basic gate class.
projectq.ops.SGate()	S gate class.
projectq.ops.SqrtSwapGate()	Square-root Swap gate class.
projectq.ops.SqrtXGate()	Square-root X gate class.
projectq.ops.StatePreparation(final_state)	Gate for transforming qubits in state |0> to any desired quantum state.
projectq.ops.SwapGate()	Swap gate class (swaps 2 qubits).
projectq.ops.Tensor(gate)	Wrapper class allowing to apply a (single-qubit) gate to every qubit in a quantum register.
projectq.ops.TGate()	T gate class.

continues on next page

Table 2 – continued from previous page

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.XGate()	Pauli-X gate class.
projectq.ops.YGate()	Pauli-Y gate class.
projectq.ops.ZGate()	Pauli-Z gate class.

3.5.1 Submodules

_basics

Definitions of some of the most basic quantum gates.

Defines the BasicGate class, the base class of all gates, the BasicRotationGate class, the SelfInverseGate, the FastForwardingGate, the ClassicalInstruction gate, and the BasicMathGate class.

Gates overload the | operator to allow the following syntax:

```
Gate | (qureg1, qureg2, qureg2)
Gate | (qureg, qubit)
Gate | qureg
Gate | qubit
Gate | (qubit,)
```

This means that for more than one quantum argument (right side of |), a tuple needs to be made explicitly, while for one argument it is optional.

class projectq.ops._basics.**BasicGate**

Base class of all gates. (Don't use it directly but derive from it).

generate_command(*qubits*)

Generate a command.

The command object created consists 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

`is_identity`()

Return True if the gate is an identity gate. In this base class, always returns False.

`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>)

`to_string`(*symbols*)

Return a string representation of the object.

Achieve same function as str() but can be extended for configurable representation

`class projectq.ops._basics.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)
```

get_math_function(*qubits*)

Get the math function associated with a BasicMathGate.

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._basics.**BasicPhaseGate**(*angle*)

Base class for all phase gates.

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).

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._basics.**BasicRotationGate**(*angle*)

Base class of for all rotation gates.

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).

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.

is_identity()

Return True if the gate is equivalent to an Identity gate.

tex_str()

Return the Latex string representation of a BasicRotationGate.

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

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

to_string(*symbols=False*)

Return the string representation of a BasicRotationGate.

Parameters
symbols (*bool*) – uses the pi character and round the angle for a more user friendly display if True, full angle written in radian otherwise.

class projectq.ops._basics.**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._basics.**FastForwardingGate**

Base class for fast-forward gates.

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._basics.**MatrixGate**(*matrix=None*)

A gate class whose instances are defined by a matrix.

Note: Use this gate class only for gates acting on a small numbers of qubits. In general, consider instead using one of the provided ProjectQ gates or define a new class as this allows the compiler to work symbolically.

Example

```
gate = MatrixGate([[0, 1], [1, 0]])
gate | qubit
```

get_inverse()

Return the inverse of this gate.

property matrix

Access to the matrix property of this gate.

exception projectq.ops._basics.**NotInvertible**

Exception thrown when trying to invert a gate which is not invertable.

This exception is also thrown if the inverse is not implemented (yet).

exception projectq.ops._basics.**NotMergeable**

Exception thrown when trying to merge two gates which are not mergeable.

This exception is also thrown if the merging is not implemented (yet)).

class projectq.ops._basics.**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 of this gate.

_command

The apply_command function and the Command class.

When a gate is applied to qubits, e.g.,

```
CNOT | (qubit1, qubit2)
```

a Command object is generated which represents both the gate, qubits and control qubits. This Command object then gets sent down the compilation pipeline.

In detail, the Gate object overloads the operator| (magic method __or__) to generate a Command object which stores the qubits in a canonical order using interchangeable qubit indices defined by the gate to allow the optimizer to cancel the following two gates

```
Swap | (qubit1, qubit2)
Swap | (qubit2, qubit1)
```

The command then gets sent to the MainEngine via the apply wrapper (apply_command).

class projectq.ops._command.**Command**(*engine*, *gate*, *qubits*, *controls=()*, *tags=()*, *control_state=CtrlAll.One*)

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

add_control_qubits(*qubits*, *state=CtrlAll.One*)

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
- **state** (*int*,*str*,CtrlAll) – Control state (ie. positive or negative) for the qubits being added as control qubits.

property 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.

property control_qubits

Return a Qureg of control qubits.

property control_state

Return the state of the control qubits (ie. either positively- or negatively-controlled).

property 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.

property 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.

Example

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

is_identity()

Evaluate if the gate called in the command object is an identity gate.

Returns
True if the gate is equivalent to an Identity gate, False otherwise

property qubits

Qubits stored in a Command object.

to_string(*symbols=False*)

Get string representation of this Command object.

class projectq.ops._command.**CtrlAll**(*value*)

Enum type to initialise the control state of qubits.

One = 1

Zero = 0

exception projectq.ops._command.**IncompatibleControlState**

Exception thrown when trying to set two incompatible states for a control qubit.

projectq.ops._command.**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

_gates

Definition of the basic set of quantum gates.

Contains definitions of standard gates such as * Hadamard (H) * Pauli-X (X / NOT) * Pauli-Y (Y) * Pauli-Z (Z) * S and its inverse (S / Sdagger) * T and its inverse (T / Tdagger) * SqrtX gate (SqrtX) * Swap gate (Swap) * SqrtSwap gate (SqrtSwap) * Entangle (Entangle) * Phase gate (Ph) * Rotation-X (Rx) * Rotation-Y (Ry) * Rotation-Z (Rz) * Rotation-XX on two qubits (Rxx) * Rotation-YY on two qubits (Ryy) * Rotation-ZZ on two qubits (Rzz) * Phase-shift (R) * Measurement (Measure)

and meta gates, i.e., * Allocate / Deallocate qubits * Flush gate (end of circuit) * Barrier * FlipBits

projectq.ops._gates.**Allocate = <projectq.ops._gates.AllocateQubitGate object>**

Shortcut (instance of) *projectq.ops.AllocateQubitGate*

projectq.ops._gates.**AllocateDirty = <projectq.ops._gates.AllocateDirtyQubitGate object>**

Shortcut (instance of) *projectq.ops.AllocateDirtyQubitGate*

class projectq.ops._gates.**AllocateDirtyQubitGate**

Dirty qubit allocation gate class.

get_inverse()

Return the inverse of this gate.

class projectq.ops._gates.**AllocateQubitGate**

Qubit allocation gate class.

get_inverse()

Return the inverse of this gate.

projectq.ops._gates.**Barrier = <projectq.ops._gates.BarrierGate object>**

Shortcut (instance of) *projectq.ops.BarrierGate*

class projectq.ops._gates.**BarrierGate**

Barrier gate class.

get_inverse()

Return the inverse of this gate.

projectq.ops._gates.**Deallocate = <projectq.ops._gates.DeallocateQubitGate object>**

Shortcut (instance of) *projectq.ops.DeallocateQubitGate*

class projectq.ops._gates.**DeallocateQubitGate**

Qubit deallocation gate class.

get_inverse()

Return the inverse of this gate.

projectq.ops._gates.**Entangle = <projectq.ops._gates.EntangleGate object>**

Shortcut (instance of) *projectq.ops.EntangleGate*

class projectq.ops._gates.**EntangleGate**

Entangle gate class.

(Hadamard on first qubit, followed by CNOTs applied to all other qubits).

class projectq.ops._gates.**FlipBits**(*bits_to_flip*)

Gate for flipping qubits by means of XGates.

class projectq.ops._gates.**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*.

---

projectq.ops._gates.**H = <projectq.ops._gates.HGate object>**

Shortcut (instance of) *projectq.ops.HGate*

class projectq.ops._gates.**HGate**

Hadamard gate class.

property matrix

Access to the matrix property of this gate.

projectq.ops._gates.**Measure = <projectq.ops._gates.MeasureGate object>**

Shortcut (instance of) *projectq.ops.MeasureGate*

class projectq.ops._gates.**MeasureGate**

Measurement gate class (for single qubits).

projectq.ops._gates.**NOT = <projectq.ops._gates.XGate object>**

Shortcut (instance of) *projectq.ops.XGate*

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

Phase gate (global phase).

property matrix

Access to the matrix property of this gate.

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

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

property matrix

Access to the matrix property of this gate.

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

RotationX gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops._gates.**Rxx**(*angle*)

RotationXX gate class.

property matrix

Access to the matrix property of this gate.

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

RotationY gate class.

property **matrix**

Access to the matrix property of this gate.

class projectq.ops._gates.**Ryy**(*angle*)

RotationYY gate class.

property **matrix**

Access to the matrix property of this gate.

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

RotationZ gate class.

property **matrix**

Access to the matrix property of this gate.

class projectq.ops._gates.**Rzz**(*angle*)

RotationZZ gate class.

property **matrix**

Access to the matrix property of this gate.

projectq.ops._gates.**S = <projectq.ops._gates.SGate object>**

Shortcut (instance of) *projectq.ops.SGate*

class projectq.ops._gates.**SGate**

S gate class.

property **matrix**

Access to the matrix property of this gate.

projectq.ops._gates.**Sdag = <projectq.ops._metagates.DaggeredGate object>**

Inverse (and shortcut) of *projectq.ops.SGate*

projectq.ops._gates.**Sdagger = <projectq.ops._metagates.DaggeredGate object>**

Inverse (and shortcut) of *projectq.ops.SGate*

projectq.ops._gates.**SqrtSwap = <projectq.ops._gates.SqrtSwapGate object>**

Shortcut (instance of) *projectq.ops.SqrtSwapGate*

class projectq.ops._gates.**SqrtSwapGate**

Square-root Swap gate class.

property **matrix**

Access to the matrix property of this gate.

projectq.ops._gates.**SqrtX = <projectq.ops._gates.SqrtXGate object>**

Shortcut (instance of) *projectq.ops.SqrtXGate*

class projectq.ops._gates.**SqrtXGate**

Square-root X gate class.

property **matrix**

Access to the matrix property of this gate.

tex_str()

Return the Latex string representation of a SqrtXGate.

projectq.ops._gates.**Swap = <projectq.ops._gates.SwapGate object>**

Shortcut (instance of) *projectq.ops.SwapGate*

class projectq.ops._gates.**SwapGate**

Swap gate class (swaps 2 qubits).

property matrix

Access to the matrix property of this gate.

projectq.ops._gates.**T = <projectq.ops._gates.TGate object>**

Shortcut (instance of) *projectq.ops.TGate*

class projectq.ops._gates.**TGate**

T gate class.

property matrix

Access to the matrix property of this gate.

projectq.ops._gates.**Tdag = <projectq.ops._metagates.DaggeredGate object>**

Inverse (and shortcut) of *projectq.ops.TGate*

projectq.ops._gates.**Tdagger = <projectq.ops._metagates.DaggeredGate object>**

Inverse (and shortcut) of *projectq.ops.TGate*

projectq.ops._gates.**X = <projectq.ops._gates.XGate object>**

Shortcut (instance of) *projectq.ops.XGate*

class projectq.ops._gates.**XGate**

Pauli-X gate class.

property matrix

Access to the matrix property of this gate.

projectq.ops._gates.**Y = <projectq.ops._gates.YGate object>**

Shortcut (instance of) *projectq.ops.YGate*

class projectq.ops._gates.**YGate**

Pauli-Y gate class.

property matrix

Access to the matrix property of this gate.

projectq.ops._gates.**Z = <projectq.ops._gates.ZGate object>**

Shortcut (instance of) *projectq.ops.ZGate*

class projectq.ops._gates.**ZGate**

Pauli-Z gate class.

property matrix

Access to the matrix property of this gate.

_metagates

Definition of some *meta* gates.

Contains meta gates, i.e., * DaggeredGate (Represents the inverse of an arbitrary gate) * ControlledGate (Represents a controlled version of an arbitrary gate) * Tensor/All (Applies a single qubit gate to all supplied qubits), e.g.,

Example:

```
Tensor(H) | (qubit1, qubit2)  # apply H to qubit #1 and #2
```

As well as the meta functions * get_inverse (Tries to access the get_inverse member function of a gate and upon failure returns a DaggeredGate) * C (Creates an n-ary controlled version of an arbitrary gate)

`projectq.ops._metagates.`**`All`**

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

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

Return n-controlled version of the provided gate.

Parameters

- **gate** – Gate to turn into its controlled version
- **n_qubits** – Number of controls (default: 1)

Example

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

exception `projectq.ops._metagates.`**`ControlQubitError`**

Exception thrown when wrong number of control qubits are supplied.

class `projectq.ops._metagates.`**`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
```

---

get_inverse()

Return inverse of a controlled gate, which is the controlled inverse gate.

class projectq.ops._metagates.**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.

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._metagates.**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
```

get_inverse()

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

projectq.ops._metagates.**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)
```

projectq.ops._metagates.**is_identity**(*gate*)

Return True if the gate is an identity gate.

Tries to call gate.is_identity and, upon failure, returns False

Parameters
gate – Gate of which to get the inverse

Example

```
get_inverse(Rx(2 * math.pi))  # returns True
get_inverse(Rx(math.pi))  # returns False
```

_qaagate

Definition of the quantum amplitude amplification gate.

class projectq.ops._qaagate.**QAA**(*algorithm*, *oracle*)

Quantum Amplitude Amplification gate.

(Quick reference https://en.wikipedia.org/wiki/Amplitude_amplification. Complete reference G. Brassard, P. Hoyer, M. Mosca, A. Tapp (2000) Quantum Amplitude Amplification and Estimation https://arxiv.org/abs/quant-ph/0005055)

Quantum Amplitude Amplification (QAA) executes the algorithm, but not the final measurement required to obtain the marked state(s) with high probability. The starting state on which the QAA algorithm is executed is the one resulting of applying the Algorithm on the |0> state.

Example

```
def func_algorithm(eng, system_qubits):
    All(H) | system_qubits


def func_oracle(eng, system_qubits, qaa_ancilla):
    # This oracle selects the state |010> as the one marked
    with Compute(eng):
        All(X) | system_qubits[0::2]
    with Control(eng, system_qubits):
        X | qaa_ancilla
    Uncompute(eng)


system_qubits = eng.allocate_qureg(3)
# Prepare the qaa_ancilla qubit in the |-> state
qaa_ancilla = eng.allocate_qubit()
X | qaa_ancilla
```

(continues on next page)

(continued from previous page)

```
H | qaa_ancilla

# Creates the initial state form the Algorithm
func_algorithm(eng, system_qubits)
# Apply Quantum Amplitude Amplification the correct number of times
num_it = int(math.pi / 4.0 * math.sqrt(1 << 3))
with Loop(eng, num_it):
    QAA(func_algorithm, func_oracle) | (system_qubits, qaa_ancilla)

All(Measure) | system_qubits
```

Warning: No qubit allocation/deallocation may take place during the call to the defined Algorithm `func_algorithm`

func_algorithm
Algorithm that initialite the state and to be used in the QAA algorithm

func_oracle
The Oracle that marks the state(s) as "good"

system_qubits
the system we are interested on

qaa_ancilla
auxiliary qubit that helps to invert the amplitude of the "good" states

_qftgate

Definition of the QFT gate.

`projectq.ops._qftgate.QFT = <projectq.ops._qftgate.QFTGate object>`
Shortcut (instance of) *projectq.ops.QFTGate*

class `projectq.ops._qftgate.QFTGate`
Quantum Fourier Transform gate.

_qpegate

Definition of the quantum phase estimation gate.

class `projectq.ops._qpegate.QPE`(*unitary*)
Quantum Phase Estimation gate.

See setups.decompositions for the complete implementation

_qubit_operator

QubitOperator stores a sum of Pauli operators acting on qubits.

class projectq.ops._qubit_operator.**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:

coefficient * 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', 1.0j) | qureg  # Applies X to qubit 0 and 5 with an␣
↪additional global phase of 1.j
```

terms

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

Type
dict

compress(*abs_tol=1e-12*)

Compress the coefficient of a QubitOperator.

Eliminate 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*)

Return 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

exception projectq.ops._qubit_operator.**QubitOperatorError**

Exception raised when a QubitOperator is instantiated with some invalid data.

_shortcuts

A few shortcuts for certain gates.

These include: * CNOT = C(NOT) * CRz = C(Rz) * Toffoli = C(NOT,2) = C(CNOT)

projectq.ops._shortcuts.**CRz**(*angle*)

Shortcut for C(Rz(angle), n_qubits=1).

_state_prep

Definition of the state preparation gate.

class projectq.ops._state_prep.**StatePreparation**(*final_state*)

Gate for transforming qubits in state |0> to any desired quantum state.

_time_evolution

Definition of the time evolution gate.

exception projectq.ops._time_evolution.**NotHermitianOperatorError**

Error raised if an operator is non-hermitian.

class projectq.ops._time_evolution.**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 implicitly.

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

time t

Type
float, int

hamiltonian

hamiltonaian H

Type
QubitOperator

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.

_uniformly_controlled_rotation

Definition of uniformly controlled Ry- and Rz-rotation gates.

class projectq.ops._uniformly_controlled_rotation.**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.

get_inverse()

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

get_merged(*other*)

Return self merged with another gate.

class projectq.ops._uniformly_controlled_rotation.**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.

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

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

3.5.2 Module contents

ProjectQ module containing all basic gates (operations).

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

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

get_inverse()
Return the inverse of this gate.

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

get_inverse()
Return the inverse of this gate.

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

get_inverse()
Return the inverse of this gate.

class projectq.ops.**BasicGate**
Base class of all gates. (Don't use it directly but derive from it).

__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*)

Generate a command.

The command object created consists 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

is_identity()

Return True if the gate is an identity gate. In this base class, always returns False.

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>)

to_string(*symbols*)

Return a string representation of the object.

Achieve same function as str() but can be extended for configurable representation

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)


super().__init__(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 = 2.0**n

    def math_fun(a):
        return (int(scal * (math.sin(math.pi * a / scal))),)

    return math_fun
```

get_math_function(*qubits*)

Get the math function associated with a BasicMathGate.

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*)

Base class for all phase gates.

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.**BasicRotationGate**(*angle*)

Base class of for all rotation gates.

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.

is_identity()

Return True if the gate is equivalent to an Identity gate.

tex_str()

Return the Latex string representation of a BasicRotationGate.

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

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

to_string(*symbols=False*)

Return the string representation of a BasicRotationGate.

Parameters

symbols (*bool*) – uses the pi character and round the angle for a more user friendly display if True, full angle written in radian otherwise.

projectq.ops.**C**(*gate*, *n_qubits=1*)

Return n-controlled version of the provided gate.

Parameters

- **gate** – Gate to turn into its controlled version
- **n_qubits** – 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_qubits=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=()*, *control_state=CtrlAll.One*)

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=()*, *control_state=CtrlAll.One*)

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.
- **control_state** (*int,str,projectq.meta.CtrlAll*) –

add_control_qubits(*qubits*, *state=CtrlAll.One*)

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
- **state** (*int,str,*CtrlAll) – Control state (ie. positive or negative) for the qubits being added as control qubits.

property 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.

property control_qubits

Return a Qureg of control qubits.

property control_state

Return the state of the control qubits (ie. either positively- or negatively-controlled).

property 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.

property 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.

Example

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

is_identity()

Evaluate if the gate called in the command object is an identity gate.

Returns
True if the gate is equivalent to an Identity gate, False otherwise

property qubits

Qubits stored in a Command object.

to_string(*symbols=False*)

Get string representation of this Command object.

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.**CtrlAll**(*value*)

Enum type to initialise the control state of qubits.

One = 1

Zero = 0

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 of this gate.

class projectq.ops.**EntangleGate**

Entangle gate class.

(Hadamard on first qubit, followed by CNOTs applied to all other qubits).

class projectq.ops.**FastForwardingGate**

Base class for fast-forward gates.

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.**FlipBits**(*bits_to_flip*)

Gate for flipping qubits by means of XGates.

__init__(*bits_to_flip*)

Initialize a FlipBits gate.

Example

```
qureg = eng.allocate_qureg(2)
FlipBits([0, 1]) | qureg
```

Parameters

bits_to_flip (*list[int]|list[bool]|str|int*) – int or array of 0/1, True/False, or string of 0/1 identifying the qubits to flip. In case of int, the bits to flip are determined from the binary digits, with the least significant bit corresponding to qureg[0]. If bits_to_flip is negative, exactly all qubits which would not be flipped for the input -bits_to_flip-1 are flipped, i.e., bits_to_flip=-1 flips all qubits.

__or__(*qubits*)

Operator| overload which enables the syntax Gate | qubits.

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.

property matrix

Access to the matrix property of this gate.

exception projectq.ops.**IncompatibleControlState**

Exception thrown when trying to set two incompatible states for a control qubit.

class projectq.ops.**MatrixGate**(*matrix=None*)

A gate class whose instances are defined by a matrix.

Note: Use this gate class only for gates acting on a small numbers of qubits. In general, consider instead using one of the provided ProjectQ gates or define a new class as this allows the compiler to work symbolically.

Example

```
gate = MatrixGate([[0, 1], [1, 0]])
gate | qubit
```

__init__(*matrix=None*)

Initialize a MatrixGate object.

Parameters
matrix (*numpy.matrix*) – matrix which defines the gate. Default: None

get_inverse()

Return the inverse of this gate.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**MeasureGate**

Measurement gate class (for single qubits).

__or__(*qubits*)

Operator| overload which enables the syntax Gate | 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.

Raises
RuntimeError – Since ProjectQ v0.6.0 if the gate is applied to multiple qubits.

exception projectq.ops.**NotInvertible**

Exception thrown when trying to invert a gate which is not invertable.

This exception is also thrown if the inverse is not implemented (yet).

exception projectq.ops.**NotMergeable**

Exception thrown when trying to merge two gates which are not mergeable.

This exception is also thrown if the merging is not implemented (yet)).

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

Phase gate (global phase).

property matrix

Access to the matrix property of this gate.

class projectq.ops.**QAA**(*algorithm*, *oracle*)

Quantum Amplitude Amplification gate.

(Quick reference https://en.wikipedia.org/wiki/Amplitude_amplification. Complete reference G. Brassard, P. Hoyer, M. Mosca, A. Tapp (2000) Quantum Amplitude Amplification and Estimation https://arxiv.org/abs/quant-ph/0005055)

Quantum Amplitude Amplification (QAA) executes the algorithm, but not the final measurement required to obtain the marked state(s) with high probability. The starting state on which the QAA algorithm is executed is the one resulting of applying the Algorithm on the |0> state.

Example

```
def func_algorithm(eng, system_qubits):
    All(H) | system_qubits


def func_oracle(eng, system_qubits, qaa_ancilla):
    # This oracle selects the state |010> as the one marked
    with Compute(eng):
        All(X) | system_qubits[0::2]
    with Control(eng, system_qubits):
        X | qaa_ancilla
    Uncompute(eng)


system_qubits = eng.allocate_qureg(3)
# Prepare the qaa_ancilla qubit in the |-> state
qaa_ancilla = eng.allocate_qubit()
X | qaa_ancilla
H | qaa_ancilla

# Creates the initial state form the Algorithm
func_algorithm(eng, system_qubits)
# Apply Quantum Amplitude Amplification the correct number of times
num_it = int(math.pi / 4.0 * math.sqrt(1 << 3))
with Loop(eng, num_it):
    QAA(func_algorithm, func_oracle) | (system_qubits, qaa_ancilla)

All(Measure) | system_qubits
```

> **Warning:** No qubit allocation/deallocation may take place during the call to the defined Algorithm `func_algorithm`

func_algorithm

Algorithm that initialite the state and to be used in the QAA algorithm

func_oracle

The Oracle that marks the state(s) as "good"

system_qubits

the system we are interested on

qaa_ancilla

auxiliary qubit that helps to invert the amplitude of the "good" states

__init__(*algorithm*, *oracle*)

Initialize a QAA object.

class projectq.ops.**QFTGate**

Quantum Fourier Transform gate.

class projectq.ops.**QPE**(*unitary*)

Quantum Phase Estimation gate.

See setups.decompositions for the complete implementation

__init__(*unitary*)

Initialize a QPE 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:

coefficient * 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', 1.0j) | qureg  # Applies X to qubit 0 and 5 with an␣
↪additional global phase of 1.j
```

terms

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

Type
dict

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

Initialize a QubitOperator object.

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, empty 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 syntax Gate | qubits.

In particular, enable 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', 1.0j) | 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', 1.0j) | [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*)

Compress the coefficient of a QubitOperator.

Eliminate 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*)

Return 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).

property matrix

Access to the matrix property of this gate.

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

RotationX gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**Rxx**(*angle*)

RotationXX gate class.

property matrix

Access to the matrix property of this gate.

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

RotationY gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**Ryy**(*angle*)

RotationYY gate class.

property matrix

Access to the matrix property of this gate.

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

RotationZ gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**Rzz**(*angle*)

RotationZZ gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**SGate**

S gate class.

property matrix

Access to the matrix property of this gate.

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 of this gate.

class projectq.ops.**SqrtSwapGate**

Square-root Swap gate class.

__init__()

Initialize a SqrtSwap gate.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**SqrtXGate**

Square-root X gate class.

property matrix

Access to the matrix property of this gate.

tex_str()

Return the Latex string representation of a SqrtXGate.

class projectq.ops.**StatePreparation**(*final_state*)

Gate for transforming qubits in state |0> to any desired quantum state.

__init__(*final_state*)

Initialize a StatePreparation gate.

Example

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

Note: final_state[k] is taken to be the amplitude of the computational basis state whose string is equal to the binary representation 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 Swap gate.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**TGate**

T gate class.

property matrix

Access to the matrix property of this gate.

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*)

Operator| overload which enables the syntax Gate | 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 implicitly.

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

time t

Type
float, int

hamiltonian

hamiltonaian H

Type
QubitOperator

__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 syntax Gate | qubits.

In particular, enable 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: .. code-block:: python

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*)

Construct a UniformlyControlledRy gate.

get_inverse()

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

get_merged(*other*)

Return self merged with another gate.

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*)

Construct a UniformlyControlledRz gate.

get_inverse()

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

get_merged(*other*)

Return self merged with another gate.

class projectq.ops.**XGate**

Pauli-X gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**YGate**

Pauli-Y gate class.

property matrix

Access to the matrix property of this gate.

class projectq.ops.**ZGate**

Pauli-Z gate class.

property matrix

Access to the matrix property of this gate.

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)
```

projectq.ops.**is_identity**(*gate*)

Return True if the gate is an identity gate.

Tries to call gate.is_identity and, upon failure, returns False

Parameters
gate – Gate of which to get the inverse

Example

```
get_inverse(Rx(2 * math.pi))  # returns True
get_inverse(Rx(math.pi))  # returns False
```

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

setups.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.amplitudeamplification	Registers a decomposition for quantum amplitude amplification.
projectq.setups.decompositions.arb1qubit2rzandry	Register the Z-Y decomposition for an arbitrary one qubit gate.
projectq.setups.decompositions.barrier	Registers a decomposition rule for barriers.
projectq.setups.decompositions.carb1qubit2cnotrzandry	Register the decomposition of an controlled arbitrary single qubit gate.
projectq.setups.decompositions.cnot2cz	Registers a decomposition to for a CNOT gate in terms of CZ and Hadamard.
projectq.setups.decompositions.cnot2rxx	Register a decomposition to for a CNOT gate in terms of Rxx, Rx and Ry gates.
projectq.setups.decompositions.cnu2toffoliandcu	Register a decomposition rule for multi-controlled gates.
projectq.setups.decompositions.controlstate	Register a decomposition to replace turn negatively controlled qubits into positively controlled qubits.
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.h2rx	Register a decomposition for the H gate into an Ry and Rx gate.
projectq.setups.decompositions.ph2r	Registers a decomposition for the controlled global phase gate.
projectq.setups.decompositions.phaseestimation	Registers a decomposition for phase estimation.
projectq.setups.decompositions.qft2crandhadamard	Registers a decomposition rule for the quantum Fourier transform.
projectq.setups.decompositions.qubitop2onequbit	Register a decomposition rule for a unitary QubitOperator to one qubit gates.
projectq.setups.decompositions.r2rzandph	Registers a decomposition rule for the phase-shift gate.
projectq.setups.decompositions.rx2rz	Register a decomposition for the Rx gate into an Rz gate and Hadamard.
projectq.setups.decompositions.ry2rz	Register a decomposition for the Ry gate into an Rz and Rx(pi/2) gate.
projectq.setups.decompositions.rz2rx	Registers a decomposition for the Rz gate into an Rx and Ry(pi/2) or Ry(-pi/2) gate.
projectq.setups.decompositions.sqrtswap2cnot	Register a decomposition to achieve a SqrtSwap gate.
projectq.setups.decompositions.stateprep2cnot	Register decomposition for StatePreparation.
projectq.setups.decompositions.swap2cnot	Registers a decomposition to achieve a Swap gate.
projectq.setups.decompositions.time_evolution	Register decomposition for the TimeEvolution gates.
projectq.setups.decompositions.toffoli2cnotandtgate	Registers a decomposition rule for the Toffoli gate.
projectq.setups.decompositions.uniformlycontrolledr2cnot	Register decomposition for UnformlyControlledRy and UnformlyControlledRz.
projectq.setups.decompositions.all_defined_decomposition_rules	Built-in mutable sequence.

Submodules

amplitudeamplification

Registers a decomposition for quantum amplitude amplification.

(Quick reference https://en.wikipedia.org/wiki/Amplitude_amplification. Complete reference G. Brassard, P. Hoyer, M. Mosca, A. Tapp (2000) Quantum Amplitude Amplification and Estimation https://arxiv.org/abs/quant-ph/0005055)

Quantum Amplitude Amplification (QAA) executes the algorithm, but not the final measurement required to obtain the marked state(s) with high probability. The starting state on which the QAA algorithm is executed is the one resulting of applying the algorithm on the |0> state.

Example

```
def func_algorithm(eng, system_qubits):
    All(H) | system_qubits


def func_oracle(eng, system_qubits, qaa_ancilla):
    # This oracle selects the state |010> as the one marked
    with Compute(eng):
        All(X) | system_qubits[0::2]
    with Control(eng, system_qubits):
        X | qaa_ancilla
    Uncompute(eng)


system_qubits = eng.allocate_qureg(3)
# Prepare the qaa_ancilla qubit in the |-> state
qaa_ancilla = eng.allocate_qubit()
X | qaa_ancilla
H | qaa_ancilla

# Creates the initial state form the Algorithm
func_algorithm(eng, system_qubits)
# Apply Quantum Amplitude Amplification the correct number of times
num_it = int(math.pi / 4.0 * math.sqrt(1 << 3))
with Loop(eng, num_it):
    QAA(func_algorithm, func_oracle) | (system_qubits, qaa_ancilla)

All(Measure) | system_qubits
```

Warning: No qubit allocation/deallocation may take place during the call to the defined Algorithm `func_algorithm`

`projectq.setups.decompositions.amplitudeamplification.`**`func_algorithm`**

Algorithm that initialite the state and to be used in the QAA algorithm

`projectq.setups.decompositions.amplitudeamplification.`**`func_oracle`**

The Oracle that marks the state(s) as "good"

projectq.setups.decompositions.amplitudeamplification.**system_qubits**

the system we are interested on

projectq.setups.decompositions.amplitudeamplification.**qaa_ancilla**

auxiliary qubit that helps to invert the amplitude of the "good" states

projectq.setups.decompositions.amplitudeamplification.**all_defined_decomposition_rules = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

arb1qubit2rzandry

Register 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 = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

barrier

Registers a decomposition rule for barriers.

Deletes all barriers if they are not supported.

projectq.setups.decompositions.barrier.**all_defined_decomposition_rules = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

carb1qubit2cnotrzandry

Register the decomposition of an controlled arbitrary 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 = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

cnot2cz

Registers a decomposition to for a CNOT gate in terms of CZ and Hadamard.

projectq.setups.decompositions.cnot2cz.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

cnot2rxx

Register a decomposition to for a CNOT gate in terms of Rxx, Rx and Ry gates.

projectq.setups.decompositions.cnot2rxx.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>,
<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

cnu2toffoliandcu

Register 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** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

controlstate

Register a decomposition to replace turn negatively controlled qubits into positively controlled qubits.

This achieved by applying X gates to selected qubits.

projectq.setups.decompositions.controlstate.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

crz2cxandrz

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.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

entangle

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.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

globalphase

Registers a decomposition rule for global phases.

Deletes global phase gates (which can be ignored).

projectq.setups.decompositions.globalphase.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

h2rx

Register a decomposition for the H gate into an Ry and Rx gate.

projectq.setups.decompositions.h2rx.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>,
<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

ph2r

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._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

phaseestimation

Registers a decomposition for phase estimation.

(reference https://en.wikipedia.org/wiki/Quantum_phase_estimation_algorithm)

The Quantum Phase Estimation (QPE) executes the algorithm up to the inverse QFT included. The following steps measuring the ancillas and computing the phase should be executed outside of the QPE.

The decomposition uses as ancillas (qpe_ancillas) the first qubit/qureg in the Command and as system qubits the second qubit/qureg in the Command.

The unitary operator for which the phase estimation is estimated (unitary) is the gate in Command

Example

```
# Example using a ProjectQ gate

n_qpe_ancillas = 3
qpe_ancillas = eng.allocate_qureg(n_qpe_ancillas)
system_qubits = eng.allocate_qureg(1)
angle = cmath.pi * 2.0 * 0.125
U = Ph(angle)  # unitary_specfic_to_the_problem()

# Apply Quantum Phase Estimation
QPE(U) | (qpe_ancillas, system_qubits)

All(Measure) | qpe_ancillas
# Compute the phase from the ancilla measurement
# (https://en.wikipedia.org/wiki/Quantum_phase_estimation_algorithm)
phasebinlist = [int(q) for q in qpe_ancillas]
phase_in_bin = ''.join(str(j) for j in phasebinlist)
phase_int = int(phase_in_bin, 2)
phase = phase_int / (2**n_qpe_ancillas)
print(phase)

# Example using a function (two_qubit_gate).
# Instead of applying QPE on a gate U one could provide a function


def two_qubit_gate(system_q, time):
    CNOT | (system_q[0], system_q[1])
    Ph(2.0 * cmath.pi * (time * 0.125)) | system_q[1]
    CNOT | (system_q[0], system_q[1])


n_qpe_ancillas = 3
qpe_ancillas = eng.allocate_qureg(n_qpe_ancillas)
system_qubits = eng.allocate_qureg(2)
X | system_qubits[0]

# Apply Quantum Phase Estimation
QPE(two_qubit_gate) | (qpe_ancillas, system_qubits)

All(Measure) | qpe_ancillas
# Compute the phase from the ancilla measurement
# (https://en.wikipedia.org/wiki/Quantum_phase_estimation_algorithm)
phasebinlist = [int(q) for q in qpe_ancillas]
phase_in_bin = ''.join(str(j) for j in phasebinlist)
phase_int = int(phase_in_bin, 2)
phase = phase_int / (2**n_qpe_ancillas)
print(phase)
```

projectq.setups.decompositions.phaseestimation.**unitary**

Unitary Operation either a ProjectQ gate or a function f.

Type

BasicGate

Calling the function with the parameters system_qubits

Type
Qureg) and time (integer

i.e. f

Type
system_qubits, time

with parameter time.

projectq.setups.decompositions.phaseestimation.**all_defined_decomposition_rules = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

qft2crandhadamard

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 = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

qubitop2onequbit

Register a decomposition rule for a unitary QubitOperator to one qubit gates.

projectq.setups.decompositions.qubitop2onequbit.**all_defined_decomposition_rules = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

r2rzandph

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.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

rx2rz

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

projectq.setups.decompositions.rx2rz.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

ry2rz

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

projectq.setups.decompositions.ry2rz.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

rz2rx

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

projectq.setups.decompositions.rz2rx.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>,
<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

sqrtswap2cnot

Register a decomposition to achieve a SqrtSwap gate.

projectq.setups.decompositions.sqrtswap2cnot.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

stateprep2cnot

Register decomposition for StatePreparation.

projectq.setups.decompositions.stateprep2cnot.**all_defined_decomposition_rules** =
[<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]

Decomposition rules

swap2cnot

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.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

time_evolution

Register 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.cengines._replacer._decomposition_rule.DecompositionRule object>, <projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

toffoli2cnotandtgate

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 = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

uniformlycontrolledr2cnot

Register decomposition for UnformlyControlledRy and UnformlyControlledRz.

projectq.setups.decompositions.uniformlycontrolledr2cnot.**all_defined_decomposition_rules = [<projectq.cengines._replacer._decomposition_rule.DecompositionRule object>, <projectq.cengines._replacer._decomposition_rule.DecompositionRule object>]**

Decomposition rules

Module contents

ProjectQ's decomposition rules.

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._utils	Some utility functions common to some setups.
projectq.setups.aqt	A setup for AQT trapped ion devices.
projectq.setups.default	The default setup which provides an *engine_list* for the *MainEngine*.
projectq.setups.grid	A setup to compile to qubits placed in 2-D grid.
projectq.setups.ibm	A setup for IBM quantum chips.
projectq.setups.ionq	A setup for IonQ trapped ion devices.
projectq.setups.linear	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.
projectq.setups.trapped_ion_decomposer	Apply the restricted gate set setup for trapped ion based quantum computers.

_utils

Some utility functions common to some setups.

projectq.setups._utils.**get_engine_list_linear_grid_base**(*mapper*, *one_qubit_gates='any'*, *two_qubit_gates=(<projectq.ops._metagates.ControlledGate object>*, *<projectq.ops._gates.SwapGate object>)*)

Return 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 implementation 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._utils.**high_level_gates**(*eng*, *cmd*)

Remove any MathGates.

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

Filter out 1- and 2-qubit gates.

aqt

A setup for AQT trapped ion devices.

Defines a setup allowing to compile code for the AQT trapped ion devices: ->The 4 qubits device ->The 11 qubits simulator ->The 11 qubits noisy simulator

It provides the *engine_list* for the `MainEngine' based on the requested device. Decompose the circuit into a Rx/Ry/Rxx gate set that will be translated in the backend in the Rx/Ry/MS gate set.

projectq.setups.aqt.**get_engine_list**(*token=None*, *device=None*)

Return the default list of compiler engine for the AQT platform.

default

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**()

Return the default list of compiler engine.

grid

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>)*)

Return 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 implementation 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.

ibm

A setup for IBM quantum chips.

Defines a setup allowing to compile code for the IBM quantum chips: * Any 5 qubit devices * the ibmq online simulator * the melbourne 15 qubit device

It provides the *engine_list* for the `MainEngine' based on the requested device.

Decompose the circuit into a Rx/Ry/Rz/H/CNOT gate set that will be translated in the backend in the U1/U2/U3/CX gate set.

projectq.setups.ibm.**get_engine_list**(*token=None*, *device=None*)
Return the default list of compiler engine for the IBM QE platform.

projectq.setups.ibm.**list2set**(*coupling_list*)
Convert a list() to a set().

ionq

A setup for IonQ trapped ion devices.

Defines a setup allowing to compile code for IonQ trapped ion devices: ->The 11 qubit device ->The 29 qubits simulator

projectq.setups.ionq.**get_engine_list**(*token=None*, *device=None*)

Return the default list of compiler engine for the IonQ platform.

linear

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>)*)

Return 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 implementation 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.

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 limitations on the choice of gates).

projectq.setups.restrictedgateset.**default_chooser**(*cmd*, *decomposition_list*)

Provide the default chooser function for the AutoReplacer compiler engine.

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

Return 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 implementation 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.
- **compiler_chooser** – function selecting the decomposition to use in the Autoreplacer engine

Raises
TypeError – If input is for the gates is not "any" or a tuple. Also if element within tuple is not a class or instance of BasicGate (e.g. CRz which is a shortcut function)

Returns
A list of suitable compiler engines.

trapped_ion_decomposer

Apply the restricted gate set setup for trapped ion based quantum computers.

It provides the *engine_list* for the *MainEngine*, restricting the gate set to Rx and Ry single qubit gates and the Rxx two qubit gates.

A decomposition chooser is implemented following the ideas in QUOTE for reducing the number of Ry gates in the new circuit.

Note: Because the decomposition chooser is only called when a gate has to be decomposed, this reduction will work better when the entire circuit has to be decomposed. Otherwise, If the circuit has both superconding gates and native ion trapped gates the decomposed circuit will not be optimal.

`projectq.setups.trapped_ion_decomposer.`**`chooser_Ry_reducer`**`(`*cmd*, *decomposition_list*`)`

Choose the decomposition to maximise Ry cancellations.

Choose the decomposition so as to maximise Ry cancellations, based on the previous decomposition used for the given qubit.

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

Returns
A decomposition object from the decomposition_list.

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

Return an engine list compiling code into a trapped ion based compiled circuit code.

Note:

- Classical instructions gates such as e.g. Flush and Measure are automatically allowed.
- The restricted gate set engine does not work with Rxx gates, as ProjectQ will by default bounce back and forth between Cz gates and Cx gates. An appropriate decomposition chooser needs to be used!

Returns
A list of suitable compiler engines.

3.6.3 Module contents

ProjectQ module containing the basic setups for ProjectQ as well as the decomposition rules.

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._qubit	Definition of BasicQubit, Qubit, WeakQubit and Qureg classes.
projectq.types.BasicQubit(engine, idx)	BasicQubit objects represent qubits.
projectq.types.Qubit(engine, idx)	Qubit class.
projectq.types.Qureg([iterable])	Quantum register class.
projectq.types.WeakQubitRef(engine, idx)	WeakQubitRef objects are used inside the Command object.

3.7.1 Submodules

_qubit

Definition of BasicQubit, Qubit, WeakQubit and Qureg classes.

A Qureg represents a list of Qubit or WeakQubit objects. A Qubit represents a (logical-level) qubit with a unique index provided by the MainEngine. Qubit objects are automatically deallocated if they go out of scope and intended to be used within Qureg objects in user code.

Example

```
from projectq import MainEngine

eng = MainEngine()
qubit = eng.allocate_qubit()
```

qubit is a Qureg of size 1 with one Qubit object which is deallocated once qubit goes out of scope.

WeakQubit are used inside the Command object and are not automatically deallocated.

class projectq.types._qubit.**BasicQubit**(*engine*, *idx*)

BasicQubit objects represent qubits.

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

class projectq.types._qubit.**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.

class projectq.types._qubit.**Qureg**(*iterable=()*, */*)

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)).

property engine

Return owning engine.

class projectq.types._qubit.**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.

3.7.2 Module contents

ProjectQ module containing all basic types.

class projectq.types.**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.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.

class projectq.types.**Qureg**(*iterable=()*, */*)

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)).

property 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

J

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z