MQT Qudits

A Framework For Mixed-Dimensional Qudit Quantum Computing

Version 0.1.1.dev3

Chair for Design Automation
Technical University of Munich, Germany
quantum.cda@xcit.tum.de

May 13, 2024

MQT Qudits is an open-source C++17 and Python framework for mixed-dimensional qudit quantum computing developed as part of the Munich Quantum Toolkit (MQT) by the Chair for Design Automation at the Technical University of Munich.

Note: The tool is in an experimental stage, which is subject to frequent changes, and has limited documentation. We are working on improving that. In the meantime, users can explore how to use the framework via the *Tutorial*, showcasing its main functionality.

Furthermore, this video briefly illustrates some of the functionalities of MQT Qudits.

We appreciate any feedback and contributions to the project. If you have any questions, feel free to create a discussion or an issue on GitHub.

I MQT Qudits Tutorial

Discover a New Dimension in Quantum Computing. Embark on a journey with MQT Qudits, a framework for Mixed-Dimensional Quantum Computing.

Delve into the realm of mixed-dimensional quantum computing with NeQST—a project funded by the European Union and developed developed as part of the Munich Quantum Toolkit (MQT) by the Chair for Design Automation at the Technical University of Munich. Our team is focused on creating design automation methods and software for quantum computing. The following tutorial will guide you through the initial tools and contributions we have made to advance Quantum Information Processing for Science and Technology.

I-A Installation Steps:

```
(.venv) $ pip install mqt.qudits
```

For those seeking hands-on customization, simply clone the corresponding repository and perform a local installation.

```
$ git clone https://github.com/cda-tum/mqt-qudits.git
$ cd mqt-qudits
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install -ve .
+++

```{note}
This requires a C++17 compiler, a minimum CMake version of 3.19, and Python 3.8+.
```

I-B User Inputs

```python
import numpy as np

from mqt.qudits.quantum_circuit import QuantumCircuit
```

New QASM Extension:

Dive into a language meticulously designed to express quantum algorithms and circuits. MQT Qudits extends the OpenQASM 2.0 grammar, effortlessly adapting to mixed-dimensional registers. In the following, a **DITQASM** program is explicitly written, although several methods for importing programs from files are present in the library.

```python
qasm = """
    DITQASM 2.0;

    qreg field [7][5,5,5,5,5,5,5];
    qreg matter [2];

    creg meas_matter[7];
    creg meas_fields[3];

    h matter[0] ctl field[0] field[1] [0,0];
    cx field[2], matter[0];
    cx field[2], matter[1];
    rxy (0, 1, pi, pi/2) field[3];

    measure q[0] -> meas[0];
    measure q[1] -> meas[1];
    measure q[2] -> meas[2];
    """
```

A new feature is the **control syntax**:

```
_operation_   __ctl__   _quditline_   [list of qudit control levels]
```

We can import the DITQASM program and construct a quantum circuit.

```python
circuit = QuantumCircuit()
circuit.from_qasm(qasm)

print(f"Number of operations: {len(circuit.instructions)}")
print(f"Number of qudits in the circuit: {circuit.num_qudits}")
print(f"Dimensions: {circuit.dimensions}")
```
```

```
Number of operations: 4
Number of qudits in the circuit: 9
Dimensions: [5, 5, 5, 5, 5, 5, 5, 2, 2]
```

Python Interface

Constructing and manipulating quantum programs becomes a breeze with Python. You have the flexibility to:

1. **Initialize Quantum Circuits:** Start by creating your quantum circuits effortlessly.
2. **Create Quantum Registers:** Build dedicated quantum registers tailored to your needs.
3. **Compose Circuits:** Seamlessly bring together your quantum registers, forming a unified and powerful circuit.
4. **Apply Operations:** Easily apply a variety of qudit operations, without worrying about the right representation.

Let's construct a quantum circuit from scratch, with the python interface.

```python
from mqt.qudits.quantum_circuit import QuantumRegister

circuit = QuantumCircuit()

field_reg = QuantumRegister("fields", 1, [7])
matter_reg = QuantumRegister("matter", 1, [2])

circuit.append(field_reg)
circuit.append(matter_reg)

print(f"Number of operations: {len(circuit.instructions)}")
print(f"Number of qudits in the circuit: {circuit.num_qudits}")
print(f"Gate set: {circuit.gate_set}")
```

```
Number of operations: 0
Number of qudits in the circuit: 2
csum
cu_one
cu_two
cu_multi
cx
gellmann
h
ls
ms
pm
r
rh
randu
rz
virtrz
s
x
z
Gate set: None
```

No operations were inserted yet, let's take a look at how operations can be applied!

The size of every line is detected automatically and the right operations are applied to the right qudits

```python
circuit.h(field_reg[0])
circuit.csum([field_reg[0], matter_reg[0]])

print(f"Number of operations: {len(circuit.instructions)}")
print(f"Number of qudits in the circuit: {circuit.num_qudits}")
```

```
Number of operations: 2
Number of qudits in the circuit: 2
```

It is possible to export the code as well and share your program in a QASM file.

```
print(circuit.to_qasm())
```

```
DITQASM 2.0;
qreg fields [1][7];
qreg matter [1][2];
creg meas[2];
h fields[0];
csum fields[0], matter[0];
measure fields[0] -> meas[0];
measure matter[0] -> meas[1];
```

Let's save the circuit to a file

```
file = circuit.save_to_file("my_circuit", ".")
```

and load it back

```
circuit.load_from_file(file)

print("Program:", circuit.to_qasm(), sep="\n")
print("Dimensions: ", circuit.dimensions)
```

```
Program:
DITQASM 2.0;
qreg fields [1][7];
qreg matter [1][2];
creg meas[2];
h fields[0];
csum fields[0], matter[0];
measure fields[0] -> meas[0];
measure matter[0] -> meas[1];

Dimensions:  [7, 2]
```

Custom gates can be added to the circuit as well.

```
n = 5
random_matrix = np.random.randn(n, n) + 1j * np.random.randn(n, n)

Q, R = np.linalg.qr(random_matrix)

unitary_matrix = Q
cu = circuit.cu_one(field_reg[0], unitary_matrix)
```

Gates follow the order:

- target qudit/s : list or single number
- parameters list with order lower level, upper level, control level, theta, phi
- control data

A simple qudit gate can be added as follows:

```
r = circuit.r(field_reg[0], [0, 1, np.pi / 5, np.pi / 7])
```

Operations can also be controlled by other qudits, as shown below:

```
from mqt.qudits.quantum_circuit.gate import ControlData

r_c1 = circuit.r(field_reg[0], [0, 1, np.pi / 5, np.pi / 7], ControlData([matter_reg[0]], [1]))
```

or as

```
r_c2 = circuit.r(field_reg[0], [0, 1, np.pi / 5, np.pi / 7]).control([matter_reg[0]], [1])
```

The representation of the matrix corresponding to a gate is dynamic:

- 0: no identities
- 1: identities in between long-range gates are introduced
- 2: full circuit unitary

```
print(f"Gate matrix for {r._name}:", r.to_matrix(0), sep="\n")
```

```
Gate matrix for R7:
[[ 0.95105652+0.j         -0.13407745-0.27841469j  0.        +0.j
   0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j        ]
 [ 0.13407745-0.27841469j  0.95105652+0.j          0.        +0.j
   0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j        ]
 [ 0.        +0.j          0.        +0.j          1.        +0.j
   0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j        ]
 [ 0.        +0.j          0.        +0.j          0.        +0.j
   1.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j        ]
 [ 0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j          1.        +0.j          0.        +0.j
   0.        +0.j        ]
 [ 0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j          0.        +0.j          1.        +0.j
   0.        +0.j        ]
 [ 0.        +0.j          0.        +0.j          0.        +0.j
   0.        +0.j          0.        +0.j          0.        +0.j
   1.        +0.j        ]]
```

The inverse of any gate can easily be obtained.

```
rd = r.dag()
print(f"Inverse gate matrix for {r._name}:", rd.to_matrix(0), sep="\n")
```

```
Inverse gate matrix for R7_dag:
[[ 0.95105652-0.j          0.13407745+0.27841469j  0.        -0.j
   0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j        ]
 [-0.13407745+0.27841469j  0.95105652-0.j          0.        -0.j
   0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j        ]
 [ 0.        -0.j          0.        -0.j          1.        -0.j
   0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j        ]
 [ 0.        -0.j          0.        -0.j          0.        -0.j
   1.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j        ]
 [ 0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j          1.        -0.j          0.        -0.j
   0.        -0.j        ]
 [ 0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j          0.        -0.j          1.        -0.j
   0.        -0.j        ]
 [ 0.        -0.j          0.        -0.j          0.        -0.j
   0.        -0.j          0.        -0.j          0.        -0.j
   1.        -0.j        ]]
```

The control information can be accessed as well.

```
r_c1.control_info
```

```
{'target': 0,
 'dimensions_slice': 7,
 'params': [0, 1, 0.6283185307179586, 0.4487989505128276],
 'controls': ControlData(indices=[1], ctrl_states=[1])}
```

Two- and multi-qudit gates follow the rule:

- two : target_qudits first is control, second is target
- multi: all are controls, except last one is target

```
r_c1.reference_lines
```

```
[1, 0]
```

I-C Simulation

After crafting your quantum circuit with precision, take it for a spin using two distinct engines, each flaunting its unique set of data structures.

- **External Tensor-Network Simulator:** Delve into the quantum realm with a robust external tensor-network simulator. Can simulate all the gate-set.
- **MiSiM (C++-Powered):** Unleash the power of decision-diagram-based simulation with MiSiM, seamlessly interfaced with Python for a fluid and efficient experience. Can only simulate the machine following machine gate set:
  - csum
  - cx
  - h
  - rxy
  - rz
  - rh
  - virtrz
  - s
  - x
  - z

Basic Simulation

```
circuit = QuantumCircuit()

field_reg = QuantumRegister("fields", 1, [3])
matter_reg = QuantumRegister("matter", 1, [3])

circuit.append(field_reg)
circuit.append(matter_reg)

h = circuit.h(field_reg[0])
csum = circuit.csum([field_reg[0], matter_reg[0]])

print(f"Number of operations: {len(circuit.instructions)}")
print(f"Number of qudits in the circuit: {circuit.num_qudits}")
```

```
Number of operations: 2
Number of qudits in the circuit: 2
```

```
from mqt.qudits.simulation import MQTQuditProvider

provider = MQTQuditProvider()
provider.backends("sim")
```

```
['tnsim', 'misim']
```

```python
from mqt.qudits.visualisation import plot_counts, plot_state

backend = provider.get_backend("tnsim")

job = backend.run(circuit)
result = job.result()

state_vector = result.get_state_vector()

plot_state(state_vector, circuit)
```

```python
backend = provider.get_backend("misim")

job = backend.run(circuit)
result = job.result()

state_vector = result.get_state_vector()

plot_state(state_vector, circuit)
```

Extending Engines with Noise Model and Properties for FakeBackend

Enhance your quantum simulation experience by extending the engines with a noise model and incorporating various properties. By combining a noise model and carefully tuned properties, you can craft a FakeBackend that closely emulates the performance of the best quantum machines in experimental laboratories. This allows for more realistic and insightful quantum simulations.

Experiment, iterate, and simulate quantum circuits with the sophistication of real-world conditions, all within the controlled environment of your simulation.

```python
from mqt.qudits.simulation.noise_tools.noise import Noise, NoiseModel

# Depolarizing quantum errors
local_error = Noise(probability_depolarizing=0.001, probability_dephasing=0.001)
local_error_rz = Noise(probability_depolarizing=0.03, probability_dephasing=0.03)

entangling_error = Noise(probability_depolarizing=0.1, probability_dephasing=0.001)
entangling_error_extra = Noise(probability_depolarizing=0.1, probability_dephasing=0.1)

entangling_error_on_target = Noise(probability_depolarizing=0.1, probability_dephasing=0.0)
entangling_error_on_control = Noise(probability_depolarizing=0.01, probability_dephasing=0.0)

# Add errors to noise_tools model

noise_model = NoiseModel()  # We know that the architecture is only two qudits
# Very noisy gate
noise_model.add_all_qudit_quantum_error(local_error, ["csum"])
noise_model.add_recurrent_quantum_error_locally(local_error, ["csum"], [0])
# Entangling gates
noise_model.add_nonlocal_quantum_error(entangling_error, ["cx", "ls", "ms"])
noise_model.add_nonlocal_quantum_error_on_target(entangling_error_on_target, ["cx", "ls",
↪"ms"])
noise_model.add_nonlocal_quantum_error_on_control(entangling_error_on_control, ["csum", "cx",
↪"ls", "ms"])
# Super noisy Entangling gates
noise_model.add_nonlocal_quantum_error(entangling_error_extra, ["csum"])
# Local Gates
noise_model.add_quantum_error_locally(local_error, ["h", "rxy", "s", "x", "z"])
noise_model.add_quantum_error_locally(local_error_rz, ["rz", "virtrz"])

print(noise_model.quantum_errors)
```

```
{'csum': {'all': Noise(probability_depolarizing=0.001, probability_dephasing=0.001), (0,):
Noise(probability_depolarizing=0.001, probability_dephasing=0.001), 'control':
Noise(probability_depolarizing=0.01, probability_dephasing=0.0), 'nonlocal':
Noise(probability_depolarizing=0.1, probability_dephasing=0.1)}, 'cx': {'nonlocal':
Noise(probability_depolarizing=0.1, probability_dephasing=0.001), 'target':
Noise(probability_depolarizing=0.1, probability_dephasing=0.0), 'control': Noise(probability_
depolarizing=0.01, probability_dephasing=0.0)}, 'ls': {'nonlocal': Noise(probability_
depolarizing=0.1, probability_dephasing=0.001), 'target': Noise(probability_depolarizing=0.1,
 probability_dephasing=0.0), 'control': Noise(probability_depolarizing=0.01, probability_
dephasing=0.0)}, 'ms': {'nonlocal': Noise(probability_depolarizing=0.1, probability_
dephasing=0.001), 'target': Noise(probability_depolarizing=0.1, probability_dephasing=0.0),
'control': Noise(probability_depolarizing=0.01, probability_dephasing=0.0)}, 'h': {'local':
Noise(probability_depolarizing=0.001, probability_dephasing=0.001)}, 'rxy': {'local':
Noise(probability_depolarizing=0.001, probability_dephasing=0.001)}, 's': {'local':
Noise(probability_depolarizing=0.001, probability_dephasing=0.001)}, 'x': {'local':
Noise(probability_depolarizing=0.001, probability_dephasing=0.001)}, 'z': {'local':
Noise(probability_depolarizing=0.001, probability_dephasing=0.001)}, 'rz': {'local':
Noise(probability_depolarizing=0.03, probability_dephasing=0.03)}, 'virtrz': {'local':
Noise(probability_depolarizing=0.03, probability_dephasing=0.03)}}
```

We can set the noise model for the simulation, but also set several other flags:

- `shots`: number of shots for the stochastic simulation
- `memory`: flag for saving shots (True/False)
- `full_state_memory`: save the full noisy states
- `file_path`: file path of the h5 database storing the data
- `file_name`: name of the file

```
backend = provider.get_backend("tnsim")

job = backend.run(circuit, noise_model=noise_model)

result = job.result()
counts = result.get_counts()

plot_counts(counts, circuit)
```

```
{'00': 332,
 '01': 0,
 '02': 3,
 '10': 4,
 '11': 329,
 '12': 0,
 '20': 0,
 '21': 7,
 '22': 325}
```

You can also invoke a fake backend and retrieve a few relevant properties, that are already embedded in them

```
provider = MQTQuditProvider()
provider.backends("fake")
```

```
['faketraps2trits', 'faketraps2six']
```

```
backend_ion = provider.get_backend("faketraps2trits", shots=1000)
```

```python
import matplotlib.pyplot as plt
import networkx as nx

mapping = backend_ion.energy_level_graphs

pos = nx.circular_layout(mapping[0])
nx.draw(mapping[0], pos, with_labels=True, node_size=2000, node_color="lightblue", font_
↪size=12, font_weight="bold")
plt.show()
```

```python
job = backend_ion.run(circuit)
result = job.result()
counts = result.get_counts()

plot_counts(counts, circuit)
```

```
{'00': 328,
 '01': 0,
 '02': 1,
 '10': 2,
 '11': 322,
 '12': 0,
 '20': 0,
 '21': 2,
 '22': 345}
```

I-D Compilation

Tailor your quantum compilation process to achieve optimal performance and emulate the intricacies of experimental setups.

Compiler Customization with Modern Passes

1. **Optimization Strategies:** Implement specific optimization strategies based on your quantum algorithm's characteristics. Fine-tune compilation for better resource utilization and reduced gate counts.
2. **Gate Decomposition:** Customize gate decomposition techniques to match the capabilities of experimental quantum hardware. Aligning with the native gate set enhances the efficiency of your compiled circuits.

Experimental-Inspired Compilation

Emulate the features of the best experimental laboratories in your compilation process. Leverage modern compiler passes to customize optimization, gate decomposition, and noise-aware strategies, creating compiled circuits that closely resemble the challenges and advantages of cutting-edge quantum hardware.

Customize, compile, and push the boundaries of quantum algorithms with a tailored approach to quantum compilation.

```python
from mqt.qudits.compiler import QuditCompiler
```

```python
qudit_compiler = QuditCompiler()

passes = ["PhyLocQRPass"]
```

```python
compiled_circuit_qr = qudit_compiler.compile(backend_ion, circuit, passes)

print(f"Number of operations: {len(compiled_circuit_qr.instructions)}")
print(f"Number of qudits in the circuit: {compiled_circuit_qr.num_qudits}")
```

```
Number of operations: 10
Number of qudits in the circuit: 2
```

```
job = backend_ion.run(compiled_circuit_qr)

result = job.result()
counts = result.get_counts()

plot_counts(counts, compiled_circuit_qr)
```

```
{'00': 326,
 '01': 0,
 '02': 4,
 '10': 0,
 '11': 326,
 '12': 0,
 '20': 0,
 '21': 2,
 '22': 342}
```

```
passes = ["PhyLocAdaPass", "ZPropagationPass", "ZRemovalPass"]

compiled_circuit_ada = qudit_compiler.compile(backend_ion, circuit, passes)

print(f"Number of operations: {len(compiled_circuit_ada.instructions)}")
print(f"Number of qudits in the circuit: {compiled_circuit_ada.num_qudits}")
```

```
Number of operations: 5
Number of qudits in the circuit: 2
```

```
job = backend_ion.run(compiled_circuit_ada)

result = job.result()
counts = result.get_counts()

plot_counts(counts, compiled_circuit_ada)
```

```
{'00': 321,
 '01': 0,
 '02': 4,
 '10': 2,
 '11': 328,
 '12': 0,
 '20': 0,
 '21': 3,
 '22': 342}
```

```
from mqt.qudits.visualisation import draw_qudit_local

draw_qudit_local(compiled_circuit_ada)
```

```
|0>-----[R01(1.57,-2.09)]----[R02(1.23,-2.62)]----[R02(3.14,-2.62)]----[R01(1.57,-0.52)]----MG-
↪----=||
|0>-----MG----MG----MG----MG----MG-----=||
```

II Publications

MQT Qudits is academic software. Thus, many of its built-in algorithms have been published as scientific papers [1, 2, 3, 4, 5].

If you use *MQT Qudits* in your work, we would appreciate if you cited the respective paper(s).

III mqt.qudits

MQT Qudits - A framework for mixed-dimensional qudit quantum computing.

III-A Subpackages

mqt.qudits.compiler

Subpackages

`mqt.qudits.compiler.compilation_minitools` Common utilities for compilation.

Submodules

`mqt.qudits.compiler.compilation_minitools.local_compilation_minitools`

Module Contents

swap_elements(*list_nodes*, *i*, *j*)

new_mod(*a*, *b=2 * np.pi*)

pi_mod(*a*)

regulate_theta(*angle*)

phi_cost(*theta*)

theta_cost(*theta*)

rotation_cost_calc(*gate*, *placement*)

`mqt.qudits.compiler.compilation_minitools.naive_unitary_verifier`

Module Contents

class UnitaryVerifier(*sequence*, *target*, *dimensions*, *nodes=None*, *initial_map=None*, *final_map=None*)

Verifies unitary matrices. sequence is a list of numpy arrays target is a numpy array dimensions is list of ints, equals to the dimensions of the qudits involved in the target operation initial_map is a list representing the mapping of the logic states to the physical ones at the beginning of the computation final_map is a list representing the mapping of the logic states to the physical ones at the end of the computation

get_perm_matrix(*nodes*, *mapping*)

verify()

`mqt.qudits.compiler.compilation_minitools.numerical_ansatz_utils`

Module Contents

on1(*gate*, *other_size*)

on0(*gate*, *other_size*)

gate_expand_to_circuit(*gate*, *circuits_size*, *target*, *dims=None*)

apply_gate_to_tlines(*gate_matrix*, *circuits_size=2*, *targets=None*, *dims=None*)

Package Contents

new_mod(*a*, *b=2 * np.pi*)

phi_cost(*theta*)

pi_mod(*a*)

regulate_theta(*angle*)

rotation_cost_calc(*gate*, *placement*)

swap_elements(*list_nodes*, *i*, *j*)

theta_cost(*theta*)

class **UnitaryVerifier**(*sequence*, *target*, *dimensions*, *nodes=None*, *initial_map=None*, *final_map=None*)

Verifies unitary matrices. sequence is a list of numpy arrays target is a numpy array dimensions is list of ints, equals to the dimensions of the qudits involved in the target operation initial_map is a list representing the mapping of the logic states to the physical ones at the beginning of the computation final_map is a list representing the mapping of the logic states to the physical ones at the end of the computation

get_perm_matrix(*nodes*, *mapping*)

verify()

apply_gate_to_tlines(*gate_matrix*, *circuits_size=2*, *targets=None*, *dims=None*)

gate_expand_to_circuit(*gate*, *circuits_size*, *target*, *dims=None*)

on0(*gate*, *other_size*)

on1(*gate*, *other_size*)

`mqt.qudits.compiler.onedit`

Subpackages

`mqt.qudits.compiler.onedit.local_operation_swap`

Submodules

`mqt.qudits.compiler.onedit.local_operation_swap.swap_routine`

Module Contents

find_logic_from_phys(*lev_a*, *lev_b*, *graph*)

graph_rule_update(*gate*, *graph*) → None

graph_rule_ongate(*gate*, *graph*) → *R*

gate_chain_condition(*previous_gates*, *current*)

route_states2rotate_basic(*gate*, *orig_placement*)

cost_calculator(*gate*, *placement*, *non_zeros*)

Package Contents

cost_calculator(*gate*, *placement*, *non_zeros*)

gate_chain_condition(*previous_gates*, *current*)

graph_rule_ongate(*gate*, *graph*) → *R*

graph_rule_update(*gate*, *graph*) → None

route_states2rotate_basic(*gate*, *orig_placement*)

mqt.qudits.compiler.onedit.local_phases_transpilation

Submodules

mqt.qudits.compiler.onedit.local_phases_transpilation.propagate_virtrz

Module Contents

class ZPropagationPass(*backend*, *back=True*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

propagate_z(*circuit*, *line*, *back*)

find_intervals_with_same_target_qudits(*instructions*)

remove_z(*original_circuit*, *back=True*)

mqt.qudits.compiler.onedit.local_phases_transpilation.remove_phase_rotations

Module Contents

class ZRemovalPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

remove_rz_gates(*original_circuit*, *reverse=False*)

remove_initial_rz(*original_circuit*)

remove_trailing_rz_sequence(*original_circuit*)

Package Contents

class ZPropagationPass(*backend*, *back=True*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

propagate_z(*circuit*, *line*, *back*)

find_intervals_with_same_target_qudits(*instructions*)

remove_z(*original_circuit*, *back=True*)

class ZRemovalPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

remove_rz_gates(*original_circuit*, *reverse=False*)

remove_initial_rz(*original_circuit*)

remove_trailing_rz_sequence(*original_circuit*)

mqt.qudits.compiler.onedit.mapping_aware_transpilation

Submodules

mqt.qudits.compiler.onedit.mapping_aware_transpilation.phy_local_adaptive_decomp

Module Contents

class **PhyLocAdaPass**(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class **PhyAdaptiveDecomposition**(*gate*, *graph_orig*, *cost_limit=(0, 0)*, *dimension=-1*, *Z_prop=False*)

execute()

Z_extraction(*decomposition*, *placement*, *phase_propagation*)

DFS(*current_root*, *level=0*) → None

mqt.qudits.compiler.onedit.mapping_aware_transpilation.phy_local_qr_decomp

Module Contents

class **PhyLocQRPass**(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class **PhyQrDecomp**(*gate*, *graph_orig*, *Z_prop=False*, *not_stand_alone=True*)

execute()

Package Contents

class **PhyAdaptiveDecomposition**(*gate*, *graph_orig*, *cost_limit=(0, 0)*, *dimension=-1*, *Z_prop=False*)

execute()

Z_extraction(*decomposition*, *placement*, *phase_propagation*)

DFS(*current_root*, *level=0*) → None

class **PhyLocAdaPass**(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class **PhyLocQRPass**(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class **PhyQrDecomp**(*gate*, *graph_orig*, *Z_prop=False*, *not_stand_alone=True*)

execute()

`mqt.qudits.compiler.onedit.mapping_un_aware_transpilation`

Submodules

`mqt.qudits.compiler.onedit.mapping_un_aware_transpilation.log_local_adaptive_decomp`

Module Contents

class LogLocAdaPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class LogAdaptiveDecomposition(*gate*, *graph_orig*, *cost_limit=(0, 0)*, *dimension=-1*, *Z_prop=False*)

execute()

z_extraction(*decomposition*, *placement*, *phase_propagation*)

DFS(*current_root*, *level=0*) → None

`mqt.qudits.compiler.onedit.mapping_un_aware_transpilation.log_local_qr_decomp`

Module Contents

class LogLocQRPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class QrDecomp(*gate*, *graph_orig*, *Z_prop=False*, *not_stand_alone=True*)

execute()

Package Contents

class LogAdaptiveDecomposition(*gate*, *graph_orig*, *cost_limit=(0, 0)*, *dimension=-1*, *Z_prop=False*)

execute()

z_extraction(*decomposition*, *placement*, *phase_propagation*)

DFS(*current_root*, *level=0*) → None

class LogLocAdaPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class LogLocQRPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

Package Contents

class ZPropagationPass(*backend*, *back=True*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

propagate_z(*circuit*, *line*, *back*)

find_intervals_with_same_target_qudits(*instructions*)

remove_z(*original_circuit*, *back=True*)

class ZRemovalPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

remove_rz_gates(*original_circuit*, *reverse=False*)

remove_initial_rz(*original_circuit*)

remove_trailing_rz_sequence(*original_circuit*)

class PhyLocAdaPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class PhyLocQRPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class LogLocAdaPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class LogLocQRPass(*backend*)

Bases: *mqt.qudits.compiler.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

mqt.qudits.compiler.twodit

Subpackages

mqt.qudits.compiler.twodit.entanglement_qr

Submodules

mqt.qudits.compiler.twodit.entanglement_qr.crot

Module Contents

CEX_SEQUENCE

class CRotGen(*circuit*, *indices*)

crot_101_as_list(*theta*, *phi*)

permute_crot_101_as_list(*i*, *theta*, *phase*)

permute_doubled_crot_101_as_list(*i*, *theta*, *phase*)

z_from_crot_101_list(*i*, *phase*)

mqt.qudits.compiler.twodit.entanglement_qr.log_ent_qr_cex_decomp

Module Contents

class LogEntQRCEXPass(*backend*)

Bases: *mqt.qudits.compiler.compiler_pass.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class EntangledQRCEX(*gate*, *graph_orig_c*, *graph_orig_t*)

execute()

mqt.qudits.compiler.twodit.entanglement_qr.pswap

Module Contents

class PSwapGen(*circuit*, *indices*)

pswap_101_as_list(*teta*, *phi*)

permute_pswap_101_as_list(*pos*, *theta*, *phase*)

permute_quad_pswap_101_as_list(*pos*, *theta*, *phase*)

z_pswap_101_as_list(*i*, *phase*, *dimension_single*)

Package Contents

class CRotGen(*circuit*, *indices*)

crot_101_as_list(*theta*, *phi*)

permute_crot_101_as_list(*i*, *theta*, *phase*)

permute_doubled_crot_101_as_list(*i*, *theta*, *phase*)

z_from_crot_101_list(*i*, *phase*)

class EntangledQRCEX(*gate*, *graph_orig_c*, *graph_orig_t*)

execute()

class LogEntQRCEXPass(*backend*)

Bases: *mqt.qudits.compiler.compiler_pass.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

class PSwapGen(*circuit*, *indices*)

pswap_101_as_list(*teta*, *phi*)

permute_pswap_101_as_list(*pos*, *theta*, *phase*)

permute_quad_pswap_101_as_list(*pos*, *theta*, *phase*)

z_pswap_101_as_list(*i*, *phase*, *dimension_single*)

`mqt.qudits.compiler.twodit.variational_twodit_compilation`

Subpackages

`mqt.qudits.compiler.twodit.variational_twodit_compilation.ansatz`

Submodules

`mqt.qudits.compiler.twodit.variational_twodit_compilation.ansatz.ansatz_gen`

Module Contents

prepare_ansatz(*u*, *params*, *dims*)

cu_ansatz(*P*, *dims*)

ms_ansatz(*P*, *dims*)

ls_ansatz(*P*, *dims*)

`mqt.qudits.compiler.twodit.variational_twodit_compilation.ansatz.instantiate`

Module Contents

ansatz_decompose(*u*, *params*, *dims*)

create_cu_instance(*P*, *dims*)

create_ms_instance(*P*, *dims*)

create_ls_instance(*P*, *dims*)

`mqt.qudits.compiler.twodit.variational_twodit_compilation.ansatz.parametrize`

Module Contents

CUSTOM_PRIMITIVE

params_splitter(*params*, *dims*)

reindex(*ir*, *jc*, *num_col*)

bound_1

bound_2

bound_3

generic_sud(*params*, *dimension*) → ndarray

Package Contents

cu_ansatz(*P*, *dims*)

ls_ansatz(*P*, *dims*)

ms_ansatz(*P*, *dims*)

create_cu_instance(*P*, *dims*)

create_ls_instance(*P*, *dims*)

create_ms_instance(*P*, *dims*)

reindex(*ir*, *jc*, *num_col*)

mqt.qudits.compiler.twodit.variational_twodit_compilation.opt

Submodules

mqt.qudits.compiler.twodit.variational_twodit_compilation.opt.distance_measures FROM An alternative quantum fidelity for mixed states of qudits Xiaoguang Wang, 1, 2, * Chang-Shui Yu, 3 and x. x. Yi 3

Module Contents

size_check(*a: ndarray*, *b: ndarray*) → bool

fidelity_on_operator(*a: ndarray*, *b: ndarray*) → float

fidelity_on_unitares(*a: ndarray*, *b: ndarray*) → float

fidelity_on_density_operator(*a: ndarray*, *b: ndarray*) → float

density_operator(*state_vector*) → ndarray

frobenius_dist(*x*, *y*)

mqt.qudits.compiler.twodit.variational_twodit_compilation.opt.optimizer

Module Contents

class Optimizer

OBJ_FIDELITY = 0.0001

SINGLE_DIM_0

SINGLE_DIM_1

TARGET_GATE

MAX_NUM_LAYERS

X_SOLUTION = []

FUN_SOLUTION = []

timer_var = False

static bounds_assigner(*b1*, *b2*, *b3*, *num_params_single*, *d*)

classmethod obj_fun_core(*ansatz*, *lambdas*)

classmethod objective_fnc_ms(*lambdas*)

classmethod objective_fnc_ls(*lambdas*)

classmethod objective_fnc_cu(*lambdas*)

classmethod solve_anneal(*bounds*, *ansatz_type*, *result_queue*) → None

Package Contents

density_operator(*state_vector*) → ndarray

fidelity_on_density_operator(*a: ndarray*, *b: ndarray*) → float

fidelity_on_operator(*a: ndarray*, *b: ndarray*) → float

fidelity_on_unitares(*a: ndarray*, *b: ndarray*) → float

frobenius_dist(*x*, *y*)

size_check(*a: ndarray*, *b: ndarray*) → bool

class Optimizer

OBJ_FIDELITY = 0.0001

SINGLE_DIM_0

SINGLE_DIM_1

TARGET_GATE

MAX_NUM_LAYERS

X_SOLUTION = []

FUN_SOLUTION = []

timer_var = False

static bounds_assigner(*b1*, *b2*, *b3*, *num_params_single*, *d*)

classmethod obj_fun_core(*ansatz*, *lambdas*)

classmethod objective_fnc_ms(*lambdas*)

classmethod objective_fnc_ls(*lambdas*)

classmethod objective_fnc_cu(*lambdas*)

classmethod solve_anneal(*bounds*, *ansatz_type*, *result_queue*) → None

Submodules

mqt.qudits.compiler.twodit.variational_twodit_compilation.ansatz_solve_n_search

Module Contents

interrupt_function() → None

binary_search_compile(*max_num_layer*, *ansatz_type*)

run(*num_layer*, *ansatz_type*)

Package Contents

class LogEntQRCEXPass(*backend*)

Bases: *mqt.qudits.compiler.compiler_pass.CompilerPass*

Helper class that provides a standard way to create an ABC using inheritance.

transpile(*circuit*)

Submodules

mqt.qudits.compiler.compiler_pass

Module Contents

class CompilerPass(*backend*, ***kwargs*)

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

abstract transpile(*circuit*)

`mqt.qudits.compiler.dit_manager`

Module Contents

class QuditCompiler

passes_enabled

compile(*backend*, *circuit*, *passes_names*)

Package Contents

class CompilerPass(*backend*, ***kwargs*)

Bases: `abc.ABC`

Helper class that provides a standard way to create an ABC using inheritance.

abstract transpile(*circuit*)

`mqt.qudits.core`

Core structure used in the package.

Submodules

`mqt.qudits.core.dfs_tree`

Module Contents

class Node(*key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*, *parent_key*, *children=None*)

add(*new_key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*) → None

__str__() → str

Return str(self).

class NAryTree

property total_size

add(*new_key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*, *parent_key=None*) → None

find_node(*node*, *key*)

depth(*key*)

max_depth(*node*)

size_refresh(*node*)

found_checker(*node*)

min_cost_decomp(*node*)

retrieve_decomposition(*node*)

is_empty()

print_tree(*node*, *str_aux*)

`__str__`() → str

Return str(self).

`mqt.qudits.core.level_graph`

Module Contents

class LevelGraph(*edges*, *nodes*, *nodes_physical_mapping=None*, *initialization_nodes=None*, *qudit_index=None*, *og_circuit=None*)

Bases: `networkx.Graph`

Base class for undirected graphs.

A Graph stores nodes and edges with optional data, or attributes.

Graphs hold undirected edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes, except that *None* is not allowed as a node.

Edges are represented as links between nodes with optional key/value attributes.

Parameters

incoming_graph_data [graph (optional, default: None)] Data to initialize graph. If None (default) an empty graph is created. The data can be any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse matrix, or PyGraphviz graph.

attr [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

See Also DiGraph MultiGraph MultiDiGraph

Examples Create an empty graph structure (a “null graph”) with no nodes and no edges.

```
>>> G = nx.Graph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.Graph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714  # node must exist already to use G.nodes
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> G.add_edge(1, 2, weight=4.7)
>>> G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2]["weight"] = 4.7
>>> G.edges[1, 2]["weight"] = 4
```

Warning: we protect the graph data structure by making *G.edges* a read-only dict-like structure. However, you can assign to attributes in e.g. *G.edges[1, 2]*. Thus, use 2 sets of brackets to add/change data attributes: *G.edges[1, 2]['weight'] = 4* (For multigraphs: *MG.edges[u, v, key][name] = value*).

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are reported as an adjacency-dict *G.adj* or *G.adjacency()*

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, eattr in nbrsdict.items():
...         if "weight" in eattr:
...             # Do something useful with the edges
...             pass
```

But the edges() method is often more convenient:

```
>>> for u, v, weight in G.edges.data("weight"):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using object-attributes and methods. Reporting typically provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. *nodes[n]*, *edges[u, v]*, *adj[u][v]*) and iteration (e.g. *nodes.items()*, *nodes.data('color')*, *nodes.data('color', default='blue')* and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()*/*adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds edge data keyed by neighbor. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Each of these three dicts can be replaced in a subclass by a user defined dict-like object. In general, the dict-like features should be maintained but extra features can be added. To replace one of the dicts create a new graph class by changing the class(!) variable holding the factory for that dict-like structure.

node_dict_factory [function, (default: dict)] Factory function to be used to create the dict containing node attributes, keyed by node id. It should require no arguments and return a dict-like object

node_attr_dict_factory: [function, (default: dict)] Factory function to be used to create the node attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object

adjlist_outer_dict_factory [function, (default: dict)] Factory function to be used to create the outer-most dict in the data structure that holds adjacency info keyed by node. It should require no arguments and return a dict-like object.

adjlist_inner_dict_factory [function, (default: dict)] Factory function to be used to create the adjacency list dict which holds edge data keyed by neighbor. It should require no arguments and return a dict-like object

edge_attr_dict_factory [function, (default: dict)] Factory function to be used to create the edge attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object.

graph_attr_dict_factory [function, (default: dict)] Factory function to be used to create the graph attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object.

Typically, if your extension doesn't impact the data structure all methods will inherit without issue except: *to_directed/to_undirected*. By default these methods create a DiGraph/Graph class and you probably want them to create your extension of a DiGraph/Graph. To facilitate this we define two class variables that you can set in your subclass.

to_directed_class [callable, (default: DiGraph or MultiDiGraph)] Class to create a new graph structure in the *to_directed* method. If *None*, a NetworkX class (DiGraph or MultiDiGraph) is used.

to_undirected_class [callable, (default: Graph or MultiGraph)] Class to create a new graph structure in the *to_undirected* method. If *None*, a NetworkX class (Graph or MultiGraph) is used.

Subclassing Example

Create a low memory graph class that effectively disallows edge attributes by using a single attribute dict for all edges. This reduces the memory used, but you lose edge attributes.

```
>>> class ThinGraph(nx.Graph):
...     all_edge_dict = {"weight": 1}
...
...     def single_edge_dict(self):
...         return self.all_edge_dict
...
...     edge_attr_dict_factory = single_edge_dict
>>> G = ThinGraph()
>>> G.add_edge(2, 1)
>>> G[2][1]
{'weight': 1}
>>> G.add_edge(2, 2)
>>> G[2][1] is G[2][2]
True
```

property log_phy_map

phase_storing_setup() → None

distance_nodes(*source*, *target*)

distance_nodes_pi_pulses_fixed_ancilla(*source*, *target*)

logic_physical_map(*physical_nodes*) → None

define__states(*initialization_nodes*, *inreach_nodes*) → None

update_list(*lst_*, *num_a*, *num_b*)

deep_copy_func(*l_n*)

index(*lev_graph*, *node*)

swap_node_attributes(*node_a*, *node_b*)

swap_node_attr_simple(*node_a*, *node_b*) → None

swap_nodes(*node_a*, *node_b*)

get_VRz_gates()

get_node_sensitivity_cost(*node*)

get_edge_sensitivity(*node_a*, *node_b*)

is_irnode(*node*)

is_Inode(*node*)

__str__() → str

Returns a short summary of the graph.

Returns

info [string] Graph information including the graph name (if any), graph type, and the number of nodes and edges.

Examples

```
>>> G = nx.Graph(name="foo")
>>> str(G)
"Graph named 'foo' with 0 nodes and 0 edges"
```

```
>>> G = nx.path_graph(3)
>>> str(G)
'Graph with 3 nodes and 2 edges'
```

set_circuit(*circuit: mqt.qudits.circuit.QuantumCircuit*) → None

set_qudits_index(*index: int*) → None

Package Contents

class NAryTree

property total_size

add(*new_key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*, *parent_key=None*) → None

find_node(*node*, *key*)

depth(*key*)

max_depth(*node*)

size_refresh(*node*)

found_checker(*node*)

min_cost_decomp(*node*)

retrieve_decomposition(*node*)

is_empty()

print_tree(*node*, *str_aux*)

__str__() → str

Return str(self).

class Node(*key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*, *parent_key*, *children=None*)

add(*new_key*, *rotation*, *U_of_level*, *graph_current*, *current_cost*, *current_decomp_cost*, *max_cost*, *pi_pulses*) → None

__str__() → str

Return str(self).

class LevelGraph(*edges*, *nodes*, *nodes_physical_mapping=None*, *initialization_nodes=None*, *qudit_index=None*, *og_circuit=None*)

Bases: `networkx.Graph`

Base class for undirected graphs.

A Graph stores nodes and edges with optional data, or attributes.

Graphs hold undirected edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes, except that *None* is not allowed as a node.

Edges are represented as links between nodes with optional key/value attributes.

Parameters

incoming_graph_data [input graph (optional, default: None)] Data to initialize graph. If None (default) an empty graph is created. The data can be any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse matrix, or PyGraphviz graph.

attr [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

See Also DiGraph MultiGraph MultiDiGraph

Examples Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.Graph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.Graph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714  # node must exist already to use G.nodes
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> G.add_edge(1, 2, weight=4.7)
>>> G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2]["weight"] = 4.7
>>> G.edges[1, 2]["weight"] = 4
```

Warning: we protect the graph data structure by making *G.edges* a read-only dict-like structure. However, you can assign to attributes in e.g. *G.edges[1, 2]*. Thus, use 2 sets of brackets to add/change data attributes: *G.edges[1, 2]['weight'] = 4* (For multigraphs: *MG.edges[u, v, key][name] = value*).

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are reported as an adjacency-dict *G.adj* or *G.adjacency()*

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, eattr in nbrsdict.items():
...         if "weight" in eattr:
...             # Do something useful with the edges
...             pass
```

But the edges() method is often more convenient:

```
>>> for u, v, weight in G.edges.data("weight"):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using object-attributes and methods. Reporting typically provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. *nodes[n]*, *edges[u, v]*, *adj[u][v]*) and iteration (e.g. *nodes.items()*, *nodes.data('color')*, *nodes.data('color', default='blue')* and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()/adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds edge data keyed by neighbor. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Each of these three dicts can be replaced in a subclass by a user defined dict-like object. In general, the dict-like features should be maintained but extra features can be added. To replace one of the dicts create a new graph class by changing the class(!) variable holding the factory for that dict-like structure.

node_dict_factory [function, (default: dict)] Factory function to be used to create the dict containing node attributes, keyed by node id. It should require no arguments and return a dict-like object

node_attr_dict_factory: function, (default: dict) Factory function to be used to create the node attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object adjlist_outer_dict_factory [function, (default: dict)] Factory function to be used to create the outer-most dict in the data structure that holds adjacency info keyed by node. It should require no arguments and return a dict-like object.

adjlist_inner_dict_factory [function, (default: dict)] Factory function to be used to create the adjacency list dict which holds edge data keyed by neighbor. It should require no arguments and return a dict-like object

edge_attr_dict_factory [function, (default: dict)] Factory function to be used to create the edge attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object.

graph_attr_dict_factory [function, (default: dict)] Factory function to be used to create the graph attribute dict which holds attribute values keyed by attribute name. It should require no arguments and return a dict-like object.

Typically, if your extension doesn't impact the data structure all methods will inherit without issue except: *to_directed/to_undirected*. By default these methods create a DiGraph/Graph class and you probably want them to create your extension of a DiGraph/Graph. To facilitate this we define two class variables that you can set in your subclass.

to_directed_class [callable, (default: DiGraph or MultiDiGraph)] Class to create a new graph structure in the *to_directed* method. If *None*, a NetworkX class (DiGraph or MultiDiGraph) is used.

to_undirected_class [callable, (default: Graph or MultiGraph)] Class to create a new graph structure in the *to_undirected* method. If *None*, a NetworkX class (Graph or MultiGraph) is used.

Subclassing Example

Create a low memory graph class that effectively disallows edge attributes by using a single attribute dict for all edges. This reduces the memory used, but you lose edge attributes.

```
>>> class ThinGraph(nx.Graph):
...     all_edge_dict = {"weight": 1}
...
...     def single_edge_dict(self):
...         return self.all_edge_dict
...
...     edge_attr_dict_factory = single_edge_dict
>>> G = ThinGraph()
>>> G.add_edge(2, 1)
>>> G[2][1]
{'weight': 1}
>>> G.add_edge(2, 2)
>>> G[2][1] is G[2][2]
True
```

property log_phy_map

phase_storing_setup() → None

distance_nodes(*source*, *target*)

distance_nodes_pi_pulses_fixed_ancilla(*source*, *target*)

logic_physical_map(*physical_nodes*) → None

define__states(*initialization_nodes*, *inreach_nodes*) → None

update_list(*lst_*, *num_a*, *num_b*)

deep_copy_func(*l_n*)

index(*lev_graph*, *node*)

swap_node_attributes(*node_a*, *node_b*)

swap_node_attr_simple(*node_a*, *node_b*) → None

swap_nodes(*node_a*, *node_b*)

get_VRz_gates()

get_node_sensitivity_cost(*node*)

get_edge_sensitivity(*node_a*, *node_b*)

is_irnode(*node*)

is_Inode(*node*)

__str__() → str

Returns a short summary of the graph.

Returns

info [string] Graph information including the graph name (if any), graph type, and the number of nodes and edges.

Examples

```
>>> G = nx.Graph(name="foo")
>>> str(G)
"Graph named 'foo' with 0 nodes and 0 edges"
```

```
>>> G = nx.path_graph(3)
>>> str(G)
'Graph with 3 nodes and 2 edges'
```

set_circuit(*circuit: mqt.qudits.circuit.QuantumCircuit*) → None

set_qudits_index(*index: int*) → None

mqt.qudits.exceptions

Exceptions module.

Submodules

mqt.qudits.exceptions.backenderror

Module Contents

exception BackendNotFoundError(*message: str*)

Bases: Exception

Common base class for all non-exit exceptions.

mqt.qudits.exceptions.circuiterror

Module Contents

exception CircuitError(*message: str*)

Bases: Exception

Common base class for all non-exit exceptions.

mqt.qudits.exceptions.compilerexception

Module Contents

exception NodeNotFoundException(*value*)

Bases: Exception

Common base class for all non-exit exceptions.

__str__() → str
Return str(self).

exception SequenceFoundException(*node_key: int = -1*)
Bases: Exception

Common base class for all non-exit exceptions.

__str__() → str
Return str(self).

exception RoutingException
Bases: Exception

Common base class for all non-exit exceptions.

__str__() → str
Return str(self).

exception FidelityReachException(*message: str = ''*)
Bases: Exception

Common base class for all non-exit exceptions.

mqt.qudits.exceptions.joberror

Module Contents

exception JobError(*message: str*)
Bases: Exception

Common base class for all non-exit exceptions.

class JobTimeoutError(*message: str*)

Package Contents

exception BackendNotFoundError(*message: str*)
Bases: Exception

Common base class for all non-exit exceptions.

exception CircuitError(*message: str*)
Bases: Exception

Common base class for all non-exit exceptions.

exception FidelityReachException(*message: str = ''*)
Bases: Exception

Common base class for all non-exit exceptions.

exception NodeNotFoundException(*value*)
Bases: Exception

Common base class for all non-exit exceptions.

__str__() → str
Return str(self).

exception RoutingException
Bases: Exception

Common base class for all non-exit exceptions.

__str__() → str
Return str(self).

exception SequenceFoundException(*node_key: int = -1*)

Bases: `Exception`

Common base class for all non-exit exceptions.

__str__() → str

Return str(self).

exception JobError(*message: str*)

Bases: `Exception`

Common base class for all non-exit exceptions.

class JobTimeoutError(*message: str*)

mqt.qudits.quantum_circuit

Qudit Quantum Circuit Module.

Subpackages

`mqt.qudits.quantum_circuit.components`

Subpackages

`mqt.qudits.quantum_circuit.components.extensions`

Submodules

`mqt.qudits.quantum_circuit.components.extensions.controls`

Module Contents

class ControlData

indices: list[int] | int

ctrl_states: list[int] | int

`mqt.qudits.quantum_circuit.components.extensions.gate_types`

Module Contents

class GateTypes(**args, **kwds*)

Bases: `enum.Enum`

Enumeration for job status.

SINGLE = 'Single Qudit Gate'

TWO = 'Two Qudit Gate'

MULTI = 'Multi Qudit Gate'

CORE_GATE_TYPES = ()

`mqt.qudits.quantum_circuit.components.extensions.matrix_factory`

Module Contents

class MatrixFactory(*gate, identities_flag*)

generate_matrix()

classmethod apply_identites_and_controls(*matrix, qudits_applied, dimensions, ref_lines, controls=None, controls_levels=None*)

classmethod wrap_in_identities(*matrix, indices, sizes*)

from_dirac_to_basis(*vec*, *d*)

calculate_q0_q1(*lev*, *dim*)

insert_at(*big_arr*, *pos*, *to_insert_arr*)

Quite a forceful way of embedding a parameters into big_arr.

Submodules

`mqt.qudits.quantum_circuit.components.classic_register`

Module Contents

class ClassicRegister(*name*, *size*)

classmethod from_map(*sitemap: dict*) → list[*ClassicRegister*]

__qasm__()

__getitem__(*key*)

`mqt.qudits.quantum_circuit.components.quantum_register`

Module Contents

class QuantumRegister(*name*, *size*, *dims=None*)

classmethod from_map(*sitemap: dict*) → list[*QuantumRegister*]

__qasm__()

__getitem__(*key*)

Package Contents

class ClassicRegister(*name*, *size*)

classmethod from_map(*sitemap: dict*) → list[*ClassicRegister*]

__qasm__()

__getitem__(*key*)

class QuantumRegister(*name*, *size*, *dims=None*)

classmethod from_map(*sitemap: dict*) → list[*QuantumRegister*]

__qasm__()

__getitem__(*key*)

`mqt.qudits.quantum_circuit.gates` Instructions module.

Submodules

`mqt.qudits.quantum_circuit.gates.csum`

Module Contents

class CSum(*circuit:* QuantumCircuit, *name: str*, *target_qudits: list[int]* | *int*, *dimensions: list[int]* | *int*, *controls:* ControlData | *None* = *None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.custom_multi`

Module Contents

class CustomMulti(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, parameters: ndarray, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Multi body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.custom_one`

Module Contents

class CustomOne(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, parameters: ndarray, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

One body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.custom_two`

Module Contents

class CustomTwo(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, parameters: ndarray, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Two body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.cx`

Module Contents

class CEx(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, parameters: list | None, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.gellmann

Module Contents

class GellMann(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Gate used as generator for Givens rotations.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.h

Module Contents

class H(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.ls

Module Contents

class LS(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.ms

Module Contents

class MS(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.perm

Module Contents

class Perm(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

Verify that the input is a list of indices

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.r

Module Contents

class R(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

property cost

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.randu

Module Contents

class RandU(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter() → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.rh

Module Contents

class Rh(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

SU2 Hadamard

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.rz`

Module Contents

class Rz(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

property cost

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.s`

Module Contents

class S(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

`mqt.qudits.quantum_circuit.gates.virt_rz`

Module Contents

class VirtRz(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

property cost

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.x

Module Contents

class X(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

mqt.qudits.quantum_circuit.gates.z

Module Contents

class Z(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

Package Contents

class ControlData

indices: list[int] | int

ctrl_states: list[int] | int

class GateTypes(**args, **kwds*)

Bases: enum.Enum

Enumeration for job status.

SINGLE = 'Single Qudit Gate'

TWO = 'Two Qudit Gate'

MULTI = 'Multi Qudit Gate'

class CSum(*circuit: QuantumCircuit, name: str, target_qudits: list[int] | int, dimensions: list[int] | int, controls: ControlData | None = None*)

Bases: *mqt.qudits.quantum_circuit.gate.Gate*

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class CustomMulti(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: ndarray*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Multi body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class CustomOne(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: ndarray*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

One body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class CustomTwo(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: ndarray*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Two body custom gate

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class CEx(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class GellMann(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Gate used as generator for Givens rotations.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class H(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class LS(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class MS(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class Perm(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter*) → bool

Verify that the input is a list of indices

__str__() → str

Return str(self).

class R(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

property cost

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class RandU(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter() → bool

__str__() → str

Return str(self).

class Rh(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

SU2 Hadamard

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class Rz(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

property cost

__array__() → ndarray

levels_setter(*la*, *lb*)

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class S(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class VirtRz(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *parameters: list | None*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

property cost

__array__() → ndarray

validate_parameter(*parameter*) → bool

__str__() → str

Return str(self).

class X(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

class Z(*circuit: QuantumCircuit*, *name: str*, *target_qudits: list[int] | int*, *dimensions: list[int] | int*, *controls: ControlData | None = None*)

Bases: `mqt.qudits.quantum_circuit.gate.Gate`

Unitary gate_matrix.

__array__() → ndarray

validate_parameter(*parameter=None*) → bool

__str__() → str

Return str(self).

Submodules

`mqt.qudits.quantum_circuit.circuit`

Module Contents

is_not_none_or_empty(*variable*)

add_gate_decorator(*func*)

class QuantumCircuit(**args*)

property num_qudits

property dimensions

property gate_set: None

qasm_to_gate_set_dict

classmethod get_qasm_set()

reset() → None

copy()

append(*qreg: QuantumRegister*) → None

append_classic(*creg: ClassicRegister*) → None

csum(*qudits: list[int]*)

cu_one(*qudits: int, parameters: ndarray, controls: ControlData | None = None*)

cu_two(*qudits: list[int], parameters: ndarray, controls: ControlData | None = None*)

cu_multi(*qudits: list[int], parameters: ndarray, controls: ControlData | None = None*)

cx(*qudits: list[int], parameters: list | None = None*)

gellmann(*qudit: int, parameters: list | None = None, controls: ControlData | None = None*)

h(*qudit: int, controls: ControlData | None = None*)

rh(*qudit: int, parameters: list, controls: ControlData | None = None*)

ls(*qudits: list[int], parameters: list | None = None*)

ms(*qudits: list[int], parameters: list | None = None*)

pm(*qudits: list[int], parameters: list*)

r(*qudit: int, parameters: list, controls: ControlData | None = None*)

randu(*qudits: list[int]*)

rz(*qudit: int, parameters: list, controls: ControlData | None = None*)

virtrz(*qudit: int, parameters: list, controls: ControlData | None = None*)

s(*qudit: int, controls: ControlData | None = None*)

x(*qudit: int, controls: ControlData | None = None*)

z(*qudit: int, controls: ControlData | None = None*)

replace_gate(*gate_index: int, sequence: list[Gate]*) → None

set_instructions(*sequence: list[Gate]*)

from_qasm(*qasm_prog*) → None

Create a circuit from qasm text

to_qasm()

save_to_file(*file_name: str, file_path: str = '.'*) → str

Save qasm into a file with the specified name and path.

Parameters

- **text** (`str`) – The text to be saved into the file.
- **file_name** (`str`) – The name of the file.
- **file_path** (`str, optional`) – The path where the file will be saved. Defaults to "." (current directory).

Returns The full path of the saved file.

Return type str

load_from_file(*file_path: str*) → None

Load text from a file.

Parameters **file_path** (*str*) – The path of the file to load.
Returns The text loaded from the file.
Return type str

draw() → None

mqt.qudits.quantum_circuit.gate

Module Contents

class Instruction(*name: str*)

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

class Gate(*circuit: QuantumCircuit, name: str, gate_type: enum, target_qudits: list[int] | int, dimensions: list[int] | int, params: list | ndarray | None = None, control_set=None, label: str | None = None, duration=None, unit='dt'*)

Bases: *Instruction*

Unitary gate_matrix.

property reference_lines

property get_control_lines

property control_info

abstract __array__() → ndarray

dag()

to_matrix(*identities=0*) → ndarray

Return a np.ndarray for the gate_matrix unitary parameters.

Returns if the Gate subclass has a parameters definition.
Return type np.ndarray
Raises *CircuitError* – If a Gate subclass does not implement this method an exception will be raised when this base class method is called.

control(*indices: list[int] | int, ctrl_states: list[int] | int*)

abstract validate_parameter(*parameter*)

__qasm__() → str

Generate QASM for Gate export

abstract __str__() → str

Return str(self).

check_long_range()

set_gate_type_single() → None

set_gate_type_two() → None

set_gate_type_multi() → None

return_custom_data() → str

mqt.qudits.quantum_circuit.qasm

Module Contents

class QASM

Class that manages the parsing of QASM programs

parse_nonspecial_lines(*line*, *rgxs*, *in_comment_flag*)

parse_qreg(*line*, *rgxs*, *sitemap*) → bool

parse_creg(*line*, *rgxs*, *sitemap_classic*) → bool

safe_eval_math_expression(*expression*)

parse_gate(*line*, *rgxs*, *sitemap*, *gates*) → bool

parse_ignore(*line*, *rgxs*, *warned*) → bool

parse_ditqasm2_str(*contents*)

Parse the string contents of an OpenQASM 2.0 file. This parser only supports basic gate_matrix definitions, and is not guaranteed to check the full openqasm grammar.

parse_ditqasm2_file(*fname*)

Parse an OpenQASM 2.0 file.

Package Contents

class QuantumCircuit(**args*)

property num_qudits

property dimensions

property gate_set: None

qasm_to_gate_set_dict

classmethod get_qasm_set()

reset() → None

copy()

append(*qreg: QuantumRegister*) → None

append_classic(*creg: ClassicRegister*) → None

csum(*qudits: list[int]*)

cu_one(*qudits: int*, *parameters: ndarray*, *controls: ControlData | None = None*)

cu_two(*qudits: list[int]*, *parameters: ndarray*, *controls: ControlData | None = None*)

cu_multi(*qudits: list[int]*, *parameters: ndarray*, *controls: ControlData | None = None*)

cx(*qudits: list[int]*, *parameters: list | None = None*)

gellmann(*qudit: int*, *parameters: list | None = None*, *controls: ControlData | None = None*)

h(*qudit: int*, *controls: ControlData | None = None*)

rh(*qudit: int*, *parameters: list*, *controls: ControlData | None = None*)

ls(*qudits: list[int]*, *parameters: list | None = None*)

ms(*qudits: list[int]*, *parameters: list | None = None*)

pm(*qudits: list[int]*, *parameters: list*)

r(*qudit: int*, *parameters: list*, *controls: ControlData | None = None*)

randu(*qudits: list[int]*)

rz(*qudit: int*, *parameters: list*, *controls: ControlData | None = None*)

virtrz(*qudit: int*, *parameters: list*, *controls: ControlData | None = None*)

s(*qudit: int*, *controls: ControlData | None = None*)

x(*qudit: int*, *controls: ControlData | None = None*)

z(*qudit: int*, *controls: ControlData | None = None*)

replace_gate(*gate_index: int*, *sequence: list[Gate]*) → None

set_instructions(*sequence: list[Gate]*)

from_qasm(*qasm_prog*) → None

Create a circuit from qasm text

to_qasm()

save_to_file(*file_name: str*, *file_path: str = '.'*) → str

Save qasm into a file with the specified name and path.

Parameters

- **text** (*str*) – The text to be saved into the file.
- **file_name** (*str*) – The name of the file.
- **file_path** (*str, optional*) – The path where the file will be saved. Defaults to "." (current directory).

Returns The full path of the saved file.
Return type str

load_from_file(*file_path: str*) → None

Load text from a file.

Parameters **file_path** (*str*) – The path of the file to load.
Returns The text loaded from the file.
Return type str

draw() → None

class QuantumRegister(*name*, *size*, *dims=None*)

classmethod from_map(*sitemap: dict*) → list[*QuantumRegister*]

__qasm__()

__getitem__(*key*)

class QASM

Class that manages the parsing of QASM programs

parse_nonspecial_lines(*line*, *rgxs*, *in_comment_flag*)

parse_qreg(*line*, *rgxs*, *sitemap*) → bool

parse_creg(*line*, *rgxs*, *sitemap_classic*) → bool

safe_eval_math_expression(*expression*)

parse_gate(*line*, *rgxs*, *sitemap*, *gates*) → bool

parse_ignore(*line*, *rgxs*, *warned*) → bool

parse_ditqasm2_str(*contents*)

Parse the string contents of an OpenQASM 2.0 file. This parser only supports basic gate_matrix definitions, and is not guaranteed to check the full openqasm grammar.

parse_ditqasm2_file(*fname*)

Parse an OpenQASM 2.0 file.

mqt.qudits.simulation

Subpackages

mqt.qudits.simulation.backends

Subpackages

mqt.qudits.simulation.backends.fake_backends

Submodules

mqt.qudits.simulation.backends.fake_backends.fake_traps2six

Module Contents

class FakeIonTraps2Six(*provider: mqt.qudits.simulation.qudit_provider.QuditProvider* | *None* = *None*, *name: str* | *None* = *None*, *description: str* | *None* = *None*, *online_date: datetime* | *None* = *None*, *backend_version: str* | *None* = *None*, ***fields*)

Bases: *mqt.qudits.simulation.backends.tnsim.TNSim*

Helper class that provides a standard way to create an ABC using inheritance.

property version: int

property energy_level_graphs: list[*LevelGraph*, *LevelGraph*]

mqt.qudits.simulation.backends.fake_backends.fake_traps2three

Module Contents

class FakeIonTraps2Trits(*provider: mqt.qudits.simulation.qudit_provider.QuditProvider* | *None* = *None*, *name: str* | *None* = *None*, *description: str* | *None* = *None*, *online_date: datetime* | *None* = *None*, *backend_version: str* | *None* = *None*, ***fields*)

Bases: *mqt.qudits.simulation.backends.tnsim.TNSim*

Helper class that provides a standard way to create an ABC using inheritance.

property version: int

property energy_level_graphs: list[*LevelGraph*, *LevelGraph*]

Package Contents

class FakeIonTraps2Six(*provider: mqt.qudits.simulation.qudit_provider.QuditProvider* | *None* = *None*, *name: str* | *None* = *None*, *description: str* | *None* = *None*, *online_date: datetime* | *None* = *None*, *backend_version: str* | *None* = *None*, ***fields*)

Bases: *mqt.qudits.simulation.backends.tnsim.TNSim*

Helper class that provides a standard way to create an ABC using inheritance.

property version: int

property energy_level_graphs: list[*LevelGraph*, *LevelGraph*]

class FakeIonTraps2Trits(*provider: mqt.qudits.simulation.qudit_provider.QuditProvider | None = None, name: str | None = None, description: str | None = None, online_date: datetime | None = None, backend_version: str | None = None, **fields*)

Bases: `mqt.qudits.simulation.backends.tnsim.TNSim`

Helper class that provides a standard way to create an ABC using inheritance.

property version: int

property energy_level_graphs: list[LevelGraph, LevelGraph]

Submodules

`mqt.qudits.simulation.backends.backendv2`

Module Contents

class Backend(*provider: mqt.qudits.simulation.qudit_provider.QuditProvider | None = None, name: str | None = None, description: str | None = None, online_date: datetime | None = None, backend_version: str | None = None, **fields: Any*)

Bases: `abc.ABC`

Helper class that provides a standard way to create an ABC using inheritance.

property version: int

property instructions: list[tuple[Gate, tuple[int]]]

property operations: list[Gate]

property operation_names: list[str]

property num_qudits: int

abstract property energy_level_graphs: list[LevelGraph, LevelGraph]

property options

property provider

target

set_options(***fields*) → None

abstract run(*run_input, **options*) → *Job*

`mqt.qudits.simulation.backends.misim`

Module Contents

class MISim(***fields*)

Bases: `mqt.qudits.simulation.backends.backendv2.Backend`

Helper class that provides a standard way to create an ABC using inheritance.

run(*circuit: QuantumCircuit, **options*) → *Job*

execute(*circuit: QuantumCircuit, noise_model: NoiseModel | None = None*) → ndarray

`mqt.qudits.simulation.backends.stochastic_sim`

Module Contents

stochastic_simulation(*backend: Backend, circuit: QuantumCircuit*)

stochastic_execution(*args*)

stochastic_simulation_misim(*backend:* Backend, *circuit:* QuantumCircuit)

stochastic_execution_misim(*args*)

mqt.qudits.simulation.backends.tnsim

Module Contents

class TNSim(***fields*)

Bases: *mqt.qudits.simulation.backends.backendv2.Backend*

Helper class that provides a standard way to create an ABC using inheritance.

run(*circuit:* QuantumCircuit, ***options*)

execute(*circuit:* QuantumCircuit)

Package Contents

class MISim(***fields*)

Bases: *mqt.qudits.simulation.backends.backendv2.Backend*

Helper class that provides a standard way to create an ABC using inheritance.

run(*circuit:* QuantumCircuit, ***options*) → *Job*

execute(*circuit:* QuantumCircuit, *noise_model:* NoiseModel | *None* = *None*) → ndarray

class TNSim(***fields*)

Bases: *mqt.qudits.simulation.backends.backendv2.Backend*

Helper class that provides a standard way to create an ABC using inheritance.

run(*circuit:* QuantumCircuit, ***options*)

execute(*circuit:* QuantumCircuit)

mqt.qudits.simulation.jobs

Submodules

mqt.qudits.simulation.jobs.job

Module Contents

class Job(*backend:* Backend | *None*, *job_id:* *str* = *'auto'*, ***kwargs*)

Class to handle jobs

This first version of the Backend abstract class is written to be mostly backwards compatible with the legacy providers interface. This was done to ease the transition for users and provider maintainers to the new versioned providers. Expect future versions of this abstract class to change the data model and interface.

version = 1

job_id() → str

Return a unique id identifying the job.

backend() → *Backend*

Return the backend where this job was executed.

done() → bool

Return whether the job has successfully run.

running() → bool

Return whether the job is actively running.

cancelled() → bool

Return whether the job has been cancelled.

in_final_state() → bool

Return whether the job is in a final job state such as DONE or ERROR.

wait_for_final_state(*timeout: float | None = None*, *wait: float = 5*, *callback: Callable | None = None*) → None

Poll the job status until it progresses to a final state such as DONE or ERROR.

Parameters
- **timeout** – Seconds to wait for the job. If None, wait indefinitely.
- **wait** – Seconds between queries.
- **callback** – Callback function invoked after each query.

Raises **JobTimeoutError** – If the job does not reach a final state before the specified timeout.

abstract submit() → NoReturn

Submit the job to the backend for execution.

result()

Return the results of the job.

set_result(*result*) → None

abstract cancel() → NoReturn

Attempt to cancel the job.

abstract status() → str

Return the status of the job, among the values of BackendStatus.

`mqt.qudits.simulation.jobs.job_result`

Module Contents

class JobResult(*state_vector: list[numpy.complex128]*, *counts: list[int]*)

get_counts() → list[int]

get_state_vector() → list[numpy.complex128]

`mqt.qudits.simulation.jobs.jobstatus`

Module Contents

class JobStatus(**args*, ***kwds*)

Bases: `enum.Enum`

Enumeration for job status.

INITIALIZING = 'Initializing: Job is being initialized'

QUEUED = 'Queued: Job is waiting in the queue'

VALIDATING = 'Validating: Job is being validated'

RUNNING = 'Running: Job is actively running'

CANCELLED = 'Cancelled: Job has been cancelled'

DONE = 'Done: Job has successfully run'

ERROR = 'Error: Job incurred an error'

JOB_FINAL_STATES = ()

Package Contents

class Job(*backend: Backend | None*, *job_id: str = 'auto'*, ***kwargs*)

Class to handle jobs

This first version of the Backend abstract class is written to be mostly backwards compatible with the legacy providers interface. This was done to ease the transition for users and provider maintainers to the new versioned providers. Expect future versions of this abstract class to change the data model and interface.

version = 1

job_id() → str

Return a unique id identifying the job.

backend() → *Backend*

Return the backend where this job was executed.

done() → bool

Return whether the job has successfully run.

running() → bool

Return whether the job is actively running.

cancelled() → bool

Return whether the job has been cancelled.

in_final_state() → bool

Return whether the job is in a final job state such as DONE or ERROR.

wait_for_final_state(*timeout: float | None = None*, *wait: float = 5*, *callback: Callable | None = None*) → None

Poll the job status until it progresses to a final state such as DONE or ERROR.

Parameters

- **timeout** – Seconds to wait for the job. If None, wait indefinitely.
- **wait** – Seconds between queries.
- **callback** – Callback function invoked after each query.

Raises ***JobTimeoutError*** – If the job does not reach a final state before the specified timeout.

abstract submit() → NoReturn

Submit the job to the backend for execution.

result()

Return the results of the job.

set_result(*result*) → None

abstract cancel() → NoReturn

Attempt to cancel the job.

abstract status() → str

Return the status of the job, among the values of BackendStatus.

class JobResult(*state_vector: list[numpy.complex128]*, *counts: list[int]*)

get_counts() → list[int]

get_state_vector() → list[numpy.complex128]

class JobStatus(**args*, ***kwds*)

Bases: `enum.Enum`

Enumeration for job status.

INITIALIZING = 'Initializing: Job is being initialized'

QUEUED = 'Queued: Job is waiting in the queue'

VALIDATING = 'Validating: Job is being validated'

RUNNING = 'Running: Job is actively running'

CANCELLED = 'Cancelled: Job has been cancelled'

DONE = 'Done: Job has successfully run'

ERROR = 'Error: Job incurred an error'

mqt.qudits.simulation.noise_tools

Submodules

mqt.qudits.simulation.noise_tools.noise

Module Contents

class Noise

probability_depolarizing: float

probability_dephasing: float

class NoiseModel

property basis_gates

add_recurrent_quantum_error_locally(*noise*, *gates*, *qudits*) → None

add_quantum_error_locally(*noise*, *gates*) → None

add_all_qudit_quantum_error(*noise*, *gates*) → None

add_nonlocal_quantum_error(*noise*, *gates*) → None

add_nonlocal_quantum_error_on_target(*noise*, *gates*) → None

add_nonlocal_quantum_error_on_control(*noise*, *gates*) → None

__str__() → str

Return str(self).

mqt.qudits.simulation.noise_tools.noisy_circuit_factory

Module Contents

class NoisyCircuitFactory(*noise_model: NoiseModel*, *circuit: QuantumCircuit*)

generate_circuit()

Package Contents

class Noise

probability_depolarizing: float

probability_dephasing: float

class NoiseModel

property basis_gates

add_recurrent_quantum_error_locally(*noise*, *gates*, *qudits*) → None

add_quantum_error_locally(*noise*, *gates*) → None

add_all_qudit_quantum_error(*noise*, *gates*) → None

add_nonlocal_quantum_error(*noise*, *gates*) → None

add_nonlocal_quantum_error_on_target(*noise*, *gates*) → None

add_nonlocal_quantum_error_on_control(*noise*, *gates*) → None

__str__() → str

Return str(self).

class **NoisyCircuitFactory**(*noise_model: NoiseModel*, *circuit: QuantumCircuit*)

generate_circuit()

Submodules

mqt.qudits.simulation.qudit_provider

Module Contents

class **MQTQuditProvider**

property version: int

get_backend(*name: str | None = None*, ***kwargs*) → *Backend*

Return a single backend matching the specified filtering.

backends(*name: str | None = None*, ***kwargs*) → list[*Backend*]

Return a list of backends matching the specified filtering.

__eq__(*other: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

mqt.qudits.simulation.save_info

Module Contents

save_full_states(*list_of_vectors*, *file_path=None*, *file_name=None*) → None

save_shots(*shots*, *file_path=None*, *file_name=None*) → None

Package Contents

class **MQTQuditProvider**

property version: int

get_backend(*name: str | None = None*, ***kwargs*) → *Backend*

Return a single backend matching the specified filtering.

backends(*name: str | None = None*, ***kwargs*) → list[*Backend*]

Return a list of backends matching the specified filtering.

__eq__(*other: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

mqt.qudits.visualisation

Submodules

mqt.qudits.visualisation.drawing_routines

Module Contents

draw_qudit_local(*circuit: QuantumCircuit*) → None

mqt.qudits.visualisation.mini_quantum_information

Module Contents

get_density_matrix_from_counts(*results*, *circuit*)

partial_trace(*rho*, *qudits2keep*, *dims*, *optimize=False*)

Calculate the partial trace

p_a = Tr_b(p)

Parameters

p [2D array] Matrix to trace

qudits2keep [array] An array of indices of the spaces to keep after being traced. For instance, if the space is A x B x C x D and we want to trace out B and D, keep = [0,2]

dims [array] An array of the dimensions of each space. For instance, if the space is A x B x C x D, dims = [dim_A, dim_B, dim_C, dim_D]

Returns

p_a [2D array] Traced matrix

mqt.qudits.visualisation.plot_information

Module Contents

class HistogramWithErrors(*labels*, *counts*, *errors*, *title='Simulation'*)

generate_histogram() → None

save_to_png(*filename*) → None

state_labels(*circuit*)

plot_state(*result: ndarray*, *circuit: QuantumCircuit*, *errors=None*) → None

plot_counts(*result*, *circuit: QuantumCircuit*) → None

Package Contents

draw_qudit_local(*circuit: QuantumCircuit*) → None

get_density_matrix_from_counts(*results*, *circuit*)

partial_trace(*rho*, *qudits2keep*, *dims*, *optimize=False*)

Calculate the partial trace

p_a = Tr_b(p)

Parameters

p [2D array] Matrix to trace

qudits2keep [array] An array of indices of the spaces to keep after being traced. For instance, if the space is A x B x C x D and we want to trace out B and D, keep = [0,2]

dims [array] An array of the dimensions of each space. For instance, if the space is A x B x C x D, dims = [dim_A, dim_B, dim_C, dim_D]

Returns

p_a [2D array] Traced matrix

plot_counts(*result*, *circuit: QuantumCircuit*) → None

plot_state(*result: ndarray*, *circuit: QuantumCircuit*, *errors=None*) → None

III-B Package Contents

__version__: str

version_info: tuple[int, int, int, str, str] | tuple[int, int, int]

References

[1] Kevin Mato, Stefan Hillmich, and Robert Wille. Mixed-dimensional qudit state preparation using edge-weighted decision diagrams. In *Design Automation Conference (DAC)*. 2024.

[2] Kevin Mato, Stefan Hillmich, and Robert Wille. Mixed-dimensional quantum circuit simulation with decision diagrams. In *International Conference on Quantum Computing and Engineering (QCE)*. 2023. doi:10.1109/QCE57702.2023.00112.

[3] Kevin Mato, Stefan Hillmich, and Robert Wille. Compression of qubit circuits: Mapping to mixed-dimensional quantum systems. In *International Conference on Quantum Software (QSW)*. 2023. doi:10.1109/QSW59989.2023.00027.

[4] Kevin Mato, Martin Ringbauer, Stefan Hillmich, and Robert Wille. Compilation of entangling gates for high-dimensional quantum systems. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2023. doi:10.1145/3566097.3567930.

[5] Kevin Mato, Martin Ringbauer, Stefan Hillmich, and Robert Wille. Adaptive compilation of multi-level quantum operations. In *International Conference on Quantum Computing and Engineering (QCE)*. 2022. doi:10.1109/QCE53715.2022.00070.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

W

X

Z