Quantum SDK TKET : : CHEAT SHEET

QUANTINUUM

Pipeline of quantum computing

⓪Install pytket and pytket-extensions → ①Create quantum circuit → ②Compile quantum circuit (simplify) → ③Compile quantum circuit (fit device architecture) → ④Execute quantum circuit on quantum device or simulator

Install pytket and pytket-extensions

An implementation of TKET is currently available in the form of the pytket package for python 3.9+ on Linux, MacOS and Windows.

```
pip install pytket
```

pytket-extension modules are available for interfacing pytket with several quantum software, including Qiskit, Cirq, and for adding quantum devices and simulators to target.

Each extension module can be installed similarly as follows.

```
pip install pytket-X
```

Ex. when you install TKET extension for qiskit

```
pip install pytket-qiskit
```

A full list of available pytket extensions is shown in the webpage
https://cqcl.github.io/pytket-extensions/api/index.html

Create quantum circuit

A quantum circuit consists of three main components which are quantum and classical datum (qubits and bits), quantum gates (operators on qubits), and measurement (observation of state information of desired qubits on bits).

Preparation of quantum and classical datum

Prepare quantum data (qubits $|0\rangle^{\otimes n}$) and classical datum (0-bit string 0 … 0).
Ex. Create a circuit with name '*circ*' having 3 qubits $|0\rangle^{\otimes 3}$ with q-register name '*q*' and 2 bits with c-register name '*c*'.

```
from pytket import Circuit
circ = Circuit(3, 2 , 'circ')
```

Ex. Add 2 qubits with q-register name '*p*' and 2 bits with c-register name '*d*'.

```
circ.add_q_register(name='p', size=2)
circ.add_c_register(name='d', size=2)
```

Basic Quantum Gates

TKET supports that Circuit method appending some basic gates to the end of the circuit.
See in detail.
https://cqcl.github.io/tket/pytket/api/circuit_class.html
Also see definition of quantum gates as a matrix.
https://cqcl.github.io/tket/pytket/api/optype.html

```
circ.quantum_gate(angles, control_qubits, target_qubits)
```

Note: *angles* in rotation gates are expected to have parameters in multiples of pi (half-turns).

-Supported *quantum_gates*:
{X, Y, Z, H, S, Sdg, T, Tdg, SX, SXdg, V, Vdg, SWAP, ISWAPMax, ECR}

Ex. Apply X gate to the qubit q[0] and SWAP gate to qubits (q[1],q[2]).

```
circ.X(0)
circ.SWAP(1,2)
```

-Supported controlled *quantum_gates*:
{CX, CY, CZ, CH, CSX, CSXdg, CV, CVdg, CSWAP, CCX}

Ex. Apply CX gate to the circuit with control qubit q[0] and target qubit q[1], CCX gate with control qubits (q[0],q[2]) and target qubit q[1], and CSWAP gate with control qubit q[2] and target qubits (q[0],q[1]).

```
circ.CX(0,1)
circ.CCX(0,2,1)
circ.CSWAP(2,0,1)
```

-Supported rotation *quantum_gates*:
{Rx, Ry, Rz, U1, U2, U3, TK1, TK2, PhasedX, XXPhase, YYPhase, ZZPhase, XXPhase3, ZZMax, ISWAP, ESWAP, FSim}

Ex. Apply Rx of angle 0.5pi radians on qubit q[0].

```
circ.Rx(0.5,0)
```

-Supported controlled rotation *quantum_gates*:
{CRx, CRy, CRz, CU1, CU3}

Ex. Apply Controlled-Rz of angle 0.3pi radians with control qubit q[1] and target qubit q[0].

```
circ.CRz(0.3, 1, 0)
```

More Quantum Gates

For less commonly used gates, a wider variety is available using the OpType enum, which can be added using Circuit.add_gate method.

```
circ.add_gate(OpType, angles list, control and target qubits list )
```

-Supported Basic Quantum Gates
+ {NPhasedX, CnRy, CnX, CnY, CnZ, etc}

Ex. Using Circuit.add_gate method, apply X gate to the qubit q[0], apply Controlled-Rz of angle 0.3pi radians with control qubit q[1] and target qubit q[0], and apply n-controlled-Ry of angle 0.5pi radians with control qubits (q[0], q[2]) and target qubit q[1].

```
from pytket import OpType
circ.add_gate(OpType.X, [0])
circ.add_gate(OpType.CRx, [0.3], [1, 0])
circ.add_gate(OpType.CnRy, [0.5], [0, 2, 1])
```

Measurement

Curcuit.Measure appends a single qubit Z-basis measurement.
Curcuit.measure_register appends a measure gate to all qubits in the given register, storing the results in the given classical bits.
Curcuit.measure_all appends a measure gate to all qubits, storing the results in classical bits.

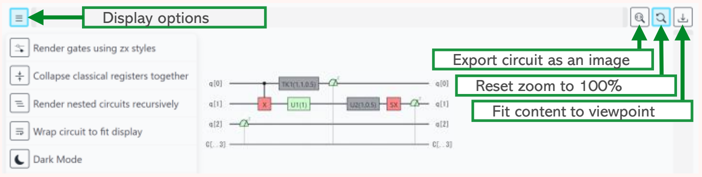
Ex. Append measure gate to q[1], storing result in c[0], append measure gates to all qubits in the qubit register with name '*q*', storing the bit register with name '*c*', and append measure gates to all qubits in the circuit.

```
circ.Measure(1, 0) #measure gate to q[1], storing c[0]
circ.measure_register(circ.get_q_register('q'), 'c') #measure q reg
circ.measure_all() #measure all qubits in the circuit
```

Quantum Circuit Visualisation

If you are working in a Jupyter environment, you can render a circuit for inline display.

```
from pytket.circuit.display import render_circuit_jupyter
render_circuit_jupyter(circ)
```


Display options
Render gates using zx styles
Collapse classical registers together
Render nested circuits recursively
Wrap circuit to fit display
Dark Mode
Export circuit as an image
Reset zoom to 100%
Fit content to viewpoint
q[0]
q[1]
q[2]
c[...3]
TK1(1,1,0.5)
X
U1(1)
U2(1,0.5)
SX

Advanced Quantum Gates

- Unitary1qBox, Unitary2qBox, Unitary3qBox

Arbitrary 1, 2, and 3 qubit unitary gates can be created using a matrix represented by numpy array. These gates are applied to the circuit using add_unitary1qbox, add_unitary2qbox, and add_unitary3qbox respectively.

Ex. Using Unitary2qBox method, create a 2-qubit gate, and then using Circuit.add_unitary2qbox method, apply the gate to the qubits (q[0], q[1]).

```
from pytket.circuit import Unitary2qBox
u2 = numpy.asarray([[0, 1, 0, 0],[0, 0, 0, -1], [1, 0, 0, 0], [0, 0, -1j, 0]])
u2box = Unitary2qBox(u2)
circ.add_unitary2qbox(u2box, 0,1)    #apply u2box to (q[0], q[1])
```

- CircBox

A circuit you have defined can be boxed as a gate. The gate is applied to the circuit using add_circbox.

Ex. Using CircBox method, create a circuit box of 2 qubits, and then using add_circbox method, apply the box to the qubits (q[1], q[2]).

```
from pytket.circuit import CircBox
sub = Circuit(2)
sub.CX(0, 1).Rz(0.2, 1)
subbox = CircBox(sub)
circ.add_circbox(subbox, [1,2])    #apply subbox to (q[1], q[2])
```

- QControlBox

A box you have defined can be extended in an n-controlled gate. The gate is applied to the circuit using add_qcontrolbox.

Ex. Using QControlBox method, create a controlled gate of the above circuit box of 2 qubits, and then using add_qcontrolbox method, apply the controlled gate with control qubits (q[1], q[2]) and target qubits (q[3], q[4]).

```
from pytket.circuit import QControlBox
qbox = QControlBox(subbox, 2)    #create 2-controlled gate
circ.add_qcontrolbox(qbox, [1,2,3,4])    #apply qbox
```

- Symbolic Gate

Symbolic gates can be constructed in pytket by defining the gate angles as sympy.Symbol or sympy.symbols.

Ex. Apply the rotation gate Rx of angle a defined as sympy.Symbol to the qubit q[0] and specialize the parameter a into 0.3 pi using symbol_substitution method.

```
a = Symbol('alpha')
circ.Rx(a, 0)        #apply symbolic Rx to the qubit q[0]
s_map = {a: 0.3}
circ.symbol_substitution(s_map)        #substitute a to 0.3
```

- Classical Conditional Gate

Any gate can be made conditional by providing the condition at the gate option.
Ex. Apply X gate to q[0] if c[0] is the condition bit 1 and X gate to q[1] if (c[0],c[1]) is the condition bits (0,1) (=2 in decimal notation).
Note that condition_value should be filled in decimal notation. In the case of the condition $(c_0, c_1, \ldots, c_n)$, fill $\sum_{k=0}^{n} c_k 2^k$ in condition_value.

```
circ.X(0, condition_bits=[0], condition_value=1)
circ.X(1, condition_bits=[0,1], condition_value=2)
```

Read circuit information (1)

circ.n_qubits	The number of qubits in the circuit
circ.qubits	A list of all qubits in the circuit
circ.q_registers	A list of all quantum registers
circ.n_bits	The number of classical bits in the circuit
circ.bits	A list of all classical bits in the circuit
circ.c_registers	A list of all classical registers

Read circuit information (2)

circ.n_gates	The number of gates in the circuit
circ.get_commands()	A set of all the commands in the circuit.
circ.get_statevector()	Calculate the unitary matrix applied to $\|0\rangle^{\otimes n}$
circ.get_unitary()	The numerical unitary matrix of the circuit

Read circuit information (3)

circ.depth()	The circuit depth
circ.depth_by_type({OpType set})	The circuit depth of the OpTypes
circ.n_gates_of_type(OpType)	The number of the OpType
circ.n_1qb_gates	The number of gates for 1 qubit
circ.n_2qb_gates	The number of gates for 2 qubits
circ.n_nqb_gates(size=n)	The number of gates for *n* qubits (*n*:int)

Other circuit operations

circ.add_gate(OpType.Reset, [i])	Reset the state q[i] into $\|0\rangle$
circ.add_barrier(qubits list, bits list)	Add a barrier
circ.copy()	Create a copy of the circuit circ
circ.dagger()	Apply the dagger operation on the circuit circ
circ.append(circ1)	Append the circuit circ1 to the circuit circ

Quantum SDK TKET : : CHEAT SHEET

QUANTINUUM

Pipeline of quantum computing

⓪Install pytket and pytket-extensions → ①Create quantum circuit → ②Compile quantum circuit (simplify) → ③Compile quantum circuit (fit device architecture) → ④Execute quantum circuit on quantum device or simulator

The primary goals of compilation are two-fold:
1. Optimising/simplifying your Circuit to make it faster, smaller, and less prone to noise, and
2. Solving the constraints of the Backend to get from your Circuit on the abstract device model to something executable on a real device.

Each step in the compilation can generally be divided into one of these two categories.

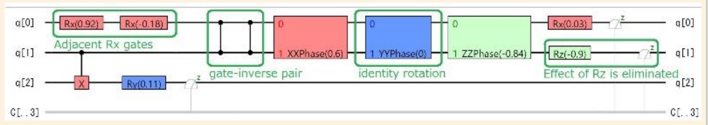
Compile quantum circuit (simplify)

Optimisation passes

Most circuit optimisations follow the sense of 'fewer expensive resources gives less opportunity for noise to creep in'. If we can find an alternative circuit that is observationally equivalent in a perfect noiseless setting but uses fewer resources (gates, time, ancilla qubits), then it is likely to perform better in a noisy context. pytket.passes class provides optimisations for finding many alternative circuits.

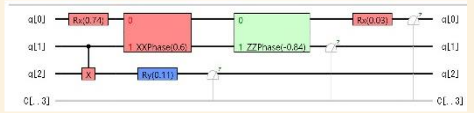
A good example from this class is RemoveRedundancies pass, which removes gate-inverse pairs, merges rotations, removes identity rotations, and removes redundant gates before measurement. For example, we suppose the following circuit circ.

```
from pytket import Circuit
circ = Circuit(3, 3)
circ.Rx(0.92, 0).CX(1, 2).Rx(-0.18, 0)
circ.CZ(0, 1).CZ(0, 1)
circ.add_gate(OpType.XXPhase, 0.6, [0, 1])
circ.add_gate(OpType.YYPhase, 0, [0, 1])
circ.add_gate(OpType.ZZPhase, -0.84, [0, 1])
circ.Rx(0.03, 0).Rz(-0.9, 1).measure_all()
```



The circuit circ includes gates which can be merged (adjacent Rx gates in above) and removed (gate-inverse pair, zero-angle rotation, and Rz gate eliminated by measurement in above).

```
from pytket.passes import RemoveRedundancies
RemoveRedundancies().apply(circ)
```



pytket.passes class provides CliffordSimp pass (looking for specific sequences of Clifford gates and reducing the number of two-qubit gates), EulerAngleReduction pass (representing three rotations in a choice of axes for single-qubit unitaries), and KAKDecomposition pass (using at most three CXs and some single-qubit gates for two-qubit unitaries) etc. which work in the same way.

```
CliffordSimp().apply(circ)
EulerAngleReduction().apply(circ)
KAKDecomposition().apply(circ)
```

Predefined Optimisation Sequences

Knowing what sequences of compiler passes to apply for maximal performance is a very hard problem and can require a lot of experimentation and intuition to predict reliably. pytket.passes provides some predefined sequences which can be applicable to virtually any scenario.

FullPeepholeOptimise pass applies Clifford simplifications, commutes single-qubit gates to the front of the circuit and applies passes to squash subcircuits of up to three qubits. This provides a one-size-approximately-fits-all 'kitchen sink' solution to Circuit optimisation.

```
from pytket.passes import FullPeepholeOptimise
FullPeepholeOptimise().apply(circ)
```

User-defined Optimisation Sequences

pytket allows users to combine passes in a desired order using SequencePass.

For a given circuit circ, we first apply a rebase pass based on the gateset composed of CZ, Rz, and Rx and then use Euler angle decompositions in Rx-Rz-Rx triples as follows.

```
from pytket.passes import auto_rebase_pass, EulerAngleReduction
rebase = auto_rebase_pass({OpType.CZ, OpType.Rz, OpType.Rx})
EARzx = EulerAngleReduction(OpType.Rz, OpType.Rx)
rebase.apply(circ)
EARzx.apply(circ)
```

The optimisation passes defined in above can be combined using SequencePass.

```
from pytket.passes import SequencePass
comp = SequencePass([rebase, EARzx])
comp.apply(circ)
```

User-defined Passes

pytket allows users to define their own custom circuit transformation using CustomPass. Here, we show how to use CustomPass by defining a simple transformation that replaces any Pauli Z gate in the Circuit with a Hadamard gate, Pauli X gate, Hadamard gate chain.

```
def z_transform(cir: Circuit) -> Circuit:
    n_qubits = cir.n_qubits #number of qubits in the input 'cir'
    cir_p = Circuit(n_qubits) #create new circuit
    for cm in cir.get_commands(): #take gate in the input 'cir'
        qubit_list = cm.qubits
        if cm.op.type == OpType.Z:  #replace Z into HXH
            cir_p.add_gate(OpType.H, qubit_list)
            cir_p.add_gate(OpType.X, qubit_list)
            cir_p.add_gate(OpType.H, qubit_list)
        else:
            cir_p.add_gate(cm.op.type, cm.op.params, qubit_list)
    return cir_p
from pytket.passes import CustomPass
DecompseZPass = CustomPass(z_transform) #define your pass
DecompseZPass.apply(circ)
```

Pytket Extension Module

The pytket extensions are separate python modules which allow pytket to interface with backends from a range of providers including quantum devices from Quantinuum and IBM. In pytket, Backend represents a connection to a QPU (Quantum Processing Unit) or simulator for processing quantum circuits. Backend can also access quantum devices and simulators via the cloud of Azure and Braket through the pytket extensions.

In addition, the extensions allow pytket to cross-compile circuits from different quantum computing libraries with the extensions for qiskit, cirq and pennylane. This enables pytket's compilation features to be used in conjunction with other software tools.

The pytket extension modules can be installed adding the extension name to the installation command for pytket. pytket-quantinuum and pytket-qiskit can be installed as follows.

```
pip install pytket-quantinuum
pip install pytket-qiskit
```

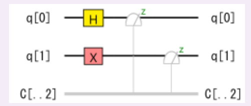
Compile quantum circuit (fit device architecture)

Every device and simulator have some restrictions to allow for a simpler implementation. For example, devices and simulators are typically designed to support only a small (but universal) gate set, so a circuit containing other gate types cannot be run immediately.

The Backend class defines the structure of a backend as something that can run quantum circuits and produce output as at least one of shots, counts, state, or unitary.

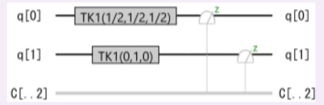
Backend.get_compiled_circuit() solves all of constraints (connectivity, allowed gates, etc.) in Backend when possible (note that conditional gates may not be fixed by compilation), and return a new circuit. We suppose the following circuit.

```
circ = Circuit(2, 2)
circ.H(0).X(1).measure_all()
```



Here, we call IBM simulator AerBackend using pytket-qiskit and compile the circuit circ into the circuit represented by the allowed gates in the backend.

```
from pytket.extensions.qiskit import AerBackend
b = AerBackend()
compiled_circ = b.get_compiled_circuit(circ)
```



Note that get_compiled_circuit() has the optional parameter of optimization levels 0, 1, and 2. The default optimisation level is 2. It can be set as below.

```
compiled_circ = b.get_compiled_circuit(circ, optimisation_level=0)
```

The level of optimisation to perform during compilation.
- Level 0 does the minimum required to solves the device constraints, without any optimisation.
- Level 1 additionally performs some light optimisations.
- Level 2 (the default) adds more computationally intensive optimisations that should give the best results from execution.

Execute quantum circuit on quantum device or simulator

Execute Circuit on Local Simulator

Now that we can prepare our circuit to be suitable for a given Backend, we can send it off to be run and examine the results. The number of shots required is passed to Backend.process_circuit. The result is retrieved using Backend.get_result; and the shots are then given as a table from get_shots.

```
handle = b.process_circuit(compiled_circ, n_shots=1000)
result = b.get_result(handle)
shots = result.get_shots()
```

The dictionary returned by get_counts maps to number of shots.

```
counts = result.get_counts()
```

Execute Circuit on Quantum Device/Simulator on Cloud

pytket provides the pytket.config class to access quantum devices or simulators on a cloud using your access keys. The access keys can be stored and loaded in the pytket configuration file using pytket extension feature.

Here's we show how to set up your IBMQ access token using pytket-qiskit. Note that everyone creates an IBMQ account and has an IBMQ access token here.

```
from pytket.extensions.qiskit import set_ibmq_config
set_ibmq_config(ibmq_api_token='your_ibm_token')
```

If you have an IBMQ premium account, fill in the options.

```
set_ibmq_config (ibmq_api_token='your_ibm_token',
hub='your_hub', group='your_group', project='your_project')
```

Now, you can see information about the available quantum devices and simulators in IBMQ using IBMQBackend.available_devices, if your access token has been successfully set up.

```
from pytket.extensions.qiskit import IBMQBackend
IBMQBackend.available_devices(hub='your_hub',
group='your_group', project='your_project')
```

After selecting an available quantum *device* or *simulator* (ex. 'ibm_oslo') in IBMQBackend as a backend, a circuit can be compiled in the same way as it was done with the local simulator.

```
from pytket.extensions.qiskit import IBMQBackend
b = IBMQBackend('device/simulator', token='your_ibm_token')
compiled_circ = b.get_compiled_circuit(circ)
handle = b.process_circuit(compiled_circ, n_shots=1000)
result = b.get_result(handle)
counts = result.get_counts()
```

A full list of available pytket backends is shown here.
- QuantinuumBackend in pytket-quantinuum allows pytket circuits to be executed on Quantinuum's quantum devices 'H1-Series', these emulators, and these syntax checkers. See pytket-quantinuum example notebookes
- BraketBackend in pytket-braket allows pytket circuits to be excuted on quantum devices in AWS braket. set_braket_config is provided to access the quantum devices.
- AzureBackend in pytket-qsharp allows pytket circuits to be excuted on quantum devices and simulators in Azure Quantum. set_azure_config is provided to access the devices and simulators.