Atos QLM Fast-Start

Introduction to quantum computing

2021/12/02

Gaëtan Rubez
Quantum Computing Expert for the CEPP
gaetan.rubez@atos.net

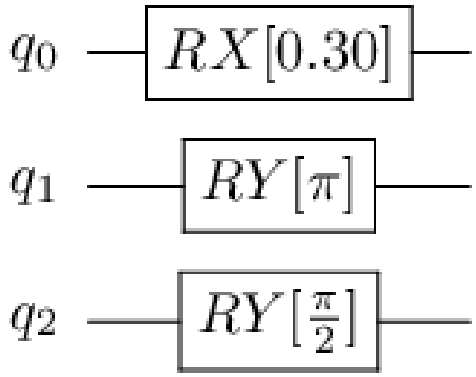
pyAQASM part 2

pyAQASM – Parametrized gates

Gate name	Keyword	Arity
Rotation x-axis	RX[θ]	1
Rotation y-axis	RY[θ]	1
Rotation z-axis	RZ[θ]	1
Phase shift	PH[θ]	1

pyAQASM – Parametrized gates

```
from qat.lang.AQASM import RX, RY
import math

#Apply some quantum Gates
my_program.apply(RX(0.3), qubits_reg[0])
my_program.apply(RY(math.pi), qubits_reg[1])
my_program.apply(RY(math.pi/2), qubits_reg[2])
```



Hands-on 2: Parametrized gates

Hands-on 2: Parametrized gates

- Log on the QLM

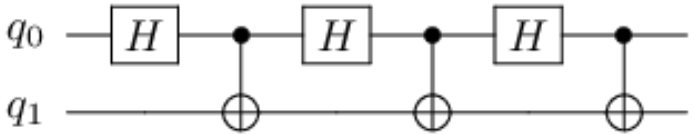
- Go to Hands-on 2 directory.:

  http://127.0.0.1:8888/tree/notebooks/Hands-on2

- Open and complete the notebook *ParametrizedGates.ipynb*

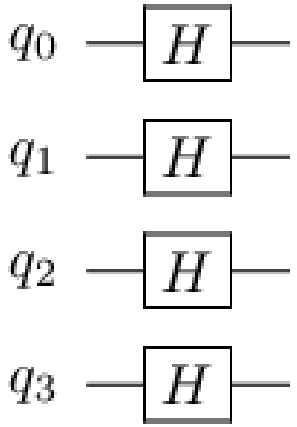
pyAQASM – QRoutine

```
from qat.lang.AQASM import H, CNOT, QRoutine
#Define your routine
my_routine = QRoutine()
my_routine.apply(H, 0)
my_routine.apply(CNOT, 0, 1)

#Use this routine
my_program.apply(my_routine, qubits_reg[0], qubits_reg[1])
my_program.apply(my_routine, qubits_reg[0:2])
my_program.apply(my_routine, qubits_reg)
```



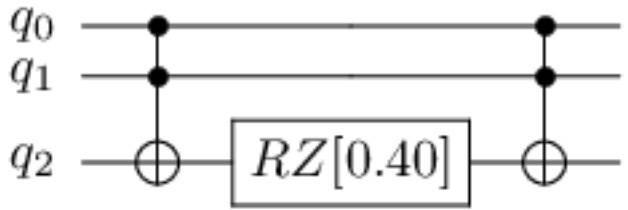
pyAQASM – QRoutine: Walsh Hadamard

```
from qat.lang.AQASM import QRoutine, H
#Define your walsh Hadamard
def WALSH_HADAMARD(n):
    walsh_Hadamard_routine =  QRoutine()
    for i in range(n):
        walsh_Hadamard_routine.apply(H, i)
    return walsh_Hadamard_routine

#Use this routine
my_program.apply(WALSH_HADAMARD(4), qubits_reg)
```



pyAQASM – QRoutine: Ancilla qubits management

```
from qat.lang.AQASM import *
#Create a QRoutine
routine = QRoutine()
#Allocate Qubits using wires
input_wires = routine.new_wires(2)
temp_wire = routine.new_wires(1)
#Apply your gates
routine.apply(CCNOT, input_wires, temp_wire)
routine.apply(RZ(0.4), temp_wire)
routine.apply(CCNOT, input_wires, temp_wire)

%qatdisplay routine
```



pyAQASM – QRoutine: Ancilla qubits management

```
#Defining temporary qubits
routine.set_ancillae(temp_wire)
#Printing arity
print("Now the routine has arity", routine.arity)
#Create a program calling our QRouting
prog = Program()
qbits = prog.qalloc(2)
prog.apply(routine, qbits) # No exceptions!
circ = prog.to_circ(include_locks=True)
print("But the circuit has arity", circ.nbqbits)

%qatdisplay circ
```

pyAQASM – QRoutine: Ancilla qubits management

```
#Defining temporary qubits
routine.set_ancillae(temp_wire)
#Printing arity
print("Now the routine has arity", routine.arity)
#Create a program calling our QRouting
prog = Program()
qbits = prog.qalloc(2)
prog.apply(routine, qbits) # No exceptions!
circ = prog.to_circ(include_locks=True)
print("But the circuit has arity", circ.nbqbits)

%qatdisplay circ
```

```
Now the routine has arity 2
```

pyAQASM – QRoutine: Ancilla qubits management

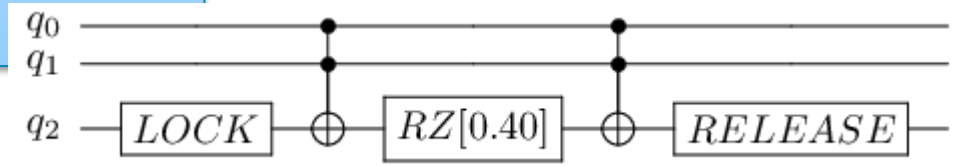
```
#Defining temporary qubits
routine.set_ancillae(temp_wire)
#Printing arity
print("Now the routine has arity", routine.arity)
#Create a program calling our QRouting
prog = Program()
qbits = prog.qalloc(2)
prog.apply(routine, qbits) # No exceptions!
circ = prog.to_circ(include_locks=True)
print("But the circuit has arity", circ.nbqbits)

%qatdisplay circ
```

```
Now the routine has arity 2
But the circuit has arity 3
```

pyAQASM – QRoutine: Ancilla qubits management

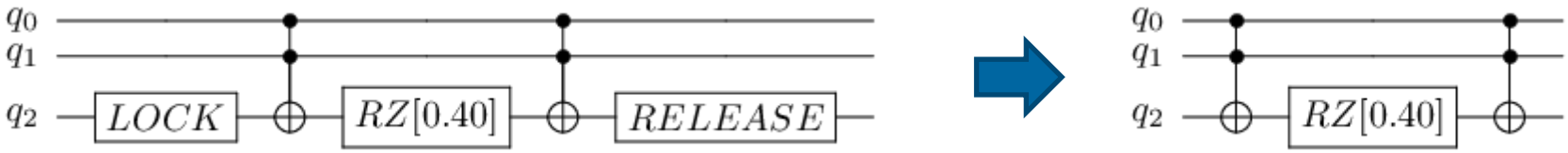
```
#Defining temporary qubits
routine.set_ancillae(temp_wire)
#Printing arity
print("Now the routine has arity", routine.arity)
#Create a program calling our QRouting
prog = Program()
qbits = prog.qalloc(2)
prog.apply(routine, qbits) # No exceptions!
circ = prog.to_circ(include_locks=True)
print("But the circuit has arity", circ.nbqbits)

%qatdisplay circ
```

```
Now the routine has arity 2
But the circuit has arity 3
```



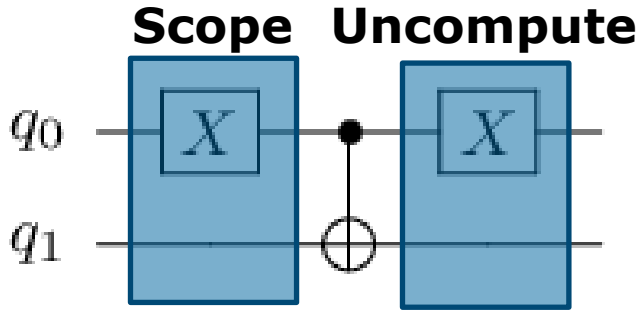
pyAQASM – QRoutine: Ancilla qubits management

```
#Removing locks
circ = circ.remove_locks()
%qatdisplay circ
```

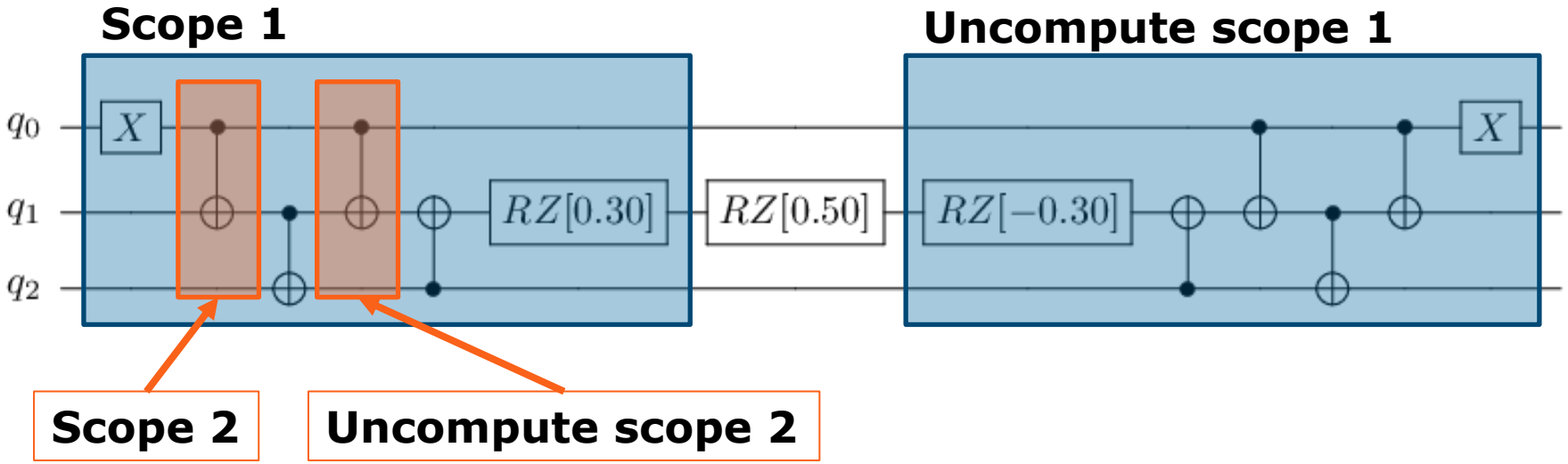


pyAQASM – QRoutine: Computation scopes

```
from qat.lang.AQASM import *

routine = QRoutine()

#Open a fresh "computation scope"
with routine.compute():
    # This gate will be stored in the scope
    routine.apply(X, 0)

#Out of the scope and apply another gate
routine.apply(CNOT, 0, 1)

#Uncompute the last scope
routine.uncompute()
```



pyAQASM – QRoutine: Nested scopes manipulation



Hands-on 3: QRoutines

Hands-on 3: QRoutines

- Log on the QLM

- Go to Hands-on 3 directory:

  http://127.0.0.1:8888/tree/notebooks/Hands-on3

- Open and complete the notebook *QRoutine.ipynb*

Classical bits and measurement

- Allocate classical bits

```
cbits_reg = my_program.calloc(10)
```

- Measure

```
my_program.measure(qbits[0], cbits[0])
```

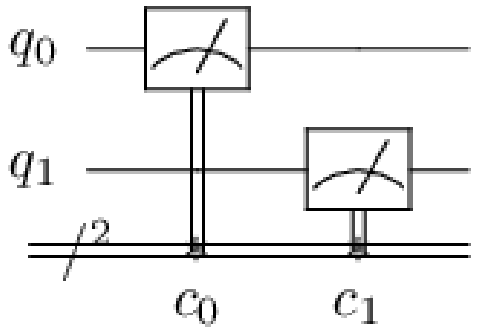
```
my_program.measure(qbits, cbits)
```

Classical bits and measurement

```
from qat.lang.AQASM import Program
#Create a program
prog = Program()
#Allocate qubits
qbits = prog.qalloc(2)
#Allocate bits
cbits = prog.calloc(2)
#Measure qbits in cbits
prog.measure(qbits, cbits)
#Create the circuit and print it
circ = prog.to_circ()
%qatdisplay circ
```

Classical bits and measurement

```
from qat.lang.AQASM import Program
#Create a program
prog = Program()
#Allocate qubits
qbits = prog.qalloc(2)
#Allocate bits
cbits = prog.calloc(2)
#Measure qbits in cbits
prog.measure(qbits, cbits)
#Create the circuit and print it
circ = prog.to_circ()
%qatdisplay circ
```

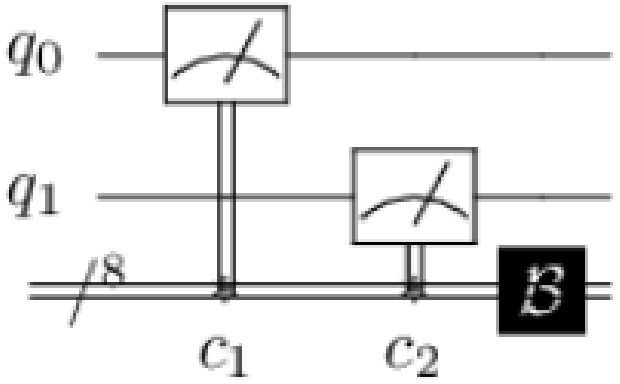


Logic operations on classical bits

```
from qat.lang.AQASM import Program
prog = Program()
qbits = prog.qalloc(2)
cbits = prog.calloc(8)

#Measure qbits in cbits
prog.measure(qbits, cbits[1:3])
#Measure qbits in cbits
prog.logic(cbits[0], cbits[1] & cbits[2])

circ = prog.to_circ()
%qatdisplay circ
```

Logic operations on classical bits

```
from qat.lang.AQASM import Program
prog = Program()
qbits = prog.qalloc(2)
cbits = prog.calloc(8)

#Measure qbits in cbits
prog.measure(qbits, cbits[1:3])
#Measure qbits in cbits
prog.logic(cbits[0], cbits[1] & cbits[2])

circ = prog.to_circ()
%qatdisplay circ
```



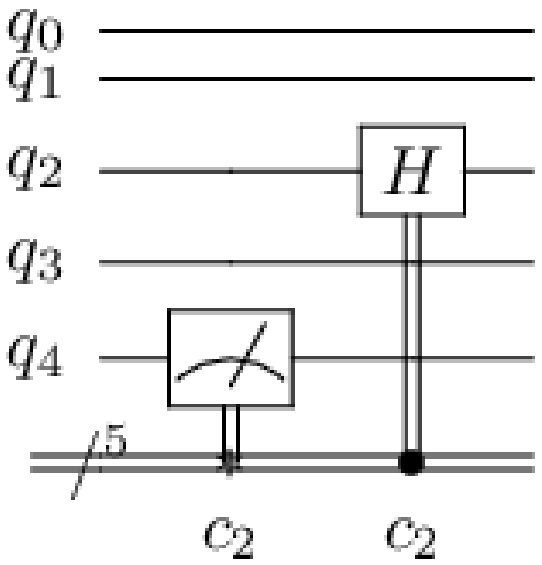
Boolean operators '&', '|', '^', '~'

Classical control

```
from qat.lang.AQASM import Program, H
prog = Program()
cbits = prog.calloc(5)
qbits = prog.qalloc(5)

# Initializing cbits[2]
prog.measure(qbits[4], cbits[2])
# Apply Hadamard only if cbits[2] is set
prog.cc_apply(cbits[2], H, qbits[2])

circ = prog.to_circ()
%qatdisplay circ
```

Classical control

```
from qat.lang.AQASM import Program, H
prog = Program()
cbits = prog.calloc(5)
qbits = prog.qalloc(5)

# Initializing cbits[2]
prog.measure(qbits[4], cbits[2])
# Apply Hadamard only if cbits[2] is set
prog.cc_apply(cbits[2], H, qbits[2])

circ = prog.to_circ()
%qatdisplay circ
```



Qubit reset & Cbit reset

```
from qat.lang.AQASM import Program

prog = Program()
qbits = prog.qalloc(8)
cbits = prog.calloc(8)

# Reseting qubits 1, 3 and 4, and cbit 0.
prog.reset([qbits[1], qbits[3:5]], [cbits[0]])
# Reseting only cbit 1
prog.reset([], [cbits[1]])
```

Qubit reset & Cbit reset

```
from qat.lang.AQASM import Program

prog = Program()
qbits = prog.qalloc(8)
cbits = prog.calloc(8)

# Reseting qubits 1, 3 and 4, and cbit 0.
prog.reset([qbits[1], qbits[3:5]], [cbits[0]])
# Reseting only cbit 1
prog.reset([], [cbits[1]])
```

Be careful: You will not be able to print the circuit afterward

Qubit reset & Cbit reset

```
from qat.lang.AQASM import Program

prog = Program()
qbits = prog.qalloc(8)
cbits = prog.calloc(8)

# Reseting qubits 1, 3 and 4, and cbit 0.
prog.reset([qbits[1], qbits[3:5]], [cbits[0]])
# Reseting only cbit 1
prog.reset([], [cbits[1]])
```

Be careful: You will not be able to print the circuit afterward

Abstract gates

- pyAQASM natively supports 4 parametrized gates (e.g RX, RY, RZ, PH)
- New gates can be defined using AbstractGate
- *There is two main interests :*
  - *Create boxes to implement latter during the development phase*
  - *If you know that the QPU you are target has a specific gate*

Abstract gates

```
from qat.lang.AQASM import Program, AbstractGate
prog = Program()
q = prog.qalloc(2)

# Abstract gates do not require a particular matrix description:
 they are boxes with a name and a signature:
my_gate = AbstractGate("mygate", # The name of the gate
                       [float], # Its signature: (here a single float)
                       arity=2) # Its arity
prog.apply(my_gate(0.3), q[0], q[1])

circ = prog.to_circ()
%qatdisplay circ
```

Abstract gates

```
from qat.lang.AQASM import Program, AbstractGate
prog = Program()
q = prog.qalloc(2)

# Abstract gates do not require a particular matrix description:
 they are boxes with a name and a signature:
my_gate = AbstractGate("mygate", # The name of the gate
                       [float], # Its signature: (here a single float)
                       arity=2) # Its arity
prog.apply(my_gate(0.3), q[0], q[1])

circ = prog.to_circ()
%qatdisplay circ
```

Abstract gates

```
from qat.lang.AQASM import Program, AbstractGate
prog = Program()
q = prog.qalloc(2)

My_CNOT = AbstractGate("MY\_CNOT", [], arity=2,
                matrix_generator=lambda : np.array([[1,0,0,0],
                                                    [0,1,0,0],
                                                    [0,0,0,1],
                                                    [0,0,1,0]]))

prog.apply(My_CNOT(), q[0], q[1])
circ = prog.to_circ()
%qatdisplay circ
```

Abstract gates

```
from qat.lang.AQASM import Program, AbstractGate
prog = Program()
q = prog.qalloc(2)

My_CNOT = AbstractGate("MY\_CNOT", [], arity=2,
                matrix_generator=lambda : np.array([[1,0,0,0],
                                                    [0,1,0,0],
                                                    [0,0,0,1],
                                                    [0,0,1,0]]))

prog.apply(My_CNOT(), q[0], q[1])
circ = prog.to_circ()
%qatdisplay circ
```

Hands-on 4: Abstract Gates

Hands-on 4: Abstract Gates

- Log on the QLM

- Go to Hands-on 4 directory:

  http://127.0.0.1:8888/tree/notebooks/Hands-on4

- Open and complete the notebook *Plugins.ipynb*

Lecture: Plugins

Plugins

Goal: Simplify the design of applications.

Plugins can:

- process circuits (or jobs) on their way to a QPU.
- process samples (or values) on their way back.

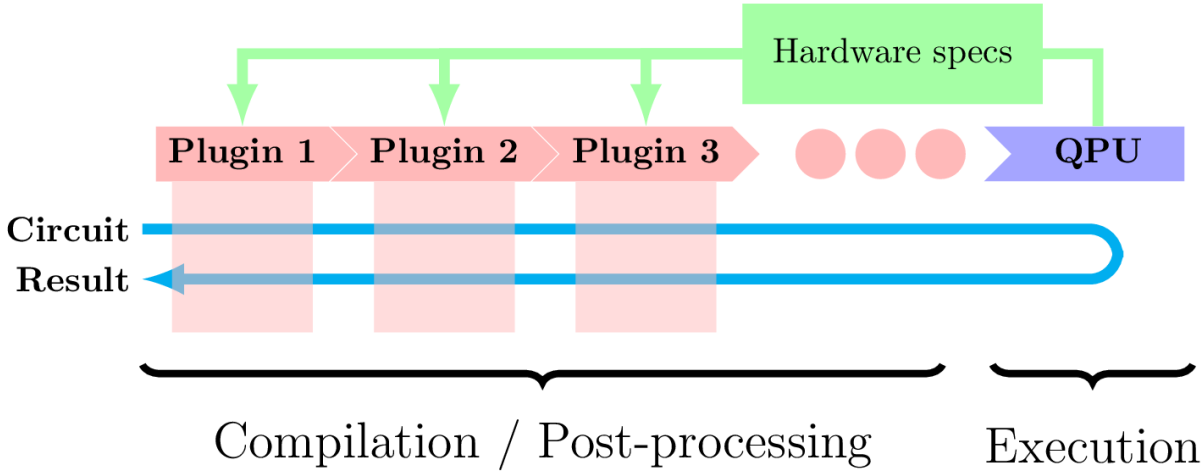
Plugins API is composed of:

- **compile** for the way in. Take a *Batch* with *HardwareSpecs* and return a new *Batch.*
- **post_process** for the way out. Process *BatchResult* and return either a *BatchResult* or a new *Batch* that should go back to the QPU.

Plugins

Creating a stack using plugins:

```
my_stack = plugin1 | plugin2 | … | my_qpu
```



Writing Plugins

```
from qat.core.plugins import AbstractPlugin

class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs):
    #do something with the batch…
        return batch
    def post_process(self, batch_result):
    #do something with the results…
        return batch_result
    def do_post_process(self):
        return True

MyPlugin()
```

Using Plugins

```
from qat.qpus import LinAlg

my_stack = MyPlugin() | LinAlg()

from qat.lang.AQASM import Program, H

prog = Program()
for qb in prog.qalloc(3):
    prof.apply(H, qb)

for sample in
my_stack.submit(prog.to_circ().to_job()):
    print(sample)
```

Using Plugins

```
from qat.qpus import LinAlg

my_stack = MyPlugin() | LinAlg()

from qat.lang.AQASM import Program, H

prog = Program()
```

```
Sample(state=|000>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|100>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|010>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|110>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|001>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|101>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|111>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|011>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
```

Example of Plugin

```
class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs):
        for i, job in enumerate(batch.jobs):
            print(">> Job #{}".format(i)):
            for op in job.circuit.iterate_simple():
                print(op)
        return batch

    def post_process(self, batch_result):
        for result in batch_result.results:
            print('Result of size', len(result.raw_data))
        return batch_result
```

Example of Plugin

```
class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs):
        for i, job in enumerate(batch.jobs):
            print(">> Job #{}".format(i)):
            for op in job.circuit.iterate_simple():
                print(op)
        return batch

    def post_process(self, batch_result):
        for result in batch_result.results:
            print('Result of size', len(result.raw_data))
        return batch_result
```

```
job = prog.to_circ().to_job()
# Let's submit 3 times our job in a
# single go
for sample in my_stack.submit([job]*3):
    print(sample)
```

Example of Plugin

```
>> Job #0
('H', [], [0])
('H', [], [1])
('H', [], [2])
>> Job #1
('H', [], [0])
('H', [], [1])
('H', [], [2])
>>Job #2
('H', [], [0])
('H', [], [1])
('H', [], [2])
Result of size 8
Result of size 8
Result of size 8
Result(raw_data=[Sample(state=|000>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
Result(raw_data=[Sample(state=|100>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
Result(raw_data=[Sample(state=|000>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
```

List of implemented Plugins

Nnizer: swap insertion plugin
PatternManager: a pattern-based quantum circuit rewriter
Graphopt: pattern & phase polynomial-based circuit optimizer
VariationalOptimizer: a plugin for variational algorithms
ObservableSplitter: turning observable sampling into qubit sampling
CircuitInliner: inlining circuit inside a stack
Display: a console displayer plugin
QuameleonPlugin: emulating hardware constraints via a plugin

Hands-on 5: Plugins

Hands-on 5: Plugins

- Log on the QLM

- Go to Hands-on 5 directory:

  http://127.0.0.1:8888/tree/notebooks/Hands-on5

- Open and complete the notebook *Plugins.ipynb*

Thank you.

Gaëtan Rubez

Quantum Computing Expert for the CEPP

gaetan.rubez@atos.net

Atos