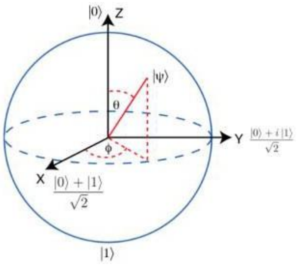
Bull myQLM

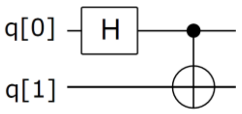
Cheat sheet

Quantum Hello World: Creation of an EPR pair



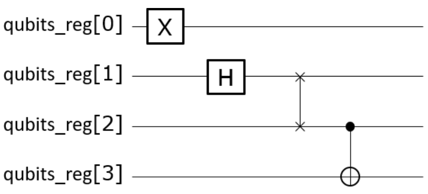
[myQLM – Quantum Python Package – myQLM documentation]

```
from qat.lang.AQASM import *
prog = Program()
qbits = prog.qalloc(2) prog.apply(H,
qbits[0]) prog.apply(CNOT,
qbits[0],qbits[1]) circuit = prog.to_circ()
%qatdisplay circuit --svg
```



Writing and Executing Circuits in pyAQASM

```
from qat.lang.AQASM import Program, X, H, CNOT, SWAP
#Create a Program
my_program = Program()
#Allocate some qubits
qubits_reg = my_program.qalloc(4)
#Apply some quantum Gates
my_program.apply(X, qubits_reg[0])
my_program.apply(H, qubits_reg[1])
my_program.apply(SWAP, qubits_reg[1], qubits_reg[2])
my_program.apply(CNOT, qubits_reg[2], qubits_reg[3])
#Export this program into a quantum circuit  my_circuit =
my_program.to_circ()
#And display it!
my_circuit.display()
```



`print(statistics(circ))` to see the number of qubits, the number of gates, etc...

`CLinalg` can be used as well from myQLM 1.7.0

```
#import one Quantum Processor Unit Factory
from qat.qpus import PyLinalg
#Create a Quantum Processor Unit
linalgqpu = PyLinalg()
#Create a job
job = my_circuit.to_job() #Submit
the job to the QPU result =
linalgqpu.submit(job)
#Iterate over the final state vector to get all final components for
sample in result:
   print("State %s amplitude %s" % (sample.state,
sample.amplitude))
```

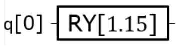
Import functions
Get a simulator
Create your job
Submit your job
Print the result

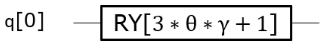
```
State |1000> amplitude (0.7071067811865475+0j)
State |1011> amplitude (0.7071067811865475+0j)
```

Parameterized Circuits – Syntax to Create & Bind variables

```
from qat.lang.AQASM import
RY  #Define your variables
prog=Program()
theta = prog.new_var(float, "\\theta") gamma =
prog.new_var(float, "\\gamma") #Apply a gate
with a variable prog.apply(RY(3 * theta *
gamma +1), qubits_reg[0])
```

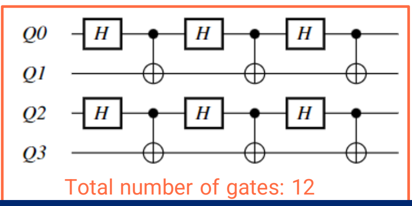
```
new_circuit = prog.to_circ().bind_variables({"\\theta": 0.5,
"\\gamma": 0.1})
```



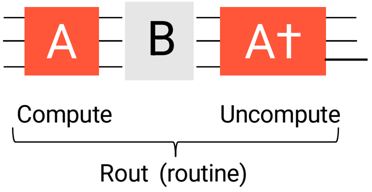


QRoutine represent subcircuits that behave as a Gate object

```
from qat.lang.AQASM import Program, QRoutine, H, CNOT
routine = QRoutine()
routine.apply(H, 0)
routine.apply(CNOT, 0, 1)
prog = Program()
qbits = prog.qalloc(4)
for _ in range(3):
    for bl in range(2):
        prog.apply(routine, qbits[2*bl:2*bl+2]) circ
= prog.to_circ()
circ.display()
print("total number of gates: ", len(circ.ops))
```




```
rout = QRoutine() with
rout.compute():
    #do computation A #
do computation B
rout.uncompute()  #undo A
```



Compute/Uncompute, in addition to simplifying code writing, also simplifies routine manipulation.
For example, if you control your routine, you will only control B.

Job – Batch

A **job** consists of 2 components: circuit and final measurement
A **batch** contains a list of jobs that allows to send several circuits to a QPU with only a single request to the QPU

Measurement has 2 types:
- « Sample » for measuring qubits on the Z axis
- « Observable » for measuring a physical observable such as a Hamiltonian

Syntax for qubit « Sample » mode:
- job = circuit.to_job() #Infinite number of shots, equivalent to "nbshots=0", all qubits
- job = circuit.to_job(nbshots=100) #Finite number of shots, all qubits
- job = circuit.to_job(nbshots=0, qubits=[1]) #Infinite number of shots, only one qubit, the qubit 1 here

Syntax for « Observable » mode:
job = circuit.to_job(observable=obs) #Infinite number of shots
job = circuit.to_job(observable=obs, nbshots=nbshots) #Finite number of shots

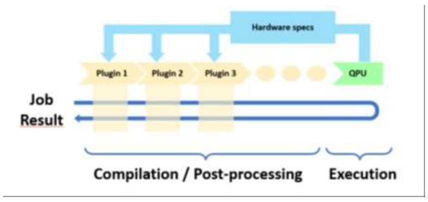
See "Observables & Hamiltonians" paragraph for the definition of obs (observable)

Many variational algorithms require computing the gradient of the cost function E. The gradient can be used in gradient-based optimization methods. QLM jobs come with methods to compute the derivative of E automatically: `differentiate()` and `gradient()`. Examples of use of this feature are given in:
https://mybinder.org/v2/gh/myQLM/myqlm-notebooks/HEAD?filepath=tutorials%2Flang%2Fdifferentiating_jobs.ipynb

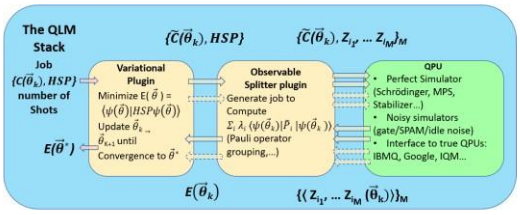
If a QPU does not natively support observable sampling, we can use ObservableSplitter to transform it into qubit sampling tasks and get the final expected value of the observable

Plugins

Plugins are objects that can process a flow of quantum jobs on their way to a QPU, and/or process a flow of information (samples or values) on their way back from a QPU.



The following plugins are offered to the users in myQLM: https://myqlm.github.io/04_api_reference/module_qat/module_plugins.html#qat-plugins



Plugin for transforming Observable sampling into Qubit sampling: https://myqlm.github.io/04_api_reference/module_qat/module_plugins/observablesplitter.html#qat.plugins.ObservableSplitter

Using Plugins

```
from qat.qpus import PyLinalg
my_stack = MyPlugin() | PyLinalg()
from qat.lang.AQASM import Program, H
prog = Program()
for qb in prog.qalloc(3):
    prog.apply(H, qb)
for sample in my_stack.submit(prog.to_circ().to_job()):
    print(sample)
```

We can also create a plugin for a specific purpose

Writing Plugins

```
from qat.core.plugins import AbstractPlugin
class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs): #do
        something with the batch… return batch
MyPlugin()
```

Interoperability

myQLM provides binders to connect myQLM to other Python-based quantum frameworks such as Qiskit, Cirq:

https://myqlm.github.io/01_getting_started/:myqlm:01_install.html#interoperability

Example for Qiskit

```
from qat.interop.qiskit import qiskit_to_qlm
qlm_circuit = qiskit_to_qlm(your_qiskit_circuit)

from qat.interop.qiskit import qlm_to_qiskit
qiskit_circuit = qlm_to_qiskit(your_qlm_circuit)
```

Observables & Hamiltonians

```
from qat.core import Observable, Term
my_observable = Observable(4, # A 4 qubits observable
                pauli_terms=[
                    Term(1., "ZZ", [0, 1]),
                    Term(4., "XZ", [2, 0]),
                    Term(3., "ZXZX", [0, 1, 2, 3])
                ],
                constant_coeff=23.)
print(my_observable)
```

Pauli_terms = Z0 Z1 + 4 X2 Z0 + 3 Z0 X1 Z2 X3
Observable = 23 I4 + Pauli_terms

New pauli term can be added using Observable.add_term

Observables can be added (obs1+obs2) and multiplied by a scalar λ*obs
Hamiltonian can be defined using observable

```
from qat.fermion.hamiltonians import SpinHamiltonian
nqbits = 4
H = SpinHamiltonian(nqbits, pauli_terms)
```

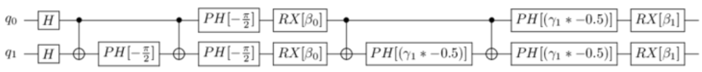
Your own spin & fermionic Hamiltonians can be built using various Hamiltonian classes
https://myqlm.github.io/04_api_reference/module_qat/module_fermion.html#module-qat.fermion.hamiltonians

Combinatorial Problems

```
from qat.opt import CombinatorialProblem
my_problem = CombinatorialProblem()
v0 = my_problem.new_var()    v1
= my_problem.new_var()
#v_array = my_problem.new_vars(4)
my_problem.add_clause(v0 |v1, weight=2.)    for
clause, weight in my_problem.clauses:
   print(clause, weight)
obs = my_problem.get_observable()
print(obs)
import numpy as np
depth = 2
ansatz = my_problem.qaoa_ansatz(depth).circuit    ansatz_gamma_0_pi =
ansatz.bind_variables({"\\gamma_{0}": np.pi})    ansatz_gamma_0_pi.display()
```

For a maximization problem:
CombinatorialProblem(maximization =True)

```
V(0) | V(1) 2.0
1.5 * I^2 +
-0.5 * (Z|[0]) +
-0.5 * (Z|[1]) +
-0.5 * (ZZ|[0, 1])
```



Open Source Libraries : Chemistry, Finance, …

Open Source Libraries for MyQLM
containing finance, chemistry and other modules

👉 https://github.com/neasqc

Arbitrary 1qubit gate & multi-qubit gates using matrix

```
from qat.lang.AQASM import AbstractGate
import numpy as np
def U3_generator(theta, phi, lamda):
  return np.array([  [np.cos(theta/2),              -np.exp(1j*lamda)*np.sin(theta/2)],
            [np.exp(1j*phi)*np.sin(theta/2), np.exp(1j*(lamda+phi))*np.cos(theta/2)]  ])
U3 = AbstractGate("U3", [float, float, float], arity=1, matrix_generator=U3_generator)
```

$$U_3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{pmatrix}$$

We can define a set_dag, if not, the standard recursive structure will be used for dag

CNOT example for syntax

```
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]]))
```