XACC Documentation

Release 1.0.0

Alex McCaskey

May 24, 2021

CONTENTS:

XACC (pronounced as it's spelled) is an extensible compilation framework for hybrid quantum-classical computing architectures. It provides extensible language frontend and hardware backend compilation components glued together via a novel, polymorphic quantum intermediate representation. XACC currently supports quantum-classical programming and enables the execution of quantum kernels on IBM, Rigetti, IonQ, and D-Wave QPUs, as well as a number of quantum computer simulators.

The XACC programming model follows the traditional co-processor model, akin to OpenCL or CUDA for GPUs, but takes into account the subtleties and complexities inherent to the interplay between classical and quantum hardware. XACC provides a high-level API that enables classical applications to offload work (represented as quantum kernels) to an attached quantum accelerator in a manner that is independent to the quantum programming language and hardware. This enables one to write quantum code once, and perform benchmarking, verification and validation, and performance studies for a set of virtual (simulators) or physical hardware.

CHAPTER

ONE

MODULAR INFRASTRUCTURE

XACC relies on a project called CppMicroServices - a native C++ implementation of the OSGi specification that enables an extensible, modular plugin infrastructure for quantum compilers and accelerators.

CHAPTER

TWO

DESCRIPTION OF ARCHITECTURE

For a comprehensive discussion of all components of the XACC programming model and architecture, please refer to this manuscript.

For class documentation, check out this site.

CHAPTER

THREE

XACC DEVELOPMENT TEAM

XACC is developed and maintained by:

- Alex McCaskey
- Travis Humble
- Eugene Dumitrescu
- Dmitry Liakh
- Mengsu Chen
- Zach Parks
- Ryan Sand
- Charles Zhao
- Jay Billings
- Thien Nguyen
- Daniel Chaves-Claudino
- Tyler Kharazi
- Anthony Santana

CHAPTER

FOUR

QUESTIONS, BUG REPORTING, AND ISSUE TRACKING

Questions, bug reporting and issue tracking are provided by GitHub. Please report all bugs by creating a new issue. You can ask questions by creating a new issue with the question tag.

4.1 Installation

Note that you must have a C++14 compliant compiler and a recent version of CMake (version 3.12+). We recommend installing with the Python API (although you may choose not to). This discussion will describe the build process with the Python API enabled. For this you will need a Python 3 executable and development install. To interact with remote QPUs, you will need CURL with OpenSSL development headers and libraries.

4.1.1 Quick-Start with Docker

To get up and running quickly and avoid installing the prerequisites you can pull the `xacc/xacc` Docker image (see here for instructions). This image provides an Ubuntu 18.04 container that serves up an Eclipse Theia IDE. XACC is already built and ready to go. Moreover, there are several variants: `xacc/xacc-tnqvm-exatn`, and `xacc/xacc-quac` to name a few.

4.1.2 Prerequisites

4.1.2.1 Ubuntu 16.04

Here we will demonstrate installing from a bare Ubuntu install using GCC 8. We install BLAS and LAPACK as well, which is required to build some optional simulators. We install libunwind-dev which is also optional, but provides verbose stack-trace printing upon execution error.

```
$ sudo apt-get update && sudo apt-get install -y software-properties-common
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test && sudo apt-get update
$ sudo apt-get -y install gcc-8 g++-8 git libcurl4-openssl-dev python3 libunwind-dev \
        libpython3-dev python3-pip libblas-dev liblapack-dev
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 50
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-8 50
```

4.1.2.2 Ubuntu 18.04

Here we will demonstrate installing from a bare Ubuntu install using GCC 7 (default on 18.04). We install BLAS and LAPACK as well, which is required to build some optional simulators. We install libunwind-dev which is also optional, but provides verbose stack-trace printing upon execution error.

```
$ sudo apt-get update
$ sudo apt-get -y install gcc g++ git libcurl4-openssl-dev python3 libunwind-dev \
        libpython3-dev python3-pip libblas-dev liblapack-dev
```

4.1.2.3 Centos 7

Here we will demonstrate installing from a bare Centos 7 install using GCC 8. We install BLAS and LAPACK as well, which is required to build some optional simulators.

```
$ sudo yum install libcurl-devel python3-devel git centos-release-scl make \
       devtoolset-8-gcc devtoolset-8-gcc-c++ blas-devel lapack-devel
$ scl enable devtoolset-8 -- bash [ you might put this in your .bashrc ]
```

4.1.2.4 Fedora 30

Here we will demonstrate installing from a bare Fedora 30 install using GCC 9. We install BLAS and LAPACK as well, which is required to build some optional simulators.

```
$ sudo dnf install python3-devel libcurl-devel git g++ gcc make blas-devel lapack-devel
$ sudo python3 -m pip install cmake
```

4.1.2.5 Mac OS X

On Mac OS X, we recommend our users install GCC 8 via Homebrew instead of relying on XCode command line tools installation and the default Apple Clang compilers

Too often we see our users on Mac with issues regarding missing standard includes like *wchar.h* and others. This is ultimately due to an improper install of XCode (see here). If you do not see these issues with Apple Clang, feel free to use it, XACC will build fine. But if you do see these issues, the easiest thing to do is to use Homebrew GCC 8.

```
$ brew install gcc@8
$ export CC=gcc-8
$ export CXX=g++-8
```

Note these last exports are very important. You could also run CMake (see below) with these variables set

```
$ CC=gcc-8 CXX=g++-8 cmake ..
```

You will need to make sure to do this for all plugins / projects that build off of XACC. You will see errors if you accidentally build other projects leveraging XACC (like tnqvm) with compilers different than what was used to build XACC.

You will also need the following 3rd party dependencies

```
$ brew install python3 openssl curl
```

4.1.3 Build XACC

The best way to install a recent version of CMake is through Python Pip.

```
$ sudo python3 -m pip install cmake
```

Now clone and build XACC

```
$ git clone https://github.com/eclipse/xacc
$ cd xacc && mkdir build && cd build
[ note tests and examples are optional ]
$ cmake .. -DXACC_BUILD_TESTS=TRUE -DXACC_BUILD_EXAMPLES=TRUE
$ make -j$(nproc --all) install
[ run tests with ]
$ ctest --output-on-failure
[ some examples executables are in build/quantum/examples ]
$ quantum/examples/base_api/bell_quil_ibm_local
```

You can run Python examples as well

```
[ you may also want to add this to your .bashrc ]
$ export PYTHONPATH=$PYTHONPATH:$HOME/.xacc
$ python3 ../python/examples/ddcl_example.py
```

Most users build and install the TNQVM Accelerator

```
$ git clone https://github.com/ornl-qci/tnqvm
$ cd tnqvm && mkdir build && cd build
$ cmake .. -DXACC_DIR=$HOME/.xacc
$ make -j$(nproc --all) install
```

4.2 Basics

Here we demonstrate leveraging the XACC framework for various quantum-classical programming tasks. We provide examples in both C++ and Python.

4.2.1 Accelerator Buffer

The `AcceleratorBuffer` represents a register of qubits. Programmers allocate this register of a certain size, and pass it by reference to all execution tasks. These execution tasks are carried out by concrete instances of the *Accelerator* interface, and these instances are responsible for persisting information to the provided buffer reference. This ensures programmers have access to all execution results and metadata upon execution completion.

Programmers can allocate a buffer through the `xacc::qalloc(const int n)` (`xacc.qalloc(int)` in Python) call. After execution, measurement results can be queried as well as backend-specific execution metadata. Below demonstrate some basic usage of the `AcceleratorBuffer`

```
#include "xacc.hpp"
...
// Create program somehow... detailed later
program = ...
auto buffer = xacc::qalloc(3);
```

(continues on next page)

(continued from previous page)

```
auto qpu = xacc::getAccelerator("ibm:ibmq_valencia");
qpu->execute(buffer, program);
std::map<std::string, int> results = buffer->getMeasurementCounts();
auto fidelities = (*buffer)["1q-gate-fidelities"].as<std::vector<double>>();
auto expValZ = buffer->getExpectationValueZ();
```

in Python

```
import xacc
...
// Create program somehow... detailed later
program = ...
buffer = xacc.qalloc(3)
qpu = xacc.getAccelerator('ibm:ibmq_valencia', {'shots':8192})
qpu.execute(buffer, program)
results = buffer.getMeasurementCounts()
fidelities = buffer['1q-gate-fidelities']
expValZ = buffer.getExpectationValueZ()
```

4.2.2 Intermediate Representation, Kernels, and Compilers

Above we mentioned a `program` variable but did not detail how it was created. This instance is represented in XACC as a `CompositeInstruction`. The creation of `Instruction` and `CompositeInstruction` is demonstrated below. First, we create this instances via an implementation of the `IRProvider`, specifically a 3 instruction circuit with one parameterized `Ry` on a variable `theta`.

```
#include "xacc.hpp"
...
auto provider = xacc::getIRProvider("quantum");
auto program = provider->createComposite("foo", {"theta"});
auto x = provider->createInstruction("X", {0});
auto ry = provider->createInstruction("Ry", {1}, {"theta"});
auto cx = provider->createInstruction("CX", {1,0});
program->addInstructions({x, ry, cx});
```

in Python

```
import xacc
...
provider = xacc.getIRProvider('quantum')
program = provider.createComposite('foo', ['theta'])
x = provider.createInstruction('X', [0])
ry = provider.createInstruction('Ry', [1], ['theta'])
cx = provider.createInstruction('CX', [1,0])
program.addInstructions([x,ry,cx])
```

We could also create IR through textual source code representations in a language that is available to the framework. Availability here implies that there exists a `Compiler` implementation for the language being used. `Compilers` take kernel source strings and produce IR (one or many `CompositeInstructions`). Here we demonstrate the same circuit as above, but using a Quil kernel

```
#include "xacc.hpp"
...
auto qpu = xacc::getAccelerator("ibm");
auto quil = xacc::getCompiler("quil");
auto ir = quil->compile(R"(
__qpu__ void ansatz(AcceleratorBuffer q, double x) {
    X 0
    Ry(x) 1
    CX 1 0
}
__qpu__ void X0X1(AcceleratorBuffer q, double x) {
    ansatz(q, x)
    H 0
    H 1
    MEASURE 0 [0]
    MEASURE 1 [1]
}
)", qpu);
auto ansatz = ir->getComposite("ansatz");
auto x0x1 = ir->getComposite("X0X1");
```

in Python

```
import xacc
...
qpu = xacc.getAccelerator('ibm')
quil = xacc.getCompiler('quil')
ir = quil.compile('''
__qpu__ void ansatz(AcceleratorBuffer q, double x) {
    X 0
    Ry(x) 1
    CX 1 0
}
__qpu__ void X0X1(AcceleratorBuffer q, double x) {
    ansatz(q, x)
    H 0
    H 1
    MEASURE 0 [0]
    MEASURE 1 [1]
}
''', qpu)
ansatz = ir.getComposite('ansatz')
x0x1 = ir.getComposite('X0X1')
```

Here, `x0x1` is a `CompositeInstruction` that can be passed to `Accelerator::execute()` for backend execution.

Next we demonstrate how one might leverate `IRTransformation` to perform general optimizations on IR instances.

```
#include "xacc.hpp"
...
auto xasmCompiler = xacc::getCompiler("xasm");
auto ir = xasmCompiler->compile(R"(__qpu__ void bell(qbit q) {
    H(q[0]);
    CX(q[0], q[1]);
```

(continues on next page)

(continued from previous page)

```
    CX(q[0], q[1]);
    CX(q[0], q[1]);
    Measure(q[0]);
    Measure(q[1]);
})", nullptr);
auto f = ir->getComposite("bell");
assert(6 == f->nInstructions());

auto opt = xacc::getIRTransformation("circuit-optimizer");
opt->apply(f, nullptr);

assert (4 == f->nInstructions());
```

in Python

```
import xacc
...
# Create a bell state program with too many cnots
xasm = xacc.getCompiler('xasm')
ir = xasm.compile('''__qpu__ void bell(qbit q) {
H(q[0]);
CX(q[0],q[1]);
CX(q[0],q[1]);
CX(q[0], q[1]);
Measure(q[0]);
Measure(q[1]);
}''')
f = ir.getComposite('bell')
assert(6 == f.nInstructions())

# Run the circuit-optimizer IRTransformation
optimizer = xacc.getIRTransformation('circuit-optimizer')
optimizer.apply(f, None, {})

# should have 4 instructions, not 6
assert(4 == f.nInstructions())
print(f.toString())
```

4.2.3 Observable

The `Observable` concept in XACC dictates measurements to be performed on unmeasured an `CompositeInstruction`. XACC provides `pauli` and `fermion` `Observable` implementations. Below we demonstrate how one might create these objects.

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
...
auto x0x1 = xacc::quantum::getObservable("pauli");
x0x1->fromString('X0 X1');

// observe() returns a list of measured circuits
```

(continues on next page)

(continued from previous page)

```
// here we only have one
auto measured_circuit = x0x1->observe(program)[0];

auto fermion = xacc::getObservable("fermion");
fermion->fromString("1^ 0");
auto jw = xacc::getService<ObservableTransform>("jordan-wigner");
auto spin = jw->transform(fermion);
```

in Python

```
import xacc
...
x0x1 = xacc.getObservable('pauli', 'X0 X1')

// observe() returns a list of measured circuits
// here we only have one
measured_circuit = x0x1.observe(program)[0]

fermion = xacc.getObservable('fermion', '1^ 0')
jw = xacc.getObservableTransform('jordan-wigner')
spin = jw.transform(fermion)
```

4.2.4 Accelerator

The `Accelerator` is the primary interface to backend quantum computers and simulators for XACC. The can be initialized with a heterogeneous map of input parameters, expose qubit connectivity information, and implement execution capabilities given a valid `AcceleratorBuffer` and `CompositeInstruction`. Here we demonstrate getting reference to an `Accelerator` and using it to execute a simple bell state. Note this is a full example, that leverages the `xasm` compiler as well as requisite C++ framework initialization and finalization.

```
#include "xacc.hpp"
int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  // Get reference to the Accelerator
  auto accelerator =
    xacc::getAccelerator("qpp", {std::make_pair("shots", 5000)});

  // Allocate some qubits
  auto buffer = xacc::qalloc(2);

  auto xasmCompiler = xacc::getCompiler("xasm");
  auto ir = xasmCompiler->compile(R"(__qpu__ void bell(qbit q) {
      H(q[0]);
      CX(q[0], q[1]);
      Measure(q[0]);
      Measure(q[1]);
  })", accelerator);

  accelerator->execute(buffer, ir->getComposites()[0]);
```

(continues on next page)

(continued from previous page)

```
  buffer->print();

  xacc::Finalize();

  return 0;
}
```

in Python

```
import xacc

accelerator = xacc.getAccelerator('qpp', {'shots':5000})
buffer = xacc.qalloc(2)
xasm = xacc.getCompiler('xasm')
ir = xasm.compile('''__qpu__ void bell(qbit q) {
H(q[0]);
CX(q[0],q[1]);
Measure(q[0]);
Measure(q[1]);
}''', accelerator)

accelerator.execute(buffer, ir.getComposites()[0])
# note accelerators can execute lists of CompositeInstructions too
# usefule for executing many circuits with one remote qpu call
# accelerator.execute(buffer, ir.getComposites())
```

4.2.5 Optimizer

This abstraction is meant for the injection of general classical multi-variate function optimization routines. XACC provides implementations leveraging NLOpt and MLPack C++ libraries. `Optimizer``s expose an ``optimize()` method that takes as input an `OptFunction`, which serves as a thin wrapper for functor-like objects exposing a specific argument structure (must take as first arg a `vector<double>` representing current iterate's parameters, and another one representing the mutable gradient vector). Below is a demonstration of how one might use this utility:

```
auto optimizer =
   xacc::getOptimizer("nlopt");

optimizer->setOptions(
   HeterogeneousMap{std::make_pair("nlopt-maxeval", 200),
                    std::make_pair("nlopt-optimizer", "l-bfgs")});
OptFunction f(
   [](const std::vector<double> &x, std::vector<double> &grad) {
     if (!grad.empty()) {
       grad[0] = -2 * (1 - x[0]) + 400 * (std::pow(x[0], 3) - x[1] * x[0]);
       grad[1] = 200 * (x[1] - std::pow(x[0],2));
     }
     return = 100 * std::pow(x[1] - std::pow(x[0], 2), 2) + std::pow(1 - x[0], 2);
   },
   2);

auto result = optimizer->optimize(f);
```

(continues on next page)

(continued from previous page)

```
auto opt_val = result.first;
auto opt_params = result.second;
```

or in Python

```
def rosen_with_grad(x):
    g = [-2*(1-x[0]) + 400.*(x[0]**3 - x[1]*x[0]), 200 * (x[1] - x[0]**2)]
    xx = (1.-x[0])**2 + 100*(x[1]-x[0]**2)**2
    return xx, g

optimizer = xacc.getOptimizer('mlpack',{'mlpack-optimizer':'l-bfgs'})
opt_val, opt_params = optimizer.optimize(rosen_with_grad,2)
```

4.2.6 xacc::qasm()

To improve programming efficiency, readability, and utility of the quantum kernel string compilation, XACC exposes a `qasm()` function. This function takes as input an enhanced quantum kernel source string syntax and compiles it to XACC IR. This source string is *enhanced* in that it requires that extra metadata be present in order to adequately compile the quantum code. Specifically, the source string must contain the following key words:

- a single *.compiler NAME*, to indicate which XACC compiler implementation to use.
- one or many *.circuit NAME* calls to give the created CompositeInstruction (circuit) a name.
- one *.parameters PARAM 1, PARAM 2, .., PARAM N* for each parameterized circuit to tell the Compiler the names of the parameters.
- A *.qbit NAME* optional keyword can be provided when the source code itself makes reference to the `qbit` or `AcceleratorBuffer`

Running this command with the appropriately provided keywords will compile the source string to XACC IR and store it an internal compilation database (standard map of CompositeInstruction names to CompositeInstructions), and users can get reference to the individual CompositeInstructions via an exposed `getCompiled()` XACC API call. The code below demonstrates how one would use `qasm()` and its overall utility.

```
#include "xacc.hpp"
...
xacc::qasm(R"(
.compiler xasm
.circuit deuteron_ansatz
.parameters x
.qbit q
for (int i = 0; i < 2; i++) {
  H(q[0]);
}
exp_i_theta(q, x, {{"pauli", "X0 Y1 - Y0 X1"}});
)");
auto ansatz =
  xacc::getCompiled("deuteron_ansatz");

// Quil example, multiple kernels
xacc::qasm(R"(.compiler quil
.circuit ansatz
```

(continues on next page)

(continued from previous page)

```
.parameters theta, phi
X 0
H 2
CNOT 2 1
CNOT 0 1
Rz(theta) 0
Ry(phi) 1
H 0
.circuit x0x1
ansatz(theta, phi)
H 0
H 1
MEASURE 0 [0]
MEASURE 1 [1]
)");
auto x0x1 = xacc::getCompiled("x0x1");
```

or in Python

```
import xacc
...
 xacc.qasm('''
.compiler xasm
.circuit deuteron_ansatz
.parameters x
.qbit q
for (int i = 0; i < 2; i++) {
  X(q[0]);
}
exp_i_theta(q, x, {{"pauli", "X0 Y1 - Y0 X1"}});
''')
ansatz =
 xacc.getCompiled('deuteron_ansatz')

# Quil example, multiple kernels
xacc.qasm('''.compiler quil
.circuit ansatz
.parameters theta, phi
X 0
H 2
CNOT 2 1
CNOT 0 1
Rz(theta) 0
Ry(phi) 1
H 0
.circuit x0x1
ansatz(theta, phi)
H 0
H 1
MEASURE 0 [0]
MEASURE 1 [1]
''')
```

(continues on next page)

(continued from previous page)

```
x0x1 = xacc.getCompiled('x0x1')
```

4.2.7 Single-source Pythonic Programming

4.2.8 Benchmarks

Since XACC provides a hardware-agnostic framework for quantum-classical computing, it is well-suited for the development of general benchmark tasks that run on available backend quantum computers. XACC provides a Pythonic benchmarking tool that enables users to define benchmarks via an input file or python dictionary, and then distribute those files to be executed on available backends. Benchmarks can be low-level and hardware-specific, or high-level, application-style benchmarks.

The suite is extensible in the benchmark itself, as well as input data required for the benchmark.

All benchmarks can be defined as INI files. These files describe named sections of key-value pairs. Each benchmark requires an XACC section (for the definition of the backend accelerator, number of shots, etc.) and a Benchmark section (specifying the benchmark name and algorithm). Other sections are specified by the concrete benchmark sub-type.

4.2.8.1 Chemistry

This Benchmark implementation enables one to define an application-level benchmark which attempts to compute the accuracy with which a given quantum backend can compute the ground state energy of a specified electronic structure computation. Below is an example of such a benchmark input file

```
[XACC]
accelerator = ibm:ibmq_johannesburg
shots = 1024
verbose = True

[Decorators]
readout_error = True

[Benchmark]
name = chemistry
algorithm = vqe

[Ansatz]
source = .compiler xasm
    .circuit ansatz2
    .parameters x
    .qbit q
    X(q[0]);
    X(q[2]);
    ucc1(q, x[0]);

[Observable]
name = psi4
basis = sto-3g
geometry = 0 1
       Na  0.000000   0.0      0.0
       H   0.0        0.0  1.914388
```

(continues on next page)

(continued from previous page)

```
       symmetry c1
fo = [0, 1, 2, 3, 4, 10, 11, 12, 13, 14]
ao = [5, 9, 15, 19]

[Optimizer]
name = nlopt
nlopt-maxeval = 20
```

Stepping, through this, we see the benchmark is to be executed on the IBM Johannesburg backend, with 1024 shots. Next, we specify what benchmark algorithm to run - the Chemistry benchmark using the VQE algorithm. After that, this benchmark enables one to specify any AcceleratorDecorators to be used, here we turn on readout-error decoration to correct computed expectation values with respect to measurement readout errors. Moving down the file, one now specifies the specific state-preparation ansatz to be used for this VQE run, using the usual XACC qasm() format. Finally, we specify the Observable we are interested in studying, and the classical optimizer to be used in searching for the optimal expecation value for that observable.

One can run this benchmark with the following command (presuming it is in a file named chem_nah_vqe_ibm.ini)

```
$ python3 -m xacc --benchmark chem_nah_vqe_ibm.ini
```

4.2.8.2 Quantum Process Tomography

```
[XACC]
accelerator = ibm:ibmq_poughkeepsie
verbose = True

[Benchmark]
name = qpt
analysis = ['fidelity', 'heat-maps']
chi-theoretical-real = [0., 0., 0., 0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 1.]

[Circuit]
# Logical circuit source code
source = .compiler xasm
    .circuit hadamard
    .qbit q
    H(q[0]);

# Can specify physical qubit to run on
qubit-map = [1]
```

4.3 Extensions

Here we detail concrete implementations of various XACC interfaces as well as any input parameters they expose.

4.3.1 Compilers

4.3.1.1 xasm

The XASM Compiler is the default compiler in XACC. It is the closest language to the underlying XACC IR data model. The XASM compiler provides a quantum assembly like language with support for custom Instructions and Composite Instruction generators as part of the language. Instructions are provided to the language via the usual XACC service registry. Most common digital gates are provided by default, and it is straightforward to add new Instructions.

4.3.1.2 quilc

XACC provides a Compiler implementation that delegates to the Rigetti-developed quilc compiler. This is acheieved through the `rigetti/quilc` Docker image and the internal XACC REST client API. The Quilc Compiler implementation makes direct REST POSTs and GETs to the users local Docker Engine. Therefore, in order to use this Compiler, you must pull down this image.

```
$ docker pull rigetti/quilc
```

With that image pulled, you can now use the Quilc compiler via the usual XACC API calls.

```
auto compiler = xacc::getCompiler("quilc");
auto ir = compiler->compile(R"##(H 0
CNOT 0 1
)##");
std::cout << ir->getComposites()[0]->toString() << "\n";
```

or in Python

```
compiler = xacc.getCompiler('quilc')
ir = compiler.compile('''__qpu__ void ansatz(qbit q, double x) {
    X 0
    CNOT 1 0
    RY(x) 1
}''')
```

Note that you can structure your input to the compiler as a typical XACC kernel source string or as a raw Quil string.

4.3.2 Optimizers

XACC provides implementations for the `Optimizer` that delegate to NLOpt and MLPack. Here we demonstrate the various ways to configure these optimizers for a number of different solver types.

In addition to the enumerated parameters below, all `Optimizers` expose an `initial-parameters` key that must be a list or vector of doubles with size equal to the number of parameters. By default, `[0.,0.,...,0.,0.]` is used.

4.3.2.1 MLPack

mlpack-optimizer	Optimizer Parameter	Parameter Description	default	type
adam	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-beta1	Exponential decay rate for the first moment estimates.	.7	double
	mlpack-beta2	Exponential decay rate for the weighted infinity norm estimates.	.999	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-eps	Value used to initialize the mean squared gradient parameter.	1e-8	double
l-bfgs	None			
adagrad	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-eps	Value used to initialize the mean squared gradient parameter.	1e-8	double
adadelta	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-eps	Value used to initialize the mean squared gradient parameter.	1e-8	double
	mlpack-rho	Smoothing constant.	.95	double
cmaes	mlpack-cmaes-lambda	The population size.	0	int
	mlpack-cmaes-upper-bound	Upper bound of decision variables.	10.	duoble

continues on next page

Table 1 – continued from previous page

mlpack-optimizer	Optimizer Parameter	Parameter Description	default	type
	mlpack-cmaes-lower-bound	Lower bound of decision variables.	-10.0	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
gd	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
momentum-sgd	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-momentum	Maximum absolute tolerance to terminate algorithm.	.05	double
momentum-nesterov	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-momentum	Maximum absolute tolerance to terminate algorithm.	.05	double
sgd	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
rms-prop	mlpack-step-size	Step size for each iteration.	.5	double
	mlpack-max-iter	Maximum number of iterations allowed	500000	int
	mlpack-tolerance	Maximum absolute tolerance to terminate algorithm.	1e-4	double
	mlpack-alpha	Smoothing constant	.99	double

continues on next page

Table 1 – continued from previous page

mlpack-optimizer	Optimizer Parameter	Parameter Description	default	type
	mlpack-eps	Value used to initialize the mean squared gradient parameter.	1e-8	double

Various examples of using the mlpack optimizer:

```
// sgd with defaults
auto optimizer = xacc::getOptimizer("mlpack", {std::make_pair("mlpack-optimizer", "sgd")}
↪);
// default adam
optimizer = xacc::getOptimizer("mlpack")
// adagrad with 30 max iters and .01 step size
auto optimizer = xacc::getOptimizer("mlpack", {std::make_pair("mlpack-optimizer",
↪"adagrad"),
                                               std::make_pair("mlpack-step-size", .01),
                                               std::make_pair("mlpack-max-iter", 30)});
```

or in Python

```
optimizer = xacc.getOptimizer('mlpack', {'mlpack-optimizer':'sgd'})
// default adam
optimizer = xacc.getOptimizer("mlpack")
// adagrad with 30 max iters and .01 step size
optimizer = xacc.getOptimizer("mlpack", {'mlpack-optimizer':'adagrad',
                                         'mlpack-step-size':.01,
                                         'mlpack-max-iter':30})
```

4.3.2.2 NLOpt

nlopt-optimizer	Optimizer Parameter	Parameter Description	default	type
cobyla	nlopt-ftol	Maximum absolute tolerance to terminate algorithm.	1e-6	double
	nlopt-maxeval	Maximum number of iterations allowed	1000	int
l-bfgs	nlopt-ftol	Maximum absolute tolerance to terminate algorithm.	1e-6	double
	nlopt-maxeval	Maximum number of iterations allowed	1000	int
nelder-mead	nlopt-ftol	Maximum absolute tolerance to terminate algorithm.	1e-6	double
	nlopt-maxeval	Maximum number of iterations allowed	1000	int

4.3.3 Accelerators

Here we detail all available XACC `Accelerators` and their exposed input parameters.

4.3.3.1 IBM

The IBM Accelerator by default targets the remote `ibmq_qasm_simulator`. You can point to a different backend in two ways:

```
auto ibm_valencia = xacc::getAccelerator("ibm:ibmq_valencia");
... or ...
auto ibm_valencia = xacc::getAccelerator("ibm", {std::make_pair("backend", "ibmq_valencia
↪")});
```

in Python

```
ibm_valencia = xacc.getAccelerator('ibm:ibmq_valencia');
... or ...
ibm_valencia = xacc.getAccelerator('ibm', {'backend':'ibmq_valencia')});
```

You can specify the number of shots in this way as well

```
auto ibm_valencia = xacc::getAccelerator("ibm:ibmq_valencia", {std::make_pair("shots",␣
↪2048)});
```

or in Python

```
ibm_valencia = xacc.getAccelerator('ibm:ibmq_valencia', {'shots':2048)});
```

In order to target the remote backend (for `initialize()` or `execute()`) you must provide your IBM credentials to XACC. To do this add the following to a plain text file `$HOME/.ibm_config`

```
key: YOUR_KEY_HERE
hub: HUB
group: GROUP
project: PROJECT
```

You can also create this file using the `xacc` Python module

```
$ python3 -m xacc -c ibm -k YOUR_KEY --group GROUP --hub HUB --project PROJECT --url URL
[ for public API ]
$ python3 -m xacc -c ibm -k YOUR_KEY
```

where you provide URL, HUB, PROJECT, GROUP, and YOUR_KEY.

4.3.3.2 Aer

The Aer Accelerator provides a great example of contributing plugins or extensions to core C++ XACC interfaces from Python. To see how this is done, checkout the code here. This Accelerator connects the XACC IR infrastructure with the `qiskit-aer` simulator, providing a robust simulator that can mimic noise models published by IBM backends. Note to use these noise models you must have setup your `$HOME/.ibm_config` file (see above discussion on IBM Accelerator).

```
aer = xacc.getAccelerator('aer')
... or ...
aer = xacc.getAccelerator('aer', {'shots':8192})
... or ...
# For ibmq_johannesburg-like readout error
aer = xacc.getAccelerator('aer', {'shots':2048, 'backend':'ibmq_johannesburg', 'readout_
↪error':True})
... or ...
# For all ibmq_johannesburg-like errors
aer = xacc.getAccelerator('aer', {'shots':2048, 'backend':'ibmq_johannesburg',
                                  'readout_error':True,
                                  'thermal_relaxation':True,
                                  'gate_error':True})
```

You can also use this simulator from C++, just make sure you load the Python external language plugin.

```
xacc::Initialize();
xacc::external::load_external_language_plugins();
auto accelerator = xacc::getAccelerator("aer", {std::make_pair("shots", 8192),
                                                std::make_pair("readout_error", true)});
.. run simulation

xacc::external::unload_external_language_plugins();
xacc::Finalize();
```

4.3.3.3 QCS

XACC provides support for the Rigetti QCS platform through the QCS Accelerator implementation. This Accelerator requires a few extra third-party libraries that you will need to install in order to get QCS support. Specifically we need `libzmq`, `cppzmq`, `msgpack-c`, and `uuid-dev`. Note that more than likely this will only be built on the QCS Centos 7 VM, so the following instructions are specifically for that OS.

```
$ git clone https://github.com/zeromq/libzmq
$ cd libzmq/ && mkdir build && cd build
$ cmake .. -DCMAKE_INSTALL_PREFIX=~/.zmq
$ make -j12 install

$ cd ../..
$ git clone https://github.com/zeromq/cppzmq
$ cd cppzmq/ && mkdir build && cd build/
$ cmake .. -DCMAKE_INSTALL_PREFIX=~/.zmq -DCMAKE_PREFIX_PATH=~/.zmq
$ make -j12 install

$ cd ../..
$ git clone https://github.com/msgpack/msgpack-c/
```

(continues on next page)

(continued from previous page)

```
$ cd msgpack-c/ && mkdir build && cd build
$ cmake .. -DCMAKE_INSTALL_PREFIX=~/.zmq
$ make -j12 install
$ cd ../..

$ sudo yum install uuid-dev devtoolset-8-gcc devtoolset-8-gcc-c++
$ scl enable devtoolset-8 -- bash

[go to your xacc build directory]
cmake .. -DUUID_LIBRARY=/usr/lib64/libuuid.so.1
make install
```

There is no further configuration for using the QCS platform.

To use the QCS Accelerator targeting something like `Aspen-4-2Q-A` (for example, replace with your lattice):

```
auto qcs = xacc::getAccelerator("qcs:Aspen-4-2Q-A", {std::make_pair("shots", 10000)});
```

or in Python

```
qcs = xacc.getAccelerator('qcs:Aspen-4-2Q-A', {'shots':10000)});
```

For now you must manually map your `CompositeInstruction` to the correct physical bits provided by your lattice. To do so, run

```
qpu = xacc.getAccelerator('qcs:Aspen-4-2Q-A')
[given CompositeInstruction f]
f.defaultPlacement(qpu)
[or manually]
f.mapBits([5,9])
```

4.3.3.4 IonQ

The IonQ Accelerator by default targets the remote `simulator` backend. You can point to the physical QPU in two ways:

```
auto ionq = xacc::getAccelerator("ionq:qpu");
... or ...
auto ionq = xacc::getAccelerator("ionq", {std::make_pair("backend", "qpu")});
```

in Python

```
ionq = xacc.getAccelerator('ionq:qpu');
... or ...
ionq = xacc.getAccelerator('ionq', {'backend':'qpu')});
```

You can specify the number of shots in this way as well

```
auto ionq = xacc::getAccelerator("ionq", {std::make_pair("shots", 2048)});
```

or in Python

```
ionq = xacc.getAccelerator('ionq', {'shots':2048)});
```

In order to target the simulator or QPU (for `initialize()` or `execute()`) you must provide your IonQ credentials to XACC. To do this add the following to a plain text file `$HOME/.ionq_config`

```
key: YOUR_KEY_HERE
url: https://api.ionq.co/v0
```

4.3.3.5 DWave

The DWave Accelerator by default targets the remote `DW_2000Q_VFYC_2_1` backend. You can point to a different backend in two ways:

```
auto dw = xacc::getAccelerator("dwave:DW_2000Q");
... or ...
auto dw = xacc::getAccelerator("dwave", {std::make_pair("backend", "DW_2000Q")});
```

in Python

```
dw = xacc.getAccelerator('dwave:DW_2000Q');
... or ...
dw = xacc.getAccelerator('dwave', {'backend':'DW_2000Q')});
```

You can specify the number of shots in this way as well

```
auto dw = xacc::getAccelerator("dwave", {std::make_pair("shots", 2048)});
```

or in Python

```
dw = xacc.getAccelerator('dwave', {'shots':2048)});
```

In order to target the remote backend (for `initialize()` or `execute()`) you must provide your DWave credentials to XACC. To do this add the following to a plain text file `$HOME/.dwave_config`

```
key: YOUR_KEY_HERE
url: https://cloud.dwavesys.com
```

You can also create this file using the `xacc` Python module

```
$ python3 -m xacc -c dwave -k YOUR_KEY
```

where you provide YOUR_KEY.

4.3.3.6 DWave Neal

The DWave Neal Accelerator provides another example of contributing plugins or extensions to core C++ XACC interfaces from Python. To see how this is done, checkout the code here. This Accelerator connects the XACC IR infrastructure with the `dwave-neal` simulator, providing a local simulator that can mimic DWave QPU execution.

```
aer = xacc.getAccelerator('dwave-neal')
... or ...
aer = xacc.getAccelerator('dwave-neal', {'shots':2000})
```

You can also use this simulator from C++, just make sure you load the Python external language plugin.

```
xacc::Initialize();
xacc::external::load_external_language_plugins();
auto accelerator = xacc::getAccelerator("dwave-neal", {std::make_pair("shots", 8192)});
.. run simulation

xacc::external::unload_external_language_plugins();
xacc::Finalize();
```

4.3.3.7 TNQVM

TNQVM provides an Accelerator implementation that leverages tensor network theory to simulate quantum circuits.

TNQVM implements a few tensor-based quantum circuit simulators that can be specified by the *tnqvm-visitor* configuration key. The list of visitors and their descriptions are:

tnqvm-visitor	Description
itensor-mps	MPS simulator based on itensor library.
exatn	Full tensor contraction simulator based on ExaTN library. Tensor element type (*float* or *double*) can also specified after a ':' separator, e.g., *exatn:double* (default) or *exatn:float*
exatn-mps	MPS simulator based on ExaTN library.
exatn-pmps	Purified-MPS (density matrix) simulator based on ExaTN library.

```
auto qpu = xacc::getAccelerator("tnqvm", {{"tnqvm-visitor", "exatn"}});
```

For the *exatn* simulator, there are additional options that users can set during initialization:

Initialization Parameter	Parameter Description	type	default
exatn-buffer-size-gb	ExaTN's host memory buffer size (in GB)	int	8 (GB)
exatn-contract-seq-optimizer	ExaTN's contraction sequence optimizer to use.	string	metis
calc-contract-cost-flops	Estimate the Flops and Memory requirements only (no tensor contraction) If true, the following info will be added to the AcceleratorBuffer: • *contract-flops*: Flops count. • *max-node-bytes*: Max intermediate tensor size in memory. • *optimizer-elapsed-time-ms*: optimization walltime.	bool	false
bitstring	If provided, the output amplitude/partial state vector associated with that *bitstring* will be computed. The length of the input *bitstring* must match the number of qubits. Non-projected bits (partial state vector) are indicated by *-1* values. Returned values in the AcceleratorBuffer: • *amplitude-real/amplitude-real-vec*: Real part of the result. • *amplitude-imag/amplitude-imag-vec*: Imaginary part of the result.	vector<int>	<unused>
contract-with-conjugate	If true, we append the conjugate of the input circuit. This is used to validate internal tensor contraction. *contract-with-conjugate-result* key in the AcceleratorBuffer will be set to *true* if the validation is successful.	bool	false
mpi-communicator	The MPI communicator to initialize ExaTN runtime with. If not provided by default, ExaTN will use *MPI_COMM_WORLD*.	void*	<unused>
exp-val-by-conjugate	If true, the expectation	bool	false

For the *exatn-mps* simulator, there are additional options that users can set during initialization:

Initialization Parameter	Parameter Description	type	default
svd-cutoff	SVD cut-off limit.	double	numeric_limits::min
max-bond-dim	Max bond dimension to keep.	int	no limit
mpi-communicator	The MPI communicator to initialize ExaTN runtime with. If not provided, by default, ExaTN will use *MPI_COMM_WORLD*.	void*	<unused>

For the *exatn-pmps* simulator, there are additional options that users can set during initialization:

Initialization Parameter	Parameter Description	type	default
backend-json	Backend configuration JSON to estimate the noise model from.	string	None
backend	Name of the IBMQ backend to query the backend configuration.	string	None

If either *backend-json* or *backend* is provided, the *exatn-pmps* simulator will simulate the backend noise associated with each quantum gate.

4.3.3.8 Atos QLM

XACC provides an Accelerator interface to the Atos QLM simulator (plugin name: "atos-qlm") via the QLMaaS (QLM-as-a-service) API. Users need to have a remote account (QLMaaS) to use this functionality.

Prerequisites: `myqlm` and `qlmaas` Python packages on your local machine. These packages can be installed via pip.

Connection configuration set-up:

- Method 1 (**recommended**): create a file named *.qlm_config* at the *$HOME* directory with the following information.

Please note that the host address can be changed to access other QLM machines if you have access to them.

```
host: quantumbull.ornl.gov
username: <your QLM user name>
password: <your QLM password>
```

- Method 2: provide the "username" and "password" fields when retrieving the qlm Accelerator, e.g.,

```
auto qlm = xacc::getAccelerator("atos-qlm", {{"username", "<your QLM user name>"}, {
↪"password", "<your QLM password>"}});
```

- Method 3: reusing your QLMaaS configuration if you have one.

For example, you have created the configuration using the Python *qlmaas* package, which is saved to *$HOME/.qlmaas/config.ini*. In this case, you don't need to provide any additional information. It should just work.

Here we list all configurations that this Accelerator supports.

Parameter	Parameter Description	type
sim-type	Name of the simulator. Options: LinAlg (default), MPS, Feynman, Bdd	std::string
shots	Number of measurement shots.	int
mps-threshold	[MPS] SVD truncation threshold.	double
max-bond	[MPS] Max bond dimension.	int
threads	Number of threads to use.	int

```
qpu = xacc.getAccelerator('atos-qlm', {'sim-type': 'MPS', 'max-bond': 512})
```

4.3.3.9 QuaC

The QuaC accelerator is a pulse-level accelerator (simulation only) that can execute quantum circuits at both gate and pulse (analog) level.

To use this accelerator, you need to build and install QuaC (see here for instructions.)

In pulse mode, you need to provide the QuaC accelerator a dynamical system model which can be constructed from an OpenPulse-format Hamiltonian JSON:

```
hamiltonianJson = {
   "description": "Hamiltonian of a one-qubit system.\n",
   "h_str": ["-0.5*omega0*Z0", "omegaa*X0||D0"],
   "osc": {},
   "qub": {
      "0": 2
   },
   "vars": {
      "omega0": 6.2831853,
      "omegaa": 0.0314159
   }
}
# Create a pulse system model object
model = xacc.createPulseModel()
# Load the Hamiltonian JSON (string) to the system model
loadResult = model.loadHamiltonianJson(json.dumps(hamiltonianJson))
```

The QuaC simulator can then be requested by

```
qpu = xacc.getAccelerator('QuaC', {'system-model': model.name()})
```

Pulse-level instructions can be constructed manually (assigning sample points)

```
pulseData = np.ones(pulseLength)
# Register the pulse named 'square' as an XACC instruction
xacc.addPulse('square', pulseData)
provider = xacc.getIRProvider('quantum')
squarePulseInst = provider.createInstruction('square', [0])
squarePulseInst.setChannel('d0')
# This instruction can be added to any XACC quantum Composite Instruction
prog.addInstruction(squarePulseInst)
```

or automatically (converting from quantum gates to pulses). To use automatic gate-to-pulse functionality, we need to load a pulse library to the accelerator as follows:

```
# Load the backend JSON file which contains a pulse library
backendJson = open('backends.json', 'r').read()
qpu.contributeInstructions(backendJson)
```

For more information, please check out these examples.

4.3.3.10 Qrack

The vm6502q/qrack simulator-based accelerator provides optional OpenCL-based GPU acceleration, as well as a novel simulator optimization layer.

```
auto qrk = xacc::getAccelerator("qrack", {std::make_pair("shots", 2048)});
```

By default, it selects initialization parameters that are commonly best for a wide range of use cases. However, it is highly configurable through a number of exposed parameters:

Initialization Parameter	Parameter Description	type	default
shots	Number of iterations to repeat the circuit for	int	-1 (Z-expectation only)
use_opencl	Use OpenCL acceleration if available, (otherwise native C++11)	bool	true
use_qunit	Turn on the novel optimization layer, (otherwise "Schrödinger method")	bool	true
device_id	The (Qrack) device ID number of the OpenCL accelerator to use	int	-1 (auto-select)
do_normalize	Enable small norm probability amplitude flooring and normalization	bool	true
zero_threshold	Norm threshold for clamping probability amplitudes to 0	double	1e-14/1e-30 float/double

4.3.4 Algorithms

XACC exposes hybrid quantum-classical Algorithm implementations for the variational quantum eigensolver (VQE), data-driven circuit learning (DDCL), and chemistry reduced density matrix generation (RDM).

4.3.4.1 VQE

The VQE Algorithm requires the following input information:

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator, vqe computes ground eigenvalue of this	std::shared_ptr<Observable>
ansatz	The unmeasured, parameterized quantum circuit	std::shared_ptr<CompositeInstruction>
optimizer	The classical optimizer to use	std::shared_ptr<Optimizer>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>

This Algorithm will add `opt-val` (`double`) and `opt-params` (`std::vector<double>`) to the provided `AcceleratorBuffer`. The results of the algorithm are therefore retrieved via these keys (see snippet below). Note you can control the initial VQE parameters with the `Optimizer` `initial-parameters` key (by default all zeros).

```
#include "xacc.hpp"
#include "xacc_observable.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);
```

(continues on next page)

(continued from previous page)

```
  // Get reference to the Accelerator
  // specified by --accelerator argument
  auto accelerator = xacc::getAccelerator();

  // Create the N=2 deuteron Hamiltonian
  auto H_N_2 = xacc::quantum::getObservable(
      "pauli", std::string("5.907 - 2.1433 X0X1 "
                           "- 2.1433 Y0Y1"
                           "+ .21829 Z0 - 6.125 Z1"));

  auto optimizer = xacc::getOptimizer("nlopt",
                          {std::make_pair("initial-parameters", {.5})});

  // JIT map Quil QASM Ansatz to IR
  xacc::qasm(R"(
.compiler xasm
.circuit deuteron_ansatz
.parameters theta
.qbit q
X(q[0]);
Ry(q[1], theta);
CNOT(q[1],q[0]);
)");
  auto ansatz = xacc::getCompiled("deuteron_ansatz");

  // Get the VQE Algorithm and initialize it
  auto vqe = xacc::getAlgorithm("vqe");
  vqe->initialize({std::make_pair("ansatz", ansatz),
                   std::make_pair("observable", H_N_2),
                   std::make_pair("accelerator", accelerator),
                   std::make_pair("optimizer", optimizer)});

  // Allocate some qubits and execute
  auto buffer = xacc::qalloc(2);
  vqe->execute(buffer);

  auto ground_energy = (*buffer)["opt-val"].as<double>();
  auto params = (*buffer)["opt-params"].as<std::vector<double>>();
}
```

In Python:

```
import xacc

# Get access to the desired QPU and
# allocate some qubits to run on
qpu = xacc.getAccelerator('tnqvm')
buffer = xacc.qalloc(2)

# Construct the Hamiltonian as an XACC-VQE PauliOperator
ham = xacc.getObservable('pauli', '5.907 - 2.1433 X0X1 - 2.1433 Y0Y1 + .21829 Z0 - 6.125␣
↪Z1')
```

(continues on next page)

(continued from previous page)

```
xacc.qasm('''.compiler xasm
.circuit ansatz2
.parameters t0
.qbit q
X(q[0]);
Ry(q[1],t0);
CX(q[1],q[0]);
''')
ansatz2 = xacc.getCompiled('ansatz2')

opt = xacc.getOptimizer('nlopt', {'initial-parameters':[.5]})

# Create the VQE algorithm
vqe = xacc.getAlgorithm('vqe', {
                        'ansatz': ansatz2,
                        'accelerator': qpu,
                        'observable': ham,
                        'optimizer': opt
                        })
vqe.execute(buffer)
energy = buffer['opt-val']
params = buffer['opt-params']
```

4.3.4.2 DDCL

The DDCL Algorithm implements the following algorithm - given a target probability distribution, propose a parameterized quantum circuit and train (minimize loss) the circuit to reproduce that given target distribution. We design DDCL to be extensible in loss function computation and gradient computation strategies.

The DDCL Algorithm requires the following input information:

Algorithm Parameter	Parameter Description	type
target_dist	The target probability distribution to reproduce	std::vector<double>
ansatz	The unmeasured, parameterized quantum circuit	std::shared_ptr<CompositeInstruction>
optimizer	The classical optimizer to use, can be gradient based	std::shared_ptr<Optimizer>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
loss	The loss strategy to use	std::string
gradient	The gradient strategy to use	std::string

As of this writing, loss can take `js` and `mmd` values for Jansen-Shannon divergence and Maximum Mean Discrepancy, respectively. More are being added. Also, gradient can take `js-parameter-shift` and `mmd-parameter-shift` values. These gradient strategies will shift each parameter by plus or minus pi over 2.

```
#include "xacc.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);
```

(continues on next page)

(continued from previous page)

```
  xacc::external::load_external_language_plugins();
  xacc::set_verbose(true);

  // Get reference to the Accelerator
  auto accelerator = xacc::getAccelerator("aer");

  auto optimizer = xacc::getOptimizer("mlpack");
  xacc::qasm(R"(
.compiler xasm
.circuit qubit2_depth1
.parameters x
.qbit q
U(q[0], x[0], -pi/2, pi/2 );
U(q[0], 0, 0, x[1]);
U(q[1], x[2], -pi/2, pi/2);
U(q[1], 0, 0, x[3]);
CNOT(q[0], q[1]);
U(q[0], 0, 0, x[4]);
U(q[0], x[5], -pi/2, pi/2);
U(q[1], 0, 0, x[6]);
U(q[1], x[7], -pi/2, pi/2);
)");
  auto ansatz = xacc::getCompiled("qubit2_depth1");

  std::vector<double> target_distribution {.5, .5, .5, .5};

  auto ddcl = xacc::getAlgorithm("ddcl");
  ddcl->initialize({std::make_pair("ansatz", ansatz),
                std::make_pair("target_dist", target_distribution),
                std::make_pair("accelerator", accelerator),
                std::make_pair("loss", "js"),
                std::make_pair("gradient", "js-parameter-shift"),
                std::make_pair("optimizer", optimizer)});

  // Allocate some qubits and execute
  auto buffer = xacc::qalloc(2);
  ddcl->execute(buffer);

  // Print the result
  std::cout << "Loss: " << buffer["opt-val"].as<double>()
         << "\n";

  xacc::external::unload_external_language_plugins();
  xacc::Finalize();
}
```

or in Python

```
import xacc
# Get the QPU and allocate a single qubit
qpu = xacc.getAccelerator('aer')
```

(continues on next page)

(continued from previous page)

```
qbits = xacc.qalloc(1)

# Get the MLPack Optimizer, default is Adam
optimizer = xacc.getOptimizer('mlpack')

# Create a simple quantum program
xacc.qasm('''
.compiler xasm
.circuit foo
.parameters x,y,z
.qbit q
Ry(q[0], x);
Ry(q[0], y);
Ry(q[0], z);
''')
f = xacc.getCompiled('foo')

# Get the DDCL Algorithm, initialize it
# with necessary parameters
ddcl = xacc.getAlgorithm('ddcl', {'ansatz': f,
                                  'accelerator': qpu,
                                  'target_dist': [.5,.5],
                                  'optimizer': optimizer,
                                  'loss': 'js',
                                  'gradient': 'js-parameter-shift'})
# execute
ddcl.execute(qbits)

print(qbits.keys())
print(qbits['opt-val'])
print(qbits['opt-params'])
```

4.3.4.3 Rotoselect

The `Rotoselect` Quantum Circuit Structure Learning Algorithm (Ostaszewski et al. (2019)) requires the following input information:

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator, Rotoselect computes ground eigenvalue of this	std::shared_ptr<Observable>/Observable*
layers	Number of circuit layers. Each layer consists of parametrized single-qubit rotations followed by a ladder of controlled-Z gates.	int
iterations	The number of training iterations	int
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>/Accelerator*

This `Rotoselect` algorithm is designed to learn a good circuit structure (generators of rotation are selected from the set of Pauli gates) at fixed depth (`layers`) to minimize the cost function (`observable`). This Algorithm will add `opt-val` (`double`) to the provided `AcceleratorBuffer`. The result of the algorithm is therefore retrieved via this key (see snippet below).

4.3.4.4 RBM Classification

The `rbm_classification` algorithm provides an implementation that trains a restricted boltzmann machine via sampling of a quantum annealer for the purpose of classification. (Caldeira et al. (2019)) It exposes the following input information:

Algorithm Parameter	Parameter Description	type
nv	The number of visible units	int
nh	The number of hidden units	int
batch-size	The batch size, defaults to 1	int
n-gibbs-steps	The number of gibbs steps to use in post-processing of dwave data	int
train-steps	Hard-code the number of training iterations/steps, by default this is set to -1, meaning unlimited iterations	int
epochs	The number of training epochs, defaults to 1	int
train-file	The location (relative to pwd) of the training data (as npy file)	string
expectation-strategy	Strategy to use in computing model expectation values, can be gibbs, quantum-annealing, discriminative, or cd	string
backend	The desired quantum-annealing backend (defaults to dwave-neal), can be any of the available D-Wave backends, must be provided as dwave:BEND	string
shots	The number of samples to draw from the dwave backend	int
embedding	The minor graph embedding to use, if not provided, one will be computed and used for subsequent calls to the dwave backend.	map<int, vector<int>>

Example usage in Python:

```
import xacc

# Create the RBM Classification algorithm
algo = xacc.getAlgorithm('rbm-classification',
            {
            'nv':64,
            'nh':64,
            'train-file':'sg_train_64bits.npy',
            'expectation-strategy':'quantum-annealing',
            'backend':'dwave:DW_2000Q_5',
            'shots':100,
            })

qbits = xacc.qalloc()
algo.execute(qbits)

# get the trained RBM weights
# for further use and post-processing
w = qbits['w']
bv = qbits['bv']
bh = qbits['bh']
```

4.3.4.5 Quantum Process Tomography

The qpt algorithm provides an implementation of Algorithm that uses linear inversion to compute the chi process matrix for a desired circuit.

Algorithm Parameter	Parameter Description	type
circuit	The circuit to characterize	pointer-like CompositeInstruction
accelerator	The backend quantum computer to use	pointer-like Accelerator
qubit-map	The physical qubits to map the logical circuit onto	vector<int>

```
#include "xacc.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);
  auto acc = xacc::getAccelerator("ibm::ibmq_poughkeepsie");

  auto compiler = xacc::getCompiler("xasm");
  auto ir = compiler->compile(R"(__qpu__ void f(qbit q) {
      H(q[0]);
  })", nullptr);
  auto h = ir->getComposite("f");

  auto qpt = xacc::getAlgorithm("qpt", {
                          std::make_pair("circuit", h),
                          std::make_pair("accelerator", acc)
                          });

  auto buffer = xacc::qalloc(1);
  qpt->execute(buffer);

  auto chi_real = (*buffer)["chi-real"];
  auto chi_imag = (*buffer)["chi-imag"];

}
```

or in Python

```
import xacc
# Choose the QPU on which to
# characterize the process matrix for a Hadamard
qpu = xacc.getAccelerator('ibm:ibmq_poughkeepsie')

# Create the CompositeInstruction containing a
# single Hadamard instruction
provider = xacc.getIRProvider('quantum')
circuit = provider.createComposite('U')
hadamard = provider.createInstruction('H', [0])
circuit.addInstruction(hadamard)

# Create the Algorithm, give it the circuit
# to characterize and the backend to target
qpt = xacc.getAlgorithm('qpt', {'circuit':circuit, 'accelerator':qpu})
```

(continues on next page)

(continued from previous page)

```
# Allocate a qubit, this will
# store our tomography results
buffer = xacc.qalloc(1)

# Execute
qpt.execute(buffer)

# Compute the fidelity with respect to
# the ideal hadamard process
F = qpt.calculate('fidelity', buffer, {'chi-theoretical-real':[0., 0., 0., 0., 0., 1., 0.
↪, 1., 0., 0., 0., 0., 1., 0., 1.]})
print('\nFidelity: ', F)
```

4.3.4.6 QAOA

The QAOA Algorithm requires the following input information:

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator represents the cost Hamiltonian.	std::shared_ptr<Observable>
optimizer	The classical optimizer to use	std::shared_ptr<Optimizer>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
steps	The number of timesteps. Corresponds to 'p' in the literature. This is optional, default = 1 if not provided.	int
parameter-scheme	The QAOA parameterization scheme ('Extended' or 'Standard'). This is optional, default = 'Extended' if not provided.	string
graph	The MaxCut graph problem. If provided, the cost Hamiltonian is constructed automatically.	std::shared_ptr<Graph>

This Algorithm will add `opt-val` (`double`) and `opt-params` (`std::vector<double>`) to the provided `AcceleratorBuffer`. The results of the algorithm are therefore retrieved via these keys (see snippet below). Note you can control the initial QAOA parameters with the `Optimizer` `initial-parameters` key (by default all zeros).

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"
#include <random>

// Use XACC built-in QAOA to solve a QUBO problem
// QUBO function:
// y = -5x1 - 3x2 - 8x3 - 6x4 + 4x1x2 + 8x1x3 + 2x2x3 + 10x3x4
int main(int argc, char **argv) {
   xacc::Initialize(argc, argv);
   // Use the Qpp simulator as the accelerator
   auto acc = xacc::getAccelerator("qpp");

   auto buffer = xacc::qalloc(4);
   // The corresponding QUBO Hamiltonian is:
   auto observable = xacc::quantum::getObservable(
```

(continues on next page)

(continued from previous page)

```
        "pauli",
        std::string("-5.0 - 0.5 Z0 - 1.0 Z2 + 0.5 Z3 + 1.0 Z0 Z1 + 2.0 Z0 Z2 + 0.5 Z1␣
↪Z2 + 2.5 Z2 Z3"));

   const int nbSteps = 12;
   const int nbParams = nbSteps*11;
   std::vector<double> initialParams;
   std::random_device rd;
   std::mt19937 gen(rd());
   std::uniform_real_distribution<> dis(-2.0, 2.0);

   // Init random parameters
   for (int i = 0; i < nbParams; ++i)
   {
      initialParams.emplace_back(dis(gen));
   }

   auto optimizer = xacc::getOptimizer("nlopt",
      xacc::HeterogeneousMap {
         std::make_pair("initial-parameters", initialParams),
         std::make_pair("nlopt-maxeval", nbParams*100) });

   auto qaoa = xacc::getService<xacc::Algorithm>("QAOA");

   const bool initOk = qaoa->initialize({
                           std::make_pair("accelerator", acc),
                           std::make_pair("optimizer", optimizer),
                           std::make_pair("observable", observable),
                           // number of time steps (p) param
                           std::make_pair("steps", nbSteps)
                        });
   qaoa->execute(buffer);
   std::cout << "Min QUBO: " << (*buffer)["opt-val"].as<double>() << "\n";
}
```

In Python:

```
import xacc,sys, numpy as np

# Get access to the desired QPU and
# allocate some qubits to run on
qpu = xacc.getAccelerator('qpp')

# Construct the Hamiltonian as an XACC PauliOperator
# This Hamiltonian corresponds to the QUBO problem:
# y = -5x_1 -3x_2 -8x_3 -6x_4 + 4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4
ham = xacc.getObservable('pauli', '-5.0 - 0.5 Z0 - 1.0 Z2 + 0.5 Z3 + 1.0 Z0 Z1 + 2.0 Z0␣
↪Z2 + 0.5 Z1 Z2 + 2.5 Z2 Z3')

# We need 4 qubits
buffer = xacc.qalloc(4)
```

(continues on next page)

(continued from previous page)

```
# There are 7 gamma terms (non-identity) in the cost Hamiltonian
# and 4 beta terms for mixer Hamiltonian
nbParamsPerStep = 7 + 4

# The number of steps (often referred to as 'p' parameter):
# alternating layers of mixer and cost Hamiltonian exponential.
nbSteps = 4

# Total number of params
nbTotalParams = nbParamsPerStep * nbSteps

# Init params randomly:
initParams = np.random.rand(nbTotalParams)

# The optimizer: nlopt
opt = xacc.getOptimizer('nlopt', { 'initial-parameters': initParams })

# Create the QAOA algorithm
qaoa = xacc.getAlgorithm('QAOA', {
                         'accelerator': qpu,
                         'observable': ham,
                         'optimizer': opt,
                         'steps': nbSteps
                         })

result = qaoa.execute(buffer)
print('Min QUBO value = ', buffer.getInformation('opt-val'))
```

4.3.4.7 Quantum Phase Estimation

The QPE algorithm (also known as quantum eigenvalue estimation algorithm) provides an implementation of Algorithm that estimates the phase (or eigenvalue) of an eigenvector of a unitary operator.

Here the unitary operator is called an *oracle* which is a quantum subroutine that acts upon a set of qubits and returns the answer as a phase. The bits precision is automatically inferred from the size of the input buffer.

Algorithm Parameter	Parameter Description	type
oracle	The circuit represents the unitary operator.	pointer-like CompositeInstruction
accelerator	The backend quantum computer to use.	pointer-like Accelerator
state-preparation	The circuit to prepare the eigen state.	pointer-like CompositeInstruction

```
#include "xacc.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
xacc::Initialize(argc, argv);
// Accelerator:
auto acc = xacc::getAccelerator("qpp", {std::make_pair("shots", 4096)});

// In this example: we want to estimate the *phase* of an arbitrary 'oracle'
```

(continues on next page)

(continued from previous page)

```
// i.e. Oracle(|State>) = exp(i*Phase)*|State>
// and we need to estimate that Phase.

// Oracle: CPhase(theta) or CU1(theta) which is defined as
// 1 0 0 0
// 0 1 0 0
// 0 0 1 0
// 0 0 0 e^(i*theta)
// The eigenstate is |11>; i.e. CPhase(theta)|11> = e^(i*theta)|11>

// Since this oracle operates on 2 qubits, we need to add more qubits to the buffer.
// The more qubits we have, the more accurate the estimate.
// Resolution := 2^(number qubits in the calculation register).
// 5-bit precision => 7 qubits in total
auto buffer = xacc::qalloc(7);
auto qpe = xacc::getService<xacc::Algorithm>("QPE");
auto compiler = xacc::getCompiler("xasm");

// Create oracle: CPhase gate with theta = 2pi/3
// i.e. the phase value to estimate is 1/3 ~ 0.33333.
auto gateRegistry = xacc::getService<xacc::IRProvider>("quantum");
auto oracle = gateRegistry->createComposite("oracle");
oracle->addInstruction(gateRegistry->createInstruction("CPhase", { 0, 1 }, { 2.0 * M_PI/␣
↪3.0 }));

// Eigenstate preparation = |11> state
auto statePrep = compiler->compile(R"(__qpu__ void prep1(qbit q) {
   X(q[0]);
   X(q[1]);
})", nullptr)->getComposite("prep1");

// Initialize the Quantum Phase Estimation:
qpe->initialize({
                  std::make_pair("accelerator", acc),
                  std::make_pair("oracle", oracle),
                  std::make_pair("state-preparation", statePrep)
                  });

// Run the algorithm
qpe->execute(buffer);
// Expected result:
// The factor here is 2^5 (precision) = 32
// we expect the two most-likely bitstring is 10 and 11
// i.e. the true result is between 10/32 = 0.3125 and 11/32 = 0.34375
std::cout << "Probability of the two most-likely bitstrings 10 (theta = 0.3125) and 11␣
↪(theta = 0.34375 ): \n";
std::cout << "Probability of |11010> (11) = " << buffer->computeMeasurementProbability(
↪"11010") << "\n";
std::cout << "Probability of |01010> (10) = " << buffer->computeMeasurementProbability(
↪"01010") << "\n";

xacc::Finalize();
```

(continues on next page)

(continued from previous page)

```
}
```

or in Python

```
import xacc,sys, numpy as np

# Get access to the desired QPU and
# allocate some qubits to run on
qpu = xacc.getAccelerator('qpp',  { 'shots': 4096 })

# In this example: we want to estimate the *phase* of an arbitrary 'oracle'
# i.e. Oracle(|State>) = exp(i*Phase)*|State>
# and we need to estimate that Phase.

# The oracle is a simple T gate, and the eigenstate is |1>
# T|1> = e^(i*pi/4)|1>
# The phase value of pi/4 = 2pi * (1/8)
# i.e. if we use a 3-bit register for estimation,
# we will get the correct answer of 1 deterministically.

xacc.qasm('''.compiler xasm
.circuit oracle
.qbit q
T(q[0]);
''')
oracle = xacc.getCompiled('oracle')

# We need to prepare the eigenstate |1>
xacc.qasm('''.compiler xasm
.circuit prep
.qbit q
X(q[0]);
''')
statePrep = xacc.getCompiled('prep')

# We need 4 qubits (3-bit precision)
buffer = xacc.qalloc(4)

# Create the QPE algorithm
qpe = xacc.getAlgorithm('QPE', {
                        'accelerator': qpu,
                        'oracle': oracle,
                        'state-preparation': statePrep
                        })

qpe.execute(buffer)
# We should only get the bit string of |100> = 1
# i.e. phase value of 1/2^3 = 1/8.
print(buffer)
```

4.3.4.8 QITE

The Quantum Imaginary Time Evolution (QITE) Algorithm requires the following input information: (Motta et al. (2020))

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator represents the cost Hamiltonian.	std::shared_ptr<Observable>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
steps	The number of Trotter steps.	int
step-size	The Trotter step size.	double

Optionally, users can provide these parameters:

Algorithm Parameter	Parameter Description	type
ansatz	State preparation circuit.	std::shared_ptr<CompositeInstruction>
analytical	If true, perform an analytical run rather than executing quantum circuits on the Accelerator backend.	boolean
initial-state	For *analytical* mode only, select the initial state.	int

This Algorithm will add `opt-val` (`double`) which is the energy value at the final Trotter step to the provided `AcceleratorBuffer`. The results of the algorithm are therefore retrieved via these keys (see snippet below). Also, energy values at each Trotter step are stored in the `exp-vals` field (`vector<double>`).

Note: during execution, the following line may be logged to the output console:

```
warning: solve(): system seems singular; attempting approx solution
```

This is completely normal and can be safely ignored.

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
   xacc::Initialize(argc, argv);
   // Use the Qpp simulator as the accelerator
   auto acc = xacc::getAccelerator("qpp");

   auto buffer = xacc::qalloc(1);
   auto observable = xacc::quantum::getObservable(
         "pauli",
         std::string("0.7071067811865475 X0 + 0.7071067811865475 Z0"));

   auto qite = xacc::getService<xacc::Algorithm>("qite");
   const int nbSteps = 25;
   const double stepSize = 0.1;

   const bool initOk =  qite->initialize({
      std::make_pair("accelerator", acc),
      std::make_pair("steps", nbSteps),
      std::make_pair("observable", observable),
      std::make_pair("step-size", stepSize)
```

(continues on next page)

(continued from previous page)

```
   });

   qite->execute(buffer);
   std::cout << "Min Energy: " << (*buffer)["opt-val"].as<double>() << "\n";
}
```

In Python:

```
import xacc,sys, numpy as np
import matplotlib.pyplot as plt

# Get access to the desired QPU and
# allocate some qubits to run on
qpu = xacc.getAccelerator('qpp')

# Construct the Hamiltonian as an XACC PauliOperator
ham = xacc.getObservable('pauli', '0.70710678118 X0 + 0.70710678118 Z0')

# We just need 1 qubit
buffer = xacc.qalloc(1)

# Horizontal axis: 0 -> 2.5
# The number of Trotter steps
nbSteps = 25

# The Trotter step size
stepSize = 0.1

# Create the QITE algorithm
qite = xacc.getAlgorithm('qite', {
                         'accelerator': qpu,
                         'observable': ham,
                         'step-size': stepSize,
                         'steps': nbSteps
                         })

result = qite.execute(buffer)

# Expected result: ~ -1
print('Min energy value = ', buffer.getInformation('opt-val'))
E = buffer.getInformation('exp-vals')
# Plot energy vs. beta
plt.plot(np.arange(0, nbSteps + 1) * stepSize, E, 'ro', label = 'XACC QITE')
plt.grid()
plt.show()
```

4.3.4.9 ADAPT

The Adaptive Derivative Assembled Problem Tailored (ADAPT) Algorithm requires the following input information: (Grismley et al. (2018), Tang et al. (2019), Zhu et al. (2020))

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator represents the Hamiltonian	std::shared_ptr<Observable>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
optimizer	The classical optimizer to use	std::shared_ptr<Optimizer>
pool	Pool of operators to construct adaptive ansatz	std::string
sub-algorithm	Algorithm called by ADAPT (VQE or QAOA)	std::string

Optionally, users can provide these parameters:

Algorithm Parameter	Parameter Description	type
initial-state	State preparation circuit.	std::shared_ptr<CompositeInstruction>
n-electrons	Required parameter for VQE, unless initial-state is provided	int
maxiter	Maximum number of ADAPT cycles/number of layers in QAOA	int
print-threshold	Value above which commutators are printed (Default 1.0e-10)	double
adapt-threshold	Stops ADAPT when norm of gradient vector falls below this value (Default 1.0e-2)	double

ADAPT-VQE

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  // Get reference to the Accelerator
  // specified by --accelerator argument
  auto accelerator = xacc::getAccelerator("qpp");

  // Get reference to the Optimizer
  // specified by --optimizer argument
  auto optimizer = xacc::getOptimizer("nlopt", {std::make_pair("nlopt-optimizer", "l-bfgs
↪")});

  // Allocate 4 qubits in the buffer
  auto buffer = xacc::qalloc(4);

  // Instantiate ADAPT algorithm
  auto adapt = xacc::getService<xacc::Algorithm>("adapt");

  // Number of electrons
  int nElectrons = 2;

  // Specify the operator pool
```

(continues on next page)

(continued from previous page)

```
  auto pool = "qubit-pool";

  // Specify the sub algorithm
  auto subAlgo_vqe = "vqe";

  // This is the H2 Hamiltonian in a fermionic basis
  auto str = std::string("(-0.165606823582,-0)  1^ 2^ 1 2 + (0.120200490713,0)  1^ 0^ 0␣
↪1 + "
                         "(-0.0454063328691,-0)  0^ 3^ 1 2 + (0.168335986252,0)  2^ 0^␣
↪0 2 + "
                         "(0.0454063328691,0)  1^ 2^ 3 0 + (0.168335986252,0)  0^ 2^ 2␣
↪0 + "
                         "(0.165606823582,0)  0^ 3^ 3 0 + (-0.0454063328691,-0)  3^ 0^␣
↪2 1 + "
                         "(-0.0454063328691,-0)  1^ 3^ 0 2 + (-0.0454063328691,-0)  3^␣
↪1^ 2 0 + "
                         "(0.165606823582,0)  1^ 2^ 2 1 + (-0.165606823582,-0)  0^ 3^ 0␣
↪3 + "
                         "(-0.479677813134,-0)  3^ 3 + (-0.0454063328691,-0)  1^ 2^ 0 3␣
↪+ "
                         "(-0.174072892497,-0)  1^ 3^ 1 3 + (-0.0454063328691,-0)  0^ 2^
↪ 1 3 + "
                         "(0.120200490713,0)  0^ 1^ 1 0 + (0.0454063328691,0)  0^ 2^ 3␣
↪1 + "
                         "(0.174072892497,0)  1^ 3^ 3 1 + (0.165606823582,0)  2^ 1^ 1 2␣
↪+ "
                         "(-0.0454063328691,-0)  2^ 1^ 3 0 + (-0.120200490713,-0)  2^ 3^
↪ 2 3 + "
                         "(0.120200490713,0)  2^ 3^ 3 2 + (-0.168335986252,-0)  0^ 2^ 0␣
↪2 + "
                         "(0.120200490713,0)  3^ 2^ 2 3 + (-0.120200490713,-0)  3^ 2^ 3␣
↪2 + "
                         "(0.0454063328691,0)  1^ 3^ 2 0 + (-1.2488468038,-0)  0^ 0 + "
                         "(0.0454063328691,0)  3^ 1^ 0 2 + (-0.168335986252,-0)  2^ 0^␣
↪2 0 + "
                         "(0.165606823582,0)  3^ 0^ 0 3 + (-0.0454063328691,-0)  2^ 0^␣
↪3 1 + "
                         "(0.0454063328691,0)  2^ 0^ 1 3 + (-1.2488468038,-0)  2^ 2 + "
                         "(0.0454063328691,0)  2^ 1^ 0 3 + (0.174072892497,0)  3^ 1^ 1␣
↪3 + "
                         "(-0.479677813134,-0)  1^ 1 + (-0.174072892497,-0)  3^ 1^ 3 1␣
↪+ "
                         "(0.0454063328691,0)  3^ 0^ 1 2 + (-0.165606823582,-0)  3^ 0^␣
↪3 0 + "
                         "(0.0454063328691,0)  0^ 3^ 2 1 + (-0.165606823582,-0)  2^ 1^␣
↪2 1 + "
                         "(-0.120200490713,-0)  0^ 1^ 0 1 + (-0.120200490713,-0)  1^ 0^␣
↪1 0 + (0.7080240981,0)");


  // Create Observable from Hamiltonian string
  auto H = xacc::quantum::getObservable("fermion", str);
```

(continues on next page)

(continued from previous page)

```
  // Pass parameters to ADAPT algorithm
  adapt->initialize({std::make_pair("accelerator", accelerator),
                                std::make_pair("observable", H),
                                std::make_pair("optimizer", optimizer),
                                std::make_pair("pool", pool),
                                std::make_pair("n-electrons", nElectrons),
                                std::make_pair("sub-algorithm", subAlgo_vqe)
                                });

  // Execute ADAPT-VQE
  adapt->execute(buffer);

  xacc::Finalize();
  return 0;
}
```

In Python:

```
import xacc

qpu = xacc.getAccelerator('qpp')
optimizer = xacc.getOptimizer('nlopt',{'nlopt-optimizer':'l-bfgs'})
buffer = xacc.qalloc(4)

opstr = '''
(-0.165606823582,-0)  1^ 2^ 1 2 + (0.120200490713,0)  1^ 0^ 0 1 +
(-0.0454063328691,-0)  0^ 3^ 1 2 + (0.168335986252,0)  2^ 0^ 0 2 +
(0.0454063328691,0)  1^ 2^ 3 0 + (0.168335986252,0)  0^ 2^ 2 0 +
(0.165606823582,0)  0^ 3^ 3 0 + (-0.0454063328691,-0)  3^ 0^ 2 1 +
(-0.0454063328691,-0)  1^ 3^ 0 2 + (-0.0454063328691,-0)  3^ 1^ 2 0 +
(0.165606823582,0)  1^ 2^ 2 1 + (-0.165606823582,-0)  0^ 3^ 0 3 +
(-0.479677813134,-0)  3^ 3 + (-0.0454063328691,-0)  1^ 2^ 0 3 +
(-0.174072892497,-0)  1^ 3^ 1 3 + (-0.0454063328691,-0)  0^ 2^ 1 3 +
(0.120200490713,0)  0^ 1^ 1 0 + (0.0454063328691,0)  0^ 2^ 3 1 +
(0.174072892497,0)  1^ 3^ 3 1 + (0.165606823582,0)  2^ 1^ 1 2 +
(-0.0454063328691,-0)  2^ 1^ 3 0 + (-0.120200490713,-0)  2^ 3^ 2 3 +
(0.120200490713,0)  2^ 3^ 3 2 + (-0.168335986252,-0)  0^ 2^ 0 2 +
(0.120200490713,0)  3^ 2^ 2 3 + (-0.120200490713,-0)  3^ 2^ 3 2 +
(0.0454063328691,0)  1^ 3^ 2 0 + (-1.2488468038,-0)  0^ 0 +
(0.0454063328691,0)  3^ 1^ 0 2 + (-0.168335986252,-0)  2^ 0^ 2 0 +
(0.165606823582,0)  3^ 0^ 0 3 + (-0.0454063328691,-0)  2^ 0^ 3 1 +
(0.0454063328691,0)  2^ 0^ 1 3 + (-1.2488468038,-0)  2^ 2 +
(0.0454063328691,0)  2^ 1^ 0 3 + (0.174072892497,0)  3^ 1^ 1 3 +
(-0.479677813134,-0)  1^ 1 + (-0.174072892497,-0)  3^ 1^ 3 1 +
(0.0454063328691,0)  3^ 0^ 1 2 + (-0.165606823582,-0)  3^ 0^ 3 0 +
(0.0454063328691,0)  0^ 3^ 2 1 + (-0.165606823582,-0)  2^ 1^ 2 1 +
(-0.120200490713,-0)  0^ 1^ 0 1 + (-0.120200490713,-0)  1^ 0^ 1 0 + (0.7080240981,0)
'''

H = xacc.getObservable('fermion', opstr)
```

(continues on next page)

(continued from previous page)

```
adapt = xacc.getAlgorithm('adapt', {'accelerator': qpu,
                                    'optimizer': optimizer,
                                    'observable': H,
                                    'n-electrons': 2,
                                    'maxiter': 2,
                                    'sub-algorithm': 'vqe',
                                    'pool': 'qubit-pool'})

adapt.execute(buffer)
```

ADAPT-QAOA

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  // Get reference to the Accelerator
  // specified by --accelerator argument
  auto accelerator = xacc::getAccelerator("qpp");

  // Get reference to the Optimizer
  // specified by --optimizer argument
  auto optimizer = xacc::getOptimizer("nlopt", {std::make_pair("nlopt-optimizer
↪", "l-bfgs")});

  // Allocate 4 qubits in the buffer
  auto buffer = xacc::qalloc(4);

  // Instantiate ADAPT algorithm
  auto adapt = xacc::getService<xacc::Algorithm>("adapt");

  // Specify the operator pool
  auto pool = "multi-qubit-qaoa";

  // Specify the sub algorithm
  auto subAlgo_qaoa = "QAOA";

  // Number of layers
  auto nLayers = 2;

  // This is the cost Hamiltonian
  auto H = xacc::quantum::getObservable(
      "pauli", std::string("-5.0 - 0.5 Z0 - 1.0 Z2 + 0.5 Z3 + 1.0 Z0 Z1 + 2.0␣
↪Z0 Z2 + 0.5 Z1 Z2 + 2.5 Z2 Z3"));

  // Pass parameters to ADAPT algorithm
  adapt->initialize({std::make_pair("accelerator", accelerator),
                                std::make_pair("observable", H),
                                std::make_pair("optimizer", optimizer),
```

(continues on next page)

(continued from previous page)

```
                                std::make_pair("pool", pool),
                                std::make_pair("maxiter", nLayers),
                                std::make_pair("sub-algorithm", subAlgo_qaoa)
                                });

  // Execute ADAPT-QAOA
  adapt->execute(buffer);

  xacc::Finalize();
  return 0;
}
```

In Python:

```
import xacc

accelerator = xacc.getAccelerator("qpp")

buffer = xacc.qalloc(4)

optimizer = xacc.getOptimizer('nlopt',{'nlopt-optimizer':'l-bfgs'})

pool = "multi-qubit-qaoa"

nLayers = 2

subAlgo_qaoa = "QAOA"

H = xacc.getObservable('pauli', '-5.0 - 0.5 Z0 - 1.0 Z2 + 0.5 Z3 + 1.0 Z0 Z1 + 2.0 Z0 Z2␣
↪+ 0.5 Z1 Z2 + 2.5 Z2 Z3')

adapt = xacc.getAlgorithm('adapt', {
                        'accelerator': accelerator,
                        'observable': H,
                        'optimizer': optimizer,
                        'pool': pool,
                        'maxiter': nLayers,
                        'sub-algorithm': subAlgo_qaoa
                        })

adapt.execute(buffer)
```

4.3.4.10 QCMX

The Quantum Connected Moments eXpansion (QCMX) Algorithm requires the following input information: (Kowalski, K. and Peng, Bo. (2018))

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator represents the Hamiltonian	std::shared_ptr<Observable>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
ansatz	State preparation circuit	std::shared_ptr<CompositeInstruction>
cmx-order	The order of the leading term in the CMX	int
expansion-type	Expansion type (Cioslowski, Knowles, PDS)	std::string

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  // Get reference to the Accelerator
  // specified by --accelerator argument
  auto accelerator = xacc::getAccelerator("qpp");

  // Get reference to the Hamiltonian
  // specified by the --observable argument
  auto H = xacc::quantum::getObservable("pauli", std::string("0.2976 + 0.3593␣
↪Z0 - 0.4826 Z1 + 0.5818 Z0 Z1 + 0.0896 X0 X1 + 0.0896 Y0 Y1"));

  // Specify the expansion type
  auto expansion = "Cioslowski";

  // Specify the CMX expansion order
  auto order = 2;

  // Create reference to the initial state
  // specified by the --ansatz argument
  auto provider = xacc::getService<xacc::IRProvider>("quantum");
  auto ansatz = provider->createComposite("initial-state");
  ansatz->addInstruction(provider->createInstruction("X", {(size_t)0}));

  // Allocate 2 qubits in the buffer
  auto buffer = xacc::qalloc(2);

  // Instantiate the QCMX algorithm
  auto qcmx = xacc::getService<xacc::Algorithm>("qcmx");

  // Pass parameters to QCMX algorithm
  qcmx->initialize({{"accelerator",accelerator},
                    {"observable", H},
                    {"ansatz", ansatz},
                    {"cmx-order", order},
```

(continues on next page)

(continued from previous page)

```
                  {"expansion-type", expansion}});

  // Execute QCMX
  qcmx->execute(buffer);

  xacc::Finalize();
  return 0;
}
```

In Python:

```
import xacc

accelerator = xacc.getAccelerator("qpp")

H = xacc.getObservable('pauli', '0.2976 + 0.3593 Z0 - 0.4826 Z1 + 0.5818 Z0 Z1 + 0.0896␣
↪X0 X1 + 0.0896 Y0 Y1')

expansion = 'Cioslowski'

order = 2

provider = xacc.getIRProvider('quantum')
ansatz = provider.createComposite('initial-state')
ansatz.addInstruction(provider.createInstruction('X', [0]))

buffer = xacc.qalloc(2)

qcmx = xacc.getAlgorithm('qcmx', {
                         'accelerator': accelerator,
                         'observable': H,
                         'ansatz': ansatz,
                         'cmx-order': order,
                         'expansion-type': expansion
                         })

qcmx.execute(buffer)
```

4.3.4.11 QEOM

The Quantum Equation of Motion (QEOM) Algorithm requires the following input information: (Ollitrault et al (2020))

Algorithm Parameter	Parameter Description	type
observable	The hermitian operator represents the Hamiltonian	std::shared_ptr<Observable>
accelerator	The Accelerator backend to target	std::shared_ptr<Accelerator>
ansatz	State preparation circuit	std::shared_ptr<CompositeInstruction>
n-electrons	The number of electrons	int
operators	Excitation operators	std::vector<std::shared_ptr<Observable>>

Please note that the algorithm requires either `n-electrons` or `operators`. In the former case, it will default to single and double excitation operators.

```
#include "xacc.hpp"
#include "xacc_service.hpp"
#include "Algorithm.hpp"
#include "xacc_observable.hpp"
#include "OperatorPool.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  // Get reference to the Accelerator
  // specified by --accelerator argument
  auto accelerator = xacc::getAccelerator("qpp");

  // Allocate 4 qubits in the buffer
  auto buffer = xacc::qalloc(4);

  // Get reference to the Hamiltonian
  // specified by the --observable argument
    auto str = std::string(
  "(-0.165606823582,-0)  1^ 2^ 1 2 + (0.120200490713,0)  1^ 0^ 0 1 + "
  "(-0.0454063328691,-0)  0^ 3^ 1 2 + (0.168335986252,0)  2^ 0^ 0 2 + "
  "(0.0454063328691,0)  1^ 2^ 3 0 + (0.168335986252,0)  0^ 2^ 2 0 + "
  "(0.165606823582,0)  0^ 3^ 3 0 + (-0.0454063328691,-0)  3^ 0^ 2 1 + "
  "(-0.0454063328691,-0)  1^ 3^ 0 2 + (-0.0454063328691,-0)  3^ 1^ 2 0 + "
  "(0.165606823582,0)  1^ 2^ 2 1 + (-0.165606823582,-0)  0^ 3^ 0 3 + "
  "(-0.479677813134,-0)  3^ 3 + (-0.0454063328691,-0)  1^ 2^ 0 3 + "
  "(-0.174072892497,-0)  1^ 3^ 1 3 + (-0.0454063328691,-0)  0^ 2^ 1 3 + "
  "(0.120200490713,0)  0^ 1^ 1 0 + (0.0454063328691,0)  0^ 2^ 3 1 + "
  "(0.174072892497,0)  1^ 3^ 3 1 + (0.165606823582,0)  2^ 1^ 1 2 + "
  "(-0.0454063328691,-0)  2^ 1^ 3 0 + (-0.120200490713,-0)  2^ 3^ 2 3 + "
  "(0.120200490713,0)  2^ 3^ 3 2 + (-0.168335986252,-0)  0^ 2^ 0 2 + "
  "(0.120200490713,0)  3^ 2^ 2 3 + (-0.120200490713,-0)  3^ 2^ 3 2 + "
  "(0.0454063328691,0)  1^ 3^ 2 0 + (-1.2488468038,-0)  0^ 0 + "
  "(0.0454063328691,0)  3^ 1^ 0 2 + (-0.168335986252,-0)  2^ 0^ 2 0 + "
  "(0.165606823582,0)  3^ 0^ 0 3 + (-0.0454063328691,-0)  2^ 0^ 3 1 + "
  "(0.0454063328691,0)  2^ 0^ 1 3 + (-1.2488468038,-0)  2^ 2 + "
  "(0.0454063328691,0)  2^ 1^ 0 3 + (0.174072892497,0)  3^ 1^ 1 3 + "
  "(-0.479677813134,-0)  1^ 1 + (-0.174072892497,-0)  3^ 1^ 3 1 + "
  "(0.0454063328691,0)  3^ 0^ 1 2 + (-0.165606823582,-0)  3^ 0^ 3 0 + "
  "(0.0454063328691,0)  0^ 3^ 2 1 + (-0.165606823582,-0)  2^ 1^ 2 1 + "
  "(-0.120200490713,-0)  0^ 1^ 0 1 + (-0.120200490713,-0)  1^ 0^ 1 0 + "
  "(0.7080240981,0)");
  auto H = xacc::quantum::getObservable("fermion", str);

  // Create reference to the initial state
  // specified by the --ansatz argument
  auto pool = xacc::getService<OperatorPool>("singlet-adapted-uccsd");
  pool->optionalParameters({{"n-electrons", 2}});
  pool->generate(buffer->size());
  auto ansatz = xacc::getIRProvider("quantum")->createComposite("ansatz");
```

(continues on next page)

(continued from previous page)

```
  ansatz->addInstruction(
      xacc::getIRProvider("quantum")->createInstruction("X", {0}));
  ansatz->addInstruction(
      xacc::getIRProvider("quantum")->createInstruction("X", {2}));
  ansatz->addVariable("x0");
  for (auto &inst : pool->getOperatorInstructions(2, 0)->getInstructions()) {
    ansatz->addInstruction(inst);
  }
  auto kernel = ansatz->operator()({0.0808});

  // Instantiate the QEOM algorithm
  auto qeom = xacc::getAlgorithm("qeom");

  // Pass parameters to QEOM algorithm
  qeom->initialize({{"accelerator",accelerator},
                    {"observable", H},
                    {"ansatz", kernel},
                    {"n-electrons", 2}});

  // Execute QEOM
  qeom->execute(buffer);

  xacc::Finalize();
  return 0;
}
```

In Python:

```
import xacc

accelerator = xacc.getAccelerator("qpp")

buffer = xacc.qalloc(4)

opstr = '''
(-0.165606823582,-0)  1^ 2^ 1 2 + (0.120200490713,0)  1^ 0^ 0 1 +
(-0.0454063328691,-0)  0^ 3^ 1 2 + (0.168335986252,0)  2^ 0^ 0 2 +
(0.0454063328691,0)  1^ 2^ 3 0 + (0.168335986252,0)  0^ 2^ 2 0 +
(0.165606823582,0)  0^ 3^ 3 0 + (-0.0454063328691,-0)  3^ 0^ 2 1 +
(-0.0454063328691,-0)  1^ 3^ 0 2 + (-0.0454063328691,-0)  3^ 1^ 2 0 +
(0.165606823582,0)  1^ 2^ 2 1 + (-0.165606823582,-0)  0^ 3^ 0 3 +
(-0.479677813134,-0)  3^ 3 + (-0.0454063328691,-0)  1^ 2^ 0 3 +
(-0.174072892497,-0)  1^ 3^ 1 3 + (-0.0454063328691,-0)  0^ 2^ 1 3 +
(0.120200490713,0)  0^ 1^ 1 0 + (0.0454063328691,0)  0^ 2^ 3 1 +
(0.174072892497,0)  1^ 3^ 3 1 + (0.165606823582,0)  2^ 1^ 1 2 +
(-0.0454063328691,-0)  2^ 1^ 3 0 + (-0.120200490713,-0)  2^ 3^ 2 3 +
(0.120200490713,0)  2^ 3^ 3 2 + (-0.168335986252,-0)  0^ 2^ 0 2 +
(0.120200490713,0)  3^ 2^ 2 3 + (-0.120200490713,-0)  3^ 2^ 3 2 +
(0.0454063328691,0)  1^ 3^ 2 0 + (-1.2488468038,-0)  0^ 0 +
(0.0454063328691,0)  3^ 1^ 0 2 + (-0.168335986252,-0)  2^ 0^ 2 0 +
(0.165606823582,0)  3^ 0^ 0 3 + (-0.0454063328691,-0)  2^ 0^ 3 1 +
(0.0454063328691,0)  2^ 0^ 1 3 + (-1.2488468038,-0)  2^ 2 +
```

(continues on next page)

(continued from previous page)

```
(0.0454063328691,0)  2^ 1^ 0 3 + (0.174072892497,0)  3^ 1^ 1 3 +
(-0.479677813134,-0)  1^ 1 + (-0.174072892497,-0)  3^ 1^ 3 1 +
(0.0454063328691,0)  3^ 0^ 1 2 + (-0.165606823582,-0)  3^ 0^ 3 0 +
(0.0454063328691,0)  0^ 3^ 2 1 + (-0.165606823582,-0)  2^ 1^ 2 1 +
(-0.120200490713,-0)  0^ 1^ 0 1 + (-0.120200490713,-0)  1^ 0^ 1 0 + (0.7080240981,0)
'''
H = xacc.getObservable('fermion', opstr)

pool = xacc.quantum.getOperatorPool("singlet-adapted-uccsd")
pool.optionalParameters({"n-electrons": 2})
pool.generate(buffer.size())
provider = xacc.getIRProvider('quantum')
ansatz = provider.createComposite('initial-state')
ansatz.addInstruction(provider.createInstruction('X', [0]))
ansatz.addInstruction(provider.createInstruction('X', [2]))
ansatz.addVariable("x0")
for inst in pool.getOperatorInstructions(2, 0).getInstructions():
    ansatz.addInstruction(inst)
kernel = ansatz.eval([0.0808])

qeom = xacc.getAlgorithm('qeom', {
                         'accelerator': accelerator,
                         'observable': H,
                         'ansatz': kernel,
                         'n-electrons': 2,
                         })

qeom.execute(buffer)
```

4.3.5 Accelerator Decorators

4.3.5.1 ROErrorDecorator

The `ROErrorDecorator` provides an `AcceleratorDecorator` implementation for affecting readout error mitigation as in the deuteron paper. It takes as input readout error probabilities `p(0|1)` and `p(1|0)` for all qubits and shifts expecation values accordingly (see paper).

By default it will request the backend properties from the decorated `Accelerator` (`Accelerator::getProperties()`). This method returns a `HeterogeneousMap`. If this map contains a vector of doubles at keys `p01s` and `p10s`, then these values will be used in the readout error correction. Alternatively, if the backend does not provide this data, users can provide a custom JSON file containing the probabilities. This file should be structured as such

```
{
    "shots": 1024,
    "backend": "qcs:Aspen-2Q-A",
    "0": {
        "0|1": 0.0565185546875,
        "1|0": 0.0089111328125,
        "+": 0.0654296875,
        "-": 0.047607421875
```

(continues on next page)

(continued from previous page)

```
    },
    "1": {
        "0|1": 0.095458984375,
        "1|0": 0.0115966796875,
        "+": 0.1070556640625,
        "-": 0.0838623046875
    }
}
```

Automating readout error mitigation with this decorator can be done in the following way:

```
qpu = xacc.getAccelerator('ibm:ibmq_johannesburg', {'shots':1024})

# Turn on readout error correction by decorating qpu
qpu = xacc.getAcceleratorDecorator('ro-error', qpu)

# Now use qpu as your Accelerator...
# execution will be automatically readout
# error corrected
```

Similarly, with a provided configuration file

```
auto qpu = xacc::getAccelerator("qcs:Aspen-2Q-A");
qpu = xacc::getAcceleratorDecorator("ro-error", qpu, {std::make_pair("file", "probs.json
↪")});
```

See readout_error_correction_aer.py for a full example demonstrating the utility of the `ROErrorDecorator`.

4.3.5.2 RDMPurificationDecorator

4.3.5.3 ImprovedSamplingDecorator

4.3.5.4 VQE Restart Decorator

4.3.6 IR Transformations

4.3.6.1 CircuitOptimizer

This `IRTransformation` of type `Optimization` will search the DAG representation of a quantum circuit and remove all zero-rotations, hadamard and cnot pairs, and merge adjacent common rotations (e.g. `Rx(.1)Rx(.1) -> Rx(.2)`).

```
# Create a bell state program with too many cnots
xacc.qasm('''
.compiler xasm
.circuit foo
.qbit q
H(q[0]);
CX(q[0], q[1]);
CX(q[0], q[1]);
CX(q[0], q[1]);
Measure(q[0]);
```

(continues on next page)

(continued from previous page)

```
Measure(q[1]);
''')
f = xacc.getCompiled('foo')
assert(6 == f.nInstructions())

# Run the circuit-optimizer IRTransformation, can pass
# accelerator (here None) and options (here empty dict())
optimizer = xacc.getIRTransformation('circuit-optimizer')
optimizer.apply(f, None, {})

# should have 4 instructions, not 6
assert(4 == f.nInstructions())
```

4.3.7 Observables

4.3.7.1 Psi4 Frozen-Core

The `psi4-frozen-core` observable generates an fermionic observable using Psi4 and based on a user provided dictionary of options. To use this Observable, ensure you have Psi4 installed under the same `python3` used for the XACC Python API.

```
$ git clone https://github.com/psi4/psi4 && cd psi4 && mkdir build && cd build
$ cmake .. -DPYTHON_EXECUTABLE=$(which python3) -DCMAKE_INSTALL_PREFIX=$(python3 -m site␣
↪--user-site)/psi4
$ make -j8 install
$ export PYTHONPATH=$(python3 -m site --user-site)/psi4/lib:$PYTHONPATH
```

This observable type takes a dictionary of options describing the molecular geometry (key `geometry`), the basis set (key `basis`), and the list of frozen (key `frozen-spin-orbitals`) and active (key `active-spin-orbitals`) spin orbital lists.

With Psi4 and XACC installed, you can use the frozen-core Observable in the following way in python.

```
import xacc

geom = '''
0 1
Na  0.000000   0.0      0.0
H   0.0        0.0  1.914388
symmetry c1
'''
fo = [0, 1, 2, 3, 4, 10, 11, 12, 13, 14]
ao = [5, 9, 15, 19]

H = xacc.getObservable('psi4-frozen-core', {'basis': 'sto-3g',
                                            'geometry': geom,
                                            'frozen-spin-orbitals': fo,
                                            'active-spin-orbitals': ao})
```

4.3.8 Circuit Generator

4.3.8.1 ASWAP Ansatz Circuit

The `ASWAP` circuit generator generates a state preparation (ansatz) circuit for the VQE Algorithm. (See Gard, Bryan T., et al.)

The `ASWAP` circuit generator requires the following input information:

Algorithm Parameter	Parameter Description	type
nbQubits	The number of qubits in the circuit.	int
nbParticles	The number of particles.	int
timeReversalSymmetry	Do we have time-reversal symmetry?	boolean

Example:

```
#include "xacc.hpp"
#include "xacc_observable.hpp"
#include "xacc_service.hpp"

int main(int argc, char **argv) {
  xacc::Initialize(argc, argv);

  auto accelerator = xacc::getAccelerator("qpp");
  auto H_N_2 = xacc::quantum::getObservable(
      "pauli", std::string("5.907 - 2.1433 X0X1 "
                           "- 2.1433 Y0Y1"
                           "+ .21829 Z0 - 6.125 Z1"));

  auto optimizer = xacc::getOptimizer("nlopt");
  // Use the ASWAP circuit as the ansatz
  xacc::qasm(R"(
      .compiler xasm
      .circuit deuteron_ansatz
      .parameters t0
      .qbit q
      ASWAP(q, t0, {{"nbQubits", 2}, {"nbParticles", 1}});
  )");
  auto ansatz = xacc::getCompiled("deuteron_ansatz");

  // Get the VQE Algorithm and initialize it
  auto vqe = xacc::getAlgorithm("vqe");
  vqe->initialize({std::make_pair("ansatz", ansatz),
                   std::make_pair("observable", H_N_2),
                   std::make_pair("accelerator", accelerator),
                   std::make_pair("optimizer", optimizer)});

  // Allocate some qubits and execute
  auto buffer = xacc::qalloc(2);
  vqe->execute(buffer);
  // Expected result: -1.74886
  std::cout << "Energy: " << (*buffer)["opt-val"].as<double>() << "\n";
 }
```

4.3.8.2 QFAST Circuit Synthesis

The `QFAST` circuit generator generates a quantum circuit for an arbitary unitary matrix. (See Ed Younis, et al.)

The `QFAST` circuit generator only requires the `unitary` input information. Optionally, we can provide additional configurations as listed below.

Algorithm Parameter	Parameter Description	type
unitary	The unitary matrix.	Eigen::MatrixXcd/numpy 2-D array
trace-distance	The target trace distance of the Hilbert-Schmidt inner produc	double (default = 0.01)
explore-trace-distance	The *stopping-condition* trace distance for the Explore phase.	double (default = 0.1)
initial-depth	The initial number of circuit layers.	int (default = 1)

By default, after each unitary matrix decomposition, the QFAST plugin will save the result (in terms of smaller block matrices) into a cache file located at `$XACC_INSTALL_DIR/tmp`. If a cache entry is found, the QFAST plugin will re-use the result automatically.

Users can modify the cache filename via the configuration key `cache-file-name`.

Example:

In Cpp,

```
auto qfast = std::dynamic_pointer_cast<quantum::Circuit>(xacc::getService<Instruction>(
↪"QFAST"));
Eigen::MatrixXcd ccnotMat = Eigen::MatrixXcd::Identity(8, 8);
ccnotMat(6, 6) = 0.0;
ccnotMat(7, 7) = 0.0;
ccnotMat(6, 7) = 1.0;
ccnotMat(7, 6) = 1.0;

const bool expandOk = qfast->expand({
  std::make_pair("unitary", ccnotMat)
});
```

In Python,

```
import xacc, numpy as np

# CCNOT matrix:
# This takes a 2-D numpy array.
ccnotMat = np.identity(8, dtype = np.cdouble)
ccnotMat[6][6] = 0.0
ccnotMat[7][7] = 0.0
ccnotMat[6][7] = 1.0
ccnotMat[7][6] = 1.0

composite = xacc.createCompositeInstruction('QFAST', { 'unitary' : ccnotMat })
print(composite)
```

4.3.9 Placement

4.3.9.1 Noise Adaptive Layout

The `triQ` placement IRTransformation plugin implements the noise adaptive layout mapping method (See Prakash Murali, et al.)

The `triQ` plugin will automatically retrieve backend information (e.g. gate and readout fidelity) from the backend `Accelerator` if available.

Otherwise, users can provide the backend configurations in JSON format using the `backend-json` key. The backend Json schema must conform to IBMQ specifications.

The `triQ` plugin depends on the Z3 SMT solver (version >= 4.8).

Z3 can be installed via:

- Ubuntu:

```
apt-get install -y libz3-dev
```

- MacOS:

```
brew install z3
```

Example:

```
auto compiler = xacc::getCompiler("staq");
// Specify a remote IBM backend
auto accelerator = xacc::getAccelerator("ibm:ibmq_16_melbourne");
auto q = xacc::qalloc(5);
q->setName("q");
xacc::storeBuffer(q);
auto IR = compiler->compile(R"(__qpu__ void f(qreg q) {
                                OPENQASM 2.0;
                                include "qelib1.inc";
                                creg c[16];
                                cx q[2],q[1];
                                cx q[1],q[2];
                                cx q[3],q[2];
                                cx q[2],q[3];
                                cx q[4],q[3];
                                cx q[3],q[4];
                                h q[0];
                                t q[4];
                                t q[3];
                                t q[0];
                                cx q[3],q[4];
                                cx q[0],q[3];
                                cx q[4],q[0];
                                tdg q[3];
                                })");

auto program = IR->getComposites()[0];
// Apply noise adaptive layout (TriQ)
```

(continues on next page)

(continued from previous page)

```
auto irt = xacc::getIRTransformation("triQ");
irt->apply(program, accelerator);
```

4.4 Advanced

4.4.1 AcceleratorBuffer Execution Data

4.4.2 Error Mitigation

4.4.3 Pulse-level Programming

4.4.3.1 Pulse-level results in AcceleratorBuffer

By default, if using the QuaC simulator backend, the following information is embedded into the Accelerator Buffer at the end of the simulation:

- <O>: expectation value of the number/occupation operator (`n`) on each qubit sub-system. This is an array of floating-point numbers, one entry for each qubit.
- `DensityMatrixDiags`: diagonal elements of the density matrix at the end of the simulation (length = `dim^n`, `dim` is the dimension of sub-systems (2 for qubits, 3 for qutrits, etc.) and `n` is the number of sub-systems)

To optimize the execution speed, we don't record time-stepping data by default when integrating the master equation. This can be enabled manually by specifying a `logging-period` parameter when requesting the QuaC simulator as follows:

```
qpu = xacc.getAccelerator('QuaC', {'system-model': model.name(), 'logging-period': 0.1 })
```

Once requested, time-stepping data will be saved as a CSV file whose path is recorded in the Accelerator Buffer's `csvFile` field. The following data is recorded for each logging period: current time, signals on channels, expectation values of the number operator and Pauli operators on each qubit sub-system.

Users can load the data, e.g. for plotting purposes, as follows:

```
csvFile = qubitReg['csvFile']
data = np.genfromtxt(csvFile, delimiter = ',', dtype=float, names=True)
# Each field can then be referred to by name
time = data['Time']
expectationX0 = data['X0']
```

4.4.3.2 Lab-frame vs. Rotating frame

Qubit (two-level) systems always have a ground-to-excited state transition frequency which corresponds to rotation around the z-axis of the excited state. Hence, driving signals are often mixed with a local-oscillator at the frequency in order to be in resonance with that transition.

In QuaC, this is done by setting the loFregs_dChannels array of the `BackendChannelConfigs`:

```
channelConfigs = xacc.BackendChannelConfigs()
channelConfigs.loFregs_dChannels = [4.98, 4.34]
```

The QuaC accelerator will mix pulse instructions assigned on each channel with its corresponding LO signals at the specified frequency.

This is the most accurate form of simulation. However, it often requires a very fine time-stepping procedure due to the oscillatory nature of the modulated signals. Users can opt for a simplified simulation setting whereby the system dynamics are specified in the rotating frame which is rotating at that transition frequency.

This can be done by:

- Setting the transition frequency variable in the Hamiltonian JSON to zero.
- Setting the LO frequency to zero.

4.4.3.3 Initial Population & Qubit Decay

By default, all qubits are initialized in the ground state. This can be changed by using the `setQubitInitialPopulation` function (first parameter is the qubit index and second parameter is the initial value of the number operator).

Similarly, qubit decay rate can be specified by proving a T1 value via the `setQubitT1` function which corresponds to a Linbladian decay rate of 1/T1 in the master equation.

```
model = xacc.createPulseModel()
model.setQubitInitialPopulation(0, 1.0)
model.setQubitT1(0, 1.0)
```

4.4.3.4 Higher-dimensional systems

Higher-dimensional systems are also supported by QuaC. The sub-system dimensions can be specified in the `qub` field of the Hamiltonian JSON.

For example, to model transmon qubits as three-level systems (e.g. to investigate qubit leakage), one can use the following Hamiltonian JSON.

```
hamiltonianJson = {
    "description": "Two-qutrit Hamiltonian",
    "h_latex": "",
    "h_str": ["w_0*O0", "w_1*O1", "d*O0*(O0-I0)", "d*O1*(O1-I1)", "J*(SP0*SM1 + SM0*SP1)
↪", "O*(SM0 + SP0)||D0"],
    "osc": {},
    "qub": {
        "0": 3,
        "1": 3
    },
    "vars": {
        "w_0": 5.114,
        "w_1": 4.914,
        "d": -0.33,
        "J": 0.004,
        "O": 0.060
    }
}
```

A few limitations of using non-qubit systems:

- The shot-count distribution (binary bit strings) simulation is not supported. Users have access to the list of diagonal elements of the density matrix embedded in the Accelerator Buffer which contains the state distribution.
- Some automatic IR transformation services are not compatible with non-qubit systems.

4.4.3.5 Pulse-level IR Transformation

Automatic quantum-circuit-to-pulse transformation is a service within the XACC which can be used in conjunction with the QuaC simulator backend to find a pulse program representing arbitrary quantum circuit.

The XACC pulse-level IR transformation service can be requested by its name, which is `quantum-control`.

```
optimizer = xacc.getIRTransformation('quantum-control')
```

In order to transform a quantum circuit (CompositeInstruction) into pulses, the optimizer will need access to an instance of the QuaC simulator backend which has been initialized with the system dynamics. Also, users will need to provide optimization options to the IR Transformation service. The available options are:

Parameter	Parameter Description	type
method	Optimization method ('GOAT' or 'GRAPE')	string
max-time	Max time horizon for pulse optimization	double
dt	Sample duration (GRAPE-only)	double
control-params	Control parameters to optimize (GOAT-only)	std::vector<string>
control-funcs	Analytical forms of control functions (GOAT-only)	std::vector<string>
initial-parameters	Initial values of control parameters (GOAT-only)	std::vector<double>

For example, we can transform a quantum circuit into an optimized pulse (Gaussian form) then verify the result by simulating with QuaC:

```
# Get the XASM compiler
xasmCompiler = xacc.getCompiler('xasm');
# Composite to be transform to pulse
ir = xasmCompiler.compile('''__qpu__ void f(qbit q) {
    Rx(q[0], 1.57);
}''', qpu);
program = ir.getComposites()[0]

# Run the pulse IRTransformation
optimizer = xacc.getIRTransformation('quantum-control')
optimizer.apply(program, qpu, {
    'method': 'GOAT',
    'control-params': ['sigma'],
    # Gaussian pulse
    'control-funcs': ['exp(-t^2/(2*sigma^2))'],
    # Initial params
    'initial-parameters': [8.0],
    'max-time': 100.0
})

# This composite should be a pulse composite now
print(program)

# Run the simulation of the optimized pulse program
```

(continues on next page)

(continued from previous page)

```
qubitReg = xacc.qalloc(1)
qpu.execute(qubitReg, program)
```

4.4.3.6 Enable MPI

Users can enable MPI multi-processing on QuaC (C++ only) by setting the `execution-mode` option to `MPI::<number of MPI processes>` when requesting the QuaC accelerator.

For example, to request a QuaC accelerator which will run on 4 MPI processes:

```
auto quaC = xacc::getAccelerator("QuaC", std::make_pair("execution-mode", "MPI::4") ... }
↪);
```

A few notes:

- The compiled executable must be started on a single MPI process, i.e. `-n 1` (or `-np 1`). QuaC runtime will spawn additional processes as required.
- We recommend the Hydra process manager (`mpiexec.hydra`) that is installed with PETSc (`--download-mpich` when `configure` PETSc).
- MPI multi-processing should only be used for large systems (>5 qubits.) There is no performance gain when using MPI for small systems.

4.5 Developers

Here we describe how XACC developers can extend the framework with new Compilers, Accelerators, Instructions, IR Transformations, etc. This can be done from both C++ and Python.

4.5.1 Quick Start with Docker

We have put together a docker image based on Ubuntu 18.04 that has all required dependencies for building XACC. Moreover, we have set this image up to serve an Eclipse Theia IDE on `localhost:3000`. To use this image run the following from some scratch development directory:

```
$ docker run --security-opt seccomp=unconfined --init -it -p 3000:3000 xacc/xacc
```

Now navigate to `localhost:3000` in your web browser. This will open the Theia IDE and you are good to go. Open a terminal with `ctrl + ``.

4.5.2 Writing a Plugin in C++

Let's demonstrate how one might add a new IR Transformation implementation to XACC. This is a simple case, but the overall structure works across most plugins.

First, we create a new project folder `test_ir_transformation` and populate it with a `CMakeLists.txt` file, and a `src` folder containing another `CMakeLists.txt` file as well as `manifest.json`, `test_ir_transformation.hpp`, `test_ir_transformation.cpp`, and `test_ir_transformation_activator.cpp`. You should have the following directory structure

```
test_ir_transformation
├── CMakeLists.txt
├── src
    ├── CMakeLists.txt
    └── manifest.json
    └── test_ir_transformation.hpp
    └── test_ir_transformation.cpp
    └── test_ir_transformation_activator.cpp
```

In the top-level `CMakeLists.txt` we add the following:

```
project(test_ir_transformation CXX)
cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
find_package(XACC REQUIRED)
add_subdirectory(src)
```

Basically here we are defining a CMake project, setting the minimum version, locating our XACC install, and adding the `src` directory to the build.

In the `src/CMakeLists.txt` file, we add the following

```
set(LIBRARY_NAME test-ir-transformation)
file(GLOB SRC *.cpp)
usfunctiongetresourcesource(TARGET
                            ${LIBRARY_NAME}
                            OUT
                            SRC)
usfunctiongeneratebundleinit(TARGET
                            ${LIBRARY_NAME}
                            OUT
                            SRC)
add_library(${LIBRARY_NAME} SHARED ${SRC})
target_link_libraries(${LIBRARY_NAME} PRIVATE xacc::xacc)
set(_bundle_name test_ir_transformation)
set_target_properties(${LIBRARY_NAME}
                      PROPERTIES COMPILE_DEFINITIONS
                                 US_BUNDLE_NAME=${_bundle_name}
                                 US_BUNDLE_NAME
                                 ${_bundle_name})
usfunctionembedresources(TARGET
                         ${LIBRARY_NAME}
                         WORKING_DIRECTORY
                         ${CMAKE_CURRENT_SOURCE_DIR}
                         FILES
                         manifest.json)
xacc_configure_plugin_rpath(${LIBRARY_NAME})
install(TARGETS ${LIBRARY_NAME} DESTINATION ${CMAKE_INSTALL_PREFIX}/plugins)
```

Here we define the library name, collect all source files, run some CppMicroServices functions that append extra information to our library, build the library and link in all required XACC libraries. Next we add more information to this shared library from the `manifest.json` file, configure the libraries RPATH, and install to the correct `plugins` folder in XACC. `manifest.json` should contain the following json

```
{
  "bundle.symbolic_name" : "test_ir_transformation",
  "bundle.activator" : true,
  "bundle.name" : "Test IR Transformation",
  "bundle.description" : ""
}
```

Next we provide the actual code for the test IR Transformation. In the `test_ir_transformation.hpp` we add the following

```
#pragma once
#include "IRTransformation.hpp"

using namespace xacc;

namespace test {

class Test : public IRTransformation {
public:
  Test() {}
  void apply(std::shared_ptr<CompositeInstruction> program,
                     const std::shared_ptr<Accelerator> accelerator,
                     const HeterogeneousMap& options = {}) override;
  const IRTransformationType type() const override {return␣
↪IRTransformationType::Optimization;}

  const std::string name() const override { return "test-irt"; }
  const std::string description() const override { return ""; }
};
}
```

and in `test_ir_transformation.cpp` we implement `apply`

```
#include "test_ir_transformation.hpp"

namespace test {

void Test::apply(std::shared_ptr<CompositeInstruction> circuit,
                              const std::shared_ptr<Accelerator> accelerator,
                              const HeterogeneousMap &options) {

  // do transformation on circuit here...
}
}
```

Finally, we add a `BundleActivator` that creates a `shared_ptr` to our IR Transformation and registers it with the CppMicroServices framework.

```
#include "test_ir_transformation.hpp"

#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceProperties.h"
```

(continues on next page)

(continued from previous page)

```
#include <memory>

using namespace cppmicroservices;

namespace {

class US_ABI_LOCAL TestIRTransformationActivator: public BundleActivator {

public:

        TestIRTransformationActivator() {
        }
        void Start(BundleContext context) {
                auto t = std::make_shared<test::Test>();
        context.RegisterService<xacc::IRTransformation>(t);
        }
        void Stop(BundleContext /*context*/) {
        }
};

}

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(TestIRTransformationActivator)
```

The majority of this is standard CppMicroservices boilerplate code. The crucial bit that requires your attention when developing a new plugin is the implementation of `Start`. Here you create a `shared_ptr` to your instances and register it against the correct XACC interface type, here `IRTransformation`.

Now, all that is left to do is build your shared library, and install it for use in the XACC framework

```
$ cd test_ir_transformation && mkdir build && cd build
$ cmake .. -DXACC_DIR=~/.xacc
$ make install
```

4.5.3 Writing a Plugin in Python

For this example, let's wrap a Qiskit transpiler pass with an XACC `IRTransformation` to demonstrate how one might integrate novel tools from vendor frameworks with XACC. This will require creating a new Python class in a standalone python file that extends the core C++ `IRTransformation` interface. Note that this can be done for other interfaces as well, including `Accelerator`, `Observable`, `Optimizer`, etc.

First lets show the code to do this, and then we'll walk through it. We will wrap the simple qiskit cx-cancellation pass (this is already in XACC from the `circuit-optimizer` `IRTransformation`, but this is for demonstration purposes). Create a python file named `easy_qiskit_pass.py` and add the following

```
import xacc
from pelix.ipopo.decorators import ComponentFactory, Property, Requires, Provides, \
    Validate, Invalidate, Instantiate

@ComponentFactory("easy_qiskit_pass_factory")
@Provides("irtransformation")
```

(continues on next page)

(continued from previous page)

```
@Property("_irtransformation", "irtransformation", "qiskit-cx-cancellation")
@Property("_name", "name", "qiskit-cx-cancellation")
@Instantiate("easy_qiskit_pass_instance")
class EasyQiskitIRTransformation(xacc.IRTransformation):
    def __init__(self):
        xacc.IRTransformation.__init__(self)

    def type(self):
        return xacc.IRTransformationType.Optimization

    def name(self):
        return 'qiskit-cx-cancellation'

    def apply(self, program, accelerator, options):
        # Import qiskit modules here so that users
        # who don't have qiskit can still use rest of xacc
        from qiskit import QuantumCircuit, transpile
        from qiskit.transpiler import PassManager
        from qiskit.transpiler.passes import CXCancellation

        # Map CompositeInstruction program to OpenQasm string
        openqasm_compiler = xacc.getCompiler('openqasm')
        src = openqasm_compiler.translate(program).replace('\\','')

        # Create a QuantumCircuit
        circuit = QuantumCircuit.from_qasm_str(src)

        # Create the PassManager and run the pass
        pass_manager = PassManager()
        pass_manager.append(CXCancellation())
        out_circuit = transpile(circuit, pass_manager=pass_manager)

        # Map the output to OpenQasm and map to XACC IR
        out_src = out_circuit.qasm()
        out_src = '__qpu__ void '+program.name()+'(qbit q) {\n'+out_src+"\n}"
        out_prog = openqasm_compiler.compile(out_src, accelerator).getComposites()[0]

        # update the given program CompositeInstruction reference
        program.clear()
        for inst in out_prog.getInstructions():
            program.addInstruction(inst)

        return
```

This class subclasses the Pybind11-exposed C++ `IRTransformation` interface, and provides implementations in python of its pertinent methods - a constructor, `type()`, `name()`, and `apply()`. The constructor must invoke the superclass constructor. We implement `type()` to indicate that this is an `IRTransformation` that is of type `Optimization`. Crucially important is the `name()` method, you must implement this to contribute the unique name of this `IRTransformation`. This name will be how users get reference to this `IRTransformation` implementation. And finally, you must implement the primary method for `IRTransformation`, `apply`. This is where the actual transformation (optimization) is performed.

To insure that users can leverage the XACC framework Python API without qiskit installed, we have to place our imports

in the `apply` method so that they are not imported at framework initialization. The rest of the `apply` code takes the XACC `CompositeInstruction` (`program`) and converts it to an OpenQasm string with the appropriate `openqasm` `Compiler` implementation. From this we can construct a Qiskit `QuantumCircuit` and pass this to the `transpile` command orchestrating the execution of the `CXCancellation` pass. Now we get the optimized circuit back out and map back to XACC IR and update the provided `program` instance.

In order to contribute this `IRTransformation` to XACC as a plugin, we rely on the IPOPO project. To expose this class as a plugin, we annotate it with the demonstrated class decorators, indicating what it provides and its unique name. These lines are basic boilerplate, update them for your specific plugin contribution.

If this file is installed to the `py-plugins` directory of your XACC install, then when someone runs `import xacc`, this plugin will be loaded and contributed to the core C++ XACC plugin registry, and users can query it like any other service.

```
import xacc

qpu = xacc.getAccelerator('aer')
qbits = xacc.qalloc(2)

# Create a bell state program with too many cnots
xacc.qasm('''
.compiler xasm
.circuit foo
.qbit q
H(q[0]);
CX(q[0], q[1]);
CX(q[0], q[1]);
CX(q[0], q[1]);
Measure(q[0]);
Measure(q[1]);
''')
f = xacc.getCompiled('foo')

# Run the python contributed IRTransformation that uses qiskit
optimizer = xacc.getIRTransformation('qiskit-cx-cancellation')
optimizer.apply(f, None, {})

# should have 4 instructions, not 6
assert(4 == f.nInstructions())
```

4.5.4 Extending Accelerator for new Simulators

Here we document how one might extend the `Accelerator` interface for new simulators.

4.6 Tutorials

4.6.1 Pulse Control Tutorial

Here we describe how users can easily leverage XACC with the QuaC Open-Pulse Simulator to conduct optimal control experiments. We currently support the following control algorithms: GRAPE [1], CRAB [2], Krotov [3], GOAT [4], and DRAG [5] with near-term plans of supporting Deep Reinforcement Learning and GRAFS [6].

4.6.1.1 Quick Start with Docker

We have put together a docker image based on Ubuntu 18.04 that has all required dependencies for building XACC and QuaC. Moreover, we have set this image up to serve an Eclipse Theia IDE on `localhost:3000`. To use this image run the following from some scratch development directory:

```
$ docker run --security-opt seccomp=unconfined --init -it -p 3000:3000 xacc/xacc-quac
```

Now navigate to `localhost:3000` in your web browser. This will open the Theia IDE and you are good to go. Open a terminal with `ctrl + ``.

4.6.1.2 Basics of Manipulating Quantum Systems in XACC

We will begin by showing how to define a quantum system in XACC, and subsqequently demonstrate how to manipulate the system. The next section will cover optimizing controls for the system through the use of XACC's Quantum Control algorithms.

Make sure to run the following imports:

```
import xacc
import sys, os, json, numpy as np

# Alternative to the following two lines is to run
# from the IDE terminal: export PYTHONPATH=$PYTHONPATH:$HOME/.xacc
from pathlib import Path
sys.path.insert(1, str(Path.home()) + '/.xacc')
```

Each file then begins by defining the Hamiltonian of the system in jSON format:

```
hamiltonianJson = {
    "description": "One-qutrit Hamiltonian.",
    "h_latex": "",
    "h_str": ["(w - 0.5*alpha)*O0", "0.5*alpha*O0*O0", "O*(SM0 + SP0)||D0"],
    "osc": {},
    "qub": {
        "0": 3
    },
    "vars": {
        "w": 31.63772297724,
        "alpha": -1.47969,
        "O": 0.0314
```

(continues on next page)

(continued from previous page)

```
    }
}
```

with the above being an example of a single qutrit system. For more information on formatting the Hamiltonian, see Advanced/Pulse-level Programming . Alternatively, in QuaC/xacc_examples/python, there are several example files outlining definitions for one-qubit, one-qutrit, two-qubit, and two-qutrit Hamiltonians that users can plug-and-play with.

Next, a pulse model must be instantiated and the Hamiltonian is passed to the module by calling:

```
model = xacc.createPulseModel()
loadResult = model.loadHamiltonianJson(json.dumps(hamiltonianJson))
qpu = xacc.getAccelerator('QuaC', {'system-model': model.name()})
channelConfig = xacc.BackendChannelConfigs()
```

See Tutorials/Alternative Hamiltonian Declaration for declaring a Hamiltonian through default backends.

Let's now define some of the parameters of the pulse, beginning with the total pulse time in nanoseconds, the number of samples, the time between the samples (dt), and the frequency of the driving envelope (typically chosen to be on resonance with the qubit):

```
T = 100
nbSamples = 100
# dt (time between data samples)
channelConfig.dt = nbSamples / T
# Drive at resonance: 31.63772297724/(2pi)
channelConfig.loFregs_dChannels = [5.0353]
```

XACC currently supports several pre-installed pulse declarations:

Pulse Type	Parameters	Parameter Description	type
SquarePulse	nbSamples	Number of samples in the pulse	int
GaussianPulse	nbSamples	Number of samples in the pulse	int
	sigma	Standard deviation of Gaussian distribution	double
GaussianSquare	duration	Total pulse duration	int
	amplitude	Min/max amplitude of pulse	double
	sigma	Standard deviation of Gaussian distribution	double
	width	Width of pulse peak/trough	int
DragPulse	duration	Total pulse duration	int
	amplitude	Amplitude of driving envelope	double
	sigma	Standard deviation of Gaussian distribution	double
	beta	Correction amplitude	double
SlepianPulse	alpha_vector	Weights for all k-orders of Slepians	array
	nbSamples	Number of samples in the pulse	int
	in_bW	Half-bandwith of Slepian sequences	double
	in_K	Max number of orders to use	int

which may be called as follows:

```
channelConfig.addOrReplacePulse('square', xacc.SquarePulse(nbSamples))
# channelConfig.addOrReplacePulse('gaussian', xacc.GaussianPulse(nSamples, sigma = 0.1))
# etc.
```

XACC currently supports the use of Discrete Prolate Spheroidal Sequences [7], or Slepians, for creating time and bandwidth limited discrete pulses. First applied directly to qubit control in [6], these show promise at creating accurate and smooth controls in the NISQ era.

```
# Typically want more samples here to maintain precision
nbSamples = 500

# Half-bandwidth \in (0.0, 0.5]
in_bW = 0.02

# Maximum number of Slepian orders to use
# Typically (2 * nbSamples * W) -- remember to make it an integer
in_K = int(2 * nbSamples * in_bW)

# Weight vector of length in_K as array. Just using one's
# as an example, but for optimal control purposes, this vector
# is the array that we seek to optimize.
alpha_vector = np.ones(in_K)

channelConfig.addOrReplacePulse('slepian', xacc.SlepianPulse(alpha_vector, nbSamples, in_
↪bW, in_K))
```

Alternatively, one may define a custom pulse in numpy array format:

```
pulseData = np.ones(nbSamples)
pulseName = 'custom'
xacc.addPulse(pulseName, pulseData)
```

Now we allocate the amount of qubits needed for the program, create the program containing the pulse, and set the channel to drive it on:

```
# Allocate qubits:
q = xacc.qalloc(1)
# Create the quantum program that contains the custom pulse
# and the drive channel (D0) is set on the instruction
provider = xacc.getIRProvider('quantum')
prog = provider.createComposite('pulse')
customPulse = provider.createInstruction(pulseName, [0])
customPulse.setChannel('d0')
prog.addInstruction(customPulse)
```

Finally, we instruct the program on what measurement we'd like it to make and execute the program:

```
# Measure Q0
prog.addInstruction(xacc.gate.create("Measure", [0]))
qpu.execute(q, prog)
```

4.6.1.3 Returning the Fidelity

Depending on the backend that you're targetting, the gate operation you're attempting to do, and the number of qubits in your system, there are different ways to return the fidelity.

Case 1: Returning the probability of the |1> state for a single qubit:

```
fidelity = q.computeMeasurementProbability('1')
```

Case 2: Returning the probability of the |1> and |2> states for a single qutrit:

```
fidelity = q['DensityMatrixDiags'][1]
leakage = q['DensityMatrixDiags'][2]
```

Case 3: Fidelity Calculation using Density Matrices

In this case, we can provide a target density matrix for the system (both the real and imaginary part) and calculate the fidelity against that matrix. Here we outline the fidelity calculation for an X-Gate on a 2-qubit system.

```
# Expected density matrix: rho = |10><10| for an X gate on the first qubit.
expectedDmReal = np.array([
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 0
], dtype = np.float64)

expectedDmImag = np.zeros(16)

# Add target density matrix info to the buffer before execution
q.addExtraInfo("target-dm-real", expectedDmReal)
q.addExtraInfo("target-dm-imag", expectedDmImag)

# Execute the program
qpu.execute(q, prog)

# Return the fidelity
fidelity = q["fidelity"]
print("\nFidelity: {}".format(fidelity))
```

Case 4: Quantum Process Tomography:

The final method is to run XACC's Quantum Process Tomography algorithm on the system. In simulation, this method will take more time than the others listed above, but on actual hardware the difference in time will be marginal. The fidelity here is calculated against a user-provided target process matrix.

```
# This line should replace the previous provider.createComposite('pulse') call
prog = provider.createComposite('pulse_qpt')

# Create the Quantum Process Tomography Algorithm
qpt = xacc.getAlgorithm('qpt', {'circuit': prog, 'accelerator': qpu, 'optimize-circuit':␣
↪False})

# Allocate qubit and execute
q = xacc.qalloc(1)
qpt.execute(q)

# Target chi matrix (X-gate)
chi_real_vec = [0., 0., 0., 0.,
                0., 2., 0., 0.,
                0., 0., 0., 0.,
                0., 0., 0., 0.]
fidelity = qpt.calculate('fidelity', q, {'chi-theoretical-real': chi_real_vec})
```

Case 4A: Quantum Process Tomography from the Gate-Level:

Instead of calcualting the target process matrix by hand, we can leverage XACC's Pulse-Level IR Transformation to convert a user-provided gate into its corresponding chi-matrix.

```
prog = provider.createComposite('pulse_qpt')

# Get Quantum Process Tomography Algo
qpt = xacc.getAlgorithm('qpt')

# Compute Theoretical Chi Matrix
q = xacc.qalloc(1)
qpu = xacc.getAccelerator('q', {'shots': 10000})
compiler = xacc.getCompiler('xasm')
# Getting IR for an X gate
ir = compiler.compile('''__qpu__ void f(qbit q) {X(q[0]);}''', None)
qppCompositeInstr = ir.getComposites()[0]
qpt.initialize({'circuit': qppCompositeInstr, 'accelerator': qpu})

# Execute the algorithm and return real and imaginary parts of process matrix
qpt.execute(q)
chi_real_vec = q["chi-real"]
chi_imag_vec = q["chi-imag"]
```

4.6.1.4 Optimizing Controls for Quantum Systems

Using XACC's IR Transformation, similarly to in Returning the Fidelity/Case 4A, users can pass a Gate-Level instruction to the backend and return an optimized pulse with the algorithm of their choosing. The following is a short code snippet using GRAPE to construct a CNOT on a two-qubit system (for the full example, see QuaC/xacc_examples/python/ir_transform_grape_cnot).

```
# Assuming users have already defined the Hamiltonian, pulse system model,
# qpu = xacc.getAccelerator(), and the channelConfigs parameters

# Get the XASM compiler
xasmCompiler = xacc.getCompiler('xasm');

# Composite to be transformed to pulse
ir = xasmCompiler.compile('''__qpu__ void f(qbit q) {CNOT(q[0], q[1]);}''', qpu);
program = ir.getComposites()[0]

# Run the pulse IRTransformation
optimizer = xacc.getIRTransformation('quantum-control')
optimizer.apply(program, qpu, {
    'method': 'GRAPE',
    'max-time': T,
    'dt': channelConfigs.dt
})

# Run the simulation of the optimized pulse program
q = xacc.qalloc(2)
qpu.execute(q, program)
print(q)
```

After calling qpu.execute(), the *program* variable is no longer a gate, but is now the optimized pulse. Similarly, here is how to optimize an X-gate on a single qubit using GOAT:

```
# Assuming users have already defined the Hamiltonian, pulse system model,
# qpu = xacc.getAccelerator(), and the channelConfigs parameters

# Get the XASM compiler
xasmCompiler = xacc.getCompiler('xasm');

# Composite to be transform to pulse
ir = xasmCompiler.compile('''__qpu__ void f(qbit q) {Rx(q[0], 1.57);}''', qpu);
program = ir.getComposites()[0]

# Run the pulse IRTransformation
optimizer = xacc.getIRTransformation('quantum-control')
optimizer.apply(program, qpu, {
    'method': 'GOAT',
    'control-params': ['sigma'],
    # Gaussian pulse
    'control-funcs': ['exp(-t^2/(2*sigma^2))'],
    # Initial params
    'initial-parameters': [8.0],
    'max-time': 100.0
})
```

See Advanced/Pulse-evel Programming/Pulse-level IR Transformation for a more comprehensive list of each optimization method and its corresponding parameters.

4.6.1.5 Alternative Hamiltonian Declaration

Currently, XACC provides a default two-qubit backend represented by the following Hamiltonian:

```
"""
{
    "description": "Two-qubit Hamiltonian",
    "h_str": ["_SUM[i,0,1,wq{i}*O{i}]", "_SUM[i,0,1,delta{i}*O{i}*(O{i}-I{i})]", "_SUM[i,
↪0,1,omegad{i}*X{i}||D{i}]", "omegad1*X0||U0", "omegad0*X1||U1", "jq0q1*Sp0*Sm1",
↪"jq0q1*Sm0*Sp1"],
    "osc": {},
    "qub": {
        "0": 2,
        "1": 2
    },
    "vars": {
        "wq0": 30.518812656662774,
        "wq1": 31.238229295532093,
        "delta0": -2.011875935,
        "delta1": -2.008734343,
        "omegad0": -1.703999855,
        "omegad1": -1.703999855,
        "jq0q1": 0.011749557
    }
}
"""
```

Accessing this backend is as simple as:

```
qpu = xacc.getAccelerator('QuaC:Default2Q')
```

Additionally, this backend comes with the following pre-calibrated pulses:

Pulse Type	Gate Operation	Description
Single Qubit	X-Gate	pi/2 rotation over X-axis on Q0 or Q1
	H-Gate	Hadamard Gate on Q0 or Q1
	U3-Gate	U3 Operation on Q0 or Q1
Double Qubit	CNOT	CNOT with U3 gates to correct local rotation errors

[1] Data-driven gradient algorithm for high-precision quantum control

[2] Chopped random-basis quantum optimization

[3] Control of Photochemical Branching: Novel Procedures for Finding Optimal Pulses and Global Upper Bounds

[4] Tunable, Flexible, and Efficient Optimization of Control Pulses for Practical Qubits

[5] Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits

[6] Quantum optimal control via gradient ascent in function space and the time-bandwidth quantum speed limit

[7] Prolate spheroidal wave functions, fourier analysis, and uncertainty — V: the discrete case

CHAPTER

FIVE

PUBLICATIONS

The following publications describe XACC or experiments leveraging the it.

[1] XACC: A System-Level Software Infrastructure for Heterogeneous Quantum-Classical Computing

[2] A language and hardware independent approach to quantum-classical computing

[3] Validating Quantum-Classical Programming Models with Tensor Network Simulations

[4] Hybrid Programming for Near-term Quantum Computing Systems

[5] Cloud Quantum Computing of an Atomic Nucleus

[6] Quantum-Classical Computations of Schwinger Model Dynamics using Quantum Computers

5.1 Indices and tables

- genindex
- modindex
- search