Overview and Comparison of Gate Level Quantum Software Platforms

Ryan LaRose[*]

Department of Computational Mathematics, Science, and Engineering, Michigan State University.

(Dated: June 22, 2018)

Quantum computers are available to use over the cloud, but the recent explosion of quantum software platforms can be overwhelming for those deciding on which to use. In this paper, we provide a current picture of the rapidly evolving quantum computing landscape by comparing four software platforms—Forest (pyQuil), QISKit, ProjectQ, and the Quantum Developer Kit—that enable researchers to use real and simulated quantum devices. Our analysis covers requirements and installation, language syntax through example programs, library support, and quantum simulator capabilities for each platform. For platforms that have quantum computer support, we compare hardware, quantum assembly languages, and quantum compilers. We conclude by covering features of each and briefly mentioning other quantum computing software packages.

CONTENTS

I. INTRODUCTION

Quantum programming languages have been thought of at least two decades ago [1–3], but these were largely theoretical and without existing hardware. Quantum computers are now a reality, and there are real quantum programming languages that let anyone with internet access use them. A critical mass of effort from researchers in industry and academia alike has produced small quantum devices that operate on the circuit model of quantum computing. These computers are small, noisy, and not nearly as powerful as current classical computers. But they are nascent, steadily growing, and heralding a future of unimaginably large computational power for problems in chemistry [4, 5], machine learning [6, 7], optimization [8], finance [9], and more [10]. These devices are a testbed for preparing the next generation of quantum software engineers to tackle current classically intractable computational problems. Indeed, cloud quantum computing has already been used to calculate the deuteron binding energy [11] and test subroutines in machine learning algorithms [12, 13].

Recently, there has been an explosion of quantum computing software over a wide range of classical computing languages. A list of open-source projects, numbering well over fifty, is available at [14], and a list of quantum computer simulators is available at [15]. This sheer number of programs, while positively reflecting the growth of the field, makes it difficult for students and researchers to decide on which software package to use, getting lost in documentation or being too overwhelmed to know where to start.

In this paper, we hope to provide a succinct overview and comparison of major general-purpose gate-level quantum computing software platforms. From the long list, we have selected four in total: three that provide the user with the ability to connect to real quantum devices—pyQuil from Rigetti [16], QISKit from IBM [17], and ProjectQ from ETH Zurich [18, 19]—and one with similar functionality but no current capability to connect to a quantum computer—the Quantum Development Kit from Microsoft [20]. The ability to connect to a real quantum device has guided our selection of these platforms. Because of this, and for the sake of succinctness, we are intentionally omitting a number of respectable programs. We briefly mention a few of these in Appendix A.

For now, our major goal is to provide a picture of the quantum computing landscape governed by these four platforms. In Section II, we cover each platform in turn, discussing requirements and installation, documentation and tutorials, language syntax, and quantum hardware.

[*] Also at Department of Physics and Astronomy, Michigan State University; laroser1@msu.edu

In Section III, we provide a detailed comparison of the platforms. This includes quantum algorithm library support in III A, quantum hardware support in III B, quantum circuit compilers in III C, and quantum computer simulators in III D. We conclude in Section IV with discussion and some subjective remarks about each platform. Appendix A briefly mentions other quantum software, Appendix B includes details on testing the quantum circuit simulators, and Appendix C shows code for the quantum teleportation circuit in each of the four languages for a side by side comparison.

II. THE SOFTWARE PLATFORMS

An overview of various quantum computers and the software needed to connect to them is shown in Figure 1. As it currently stands, these four software platforms allow one to connect to four different quantum computers—one by Rigetti, an 8 qubit quantum computer which can be connected to via pyQuil [41]; and three by IBM, the largest openly available being 16 qubits, which can be connected to via QISKit or ProjectQ. There is also a fourth 20 qubit quantum computer by IBM, but this device is only available to members of the IBM Q Network [42], a collection of companies, universities, and national laboratories interested in and investing in quantum computing. Also shown in Figure 1 are quantum computers by companies like Google, IBM, and Intel which have been announced but are not currently available to general users.

The technology of quantum hardware is rapidly changing. It is very likely that new computers will be available by the end of the year, and in two or three years this list may be completely outdated. What will remain, however, is the software used for connecting to this technology. It will be very simple to use these new quantum computers by changing just a few lines of code without changing the actual syntax used for generating or running the quantum circuit. For example, in QISKit, one would just need to change the name of the backend when executing the circuit:

```
execute(quantum_circuit, backend="name", ...)
```

Listing 1: The string "name" specifies the backend to run quantum programs using QISKit. As future quantum computers get released, running on new hardware will be as easy as changing the name.

Although the software is changing as well with new version releases [43], these are, for the most part, relatively minor syntactical changes that do not alter significantly the software functionality.

In this section, we run through each of the four platforms in turn, discussing requirements and installation, documentation and tutorials, language syntax, quantum language, quantum hardware, and simulator capabilities. Our discussion is not meant to serve as complete instruction in a language, but rather to give the reader a feel of each platform before diving into one (or more) of his/her choosing. Our analysis includes enough information to begin running algorithms on quantum computers. However, we refer the reader, once s/he has decided on a particular platform, to the specific documentation for complete information. We include links to documentation and tutorial sources for each package. We are also assuming basic familiarity with quantum computing, for which many good references now exist [21, 22].

All of the code snippets and programs included in this paper were tested and run on a Dell XPS 13 Developer Edition laptop running Linux Ubuntu 16.04 LTS, the complete specs of which are listed in [23]. Although all software packages work on all three major operating systems, in the author's experience it is notably easier to install and use the software on the platform it was developed on. In a Linux Ubuntu environment, no difficulties nor exotic error messages were encountered when installing these software packages.

A. pyQuil

pyQuil is an open-source Python library developed by Rigetti for constructing, analyzing, and running quantum programs. It is built on top of Quil, an open quantum instruction language (or simply *quantum language*), designed specifically for near-term quantum computers and based on a shared classical/quantum memory model [24] (meaning that both qubits and classical bits are available for memory). pyQuil is one of, and the main, developer libraries in Forest, which is the overarching platform for all of Rigetti's quantum software. Forest also includes Grove and the Reference QVM, to be discussed shortly.

a. Requirements and Installation To install and use pyQuil, Python 2 or 3 is required, though Python 3 is strongly recommended as future feature developments may only support Python 3. Additionally, the Anaconda python distribution is recommended for various python module dependencies, although it is not required.

The easiest way to install pyQuil is using the Python package manager pip. At a command line on Linux Ubuntu, we type

```
pip install pyquil
```

to successfully install the software. Alternatively, if Anaconda is installed, pyQuil can be installed by typing

```
conda install -c rigetti pyquil
```

at a command line. Another alternative is to download the source code from the git repository and install the software this way. To do so, one would type the following commands:

```
git clone https://github.com/rigetticomputing/
    pyquil
cd pyquil
pip install -e .
```

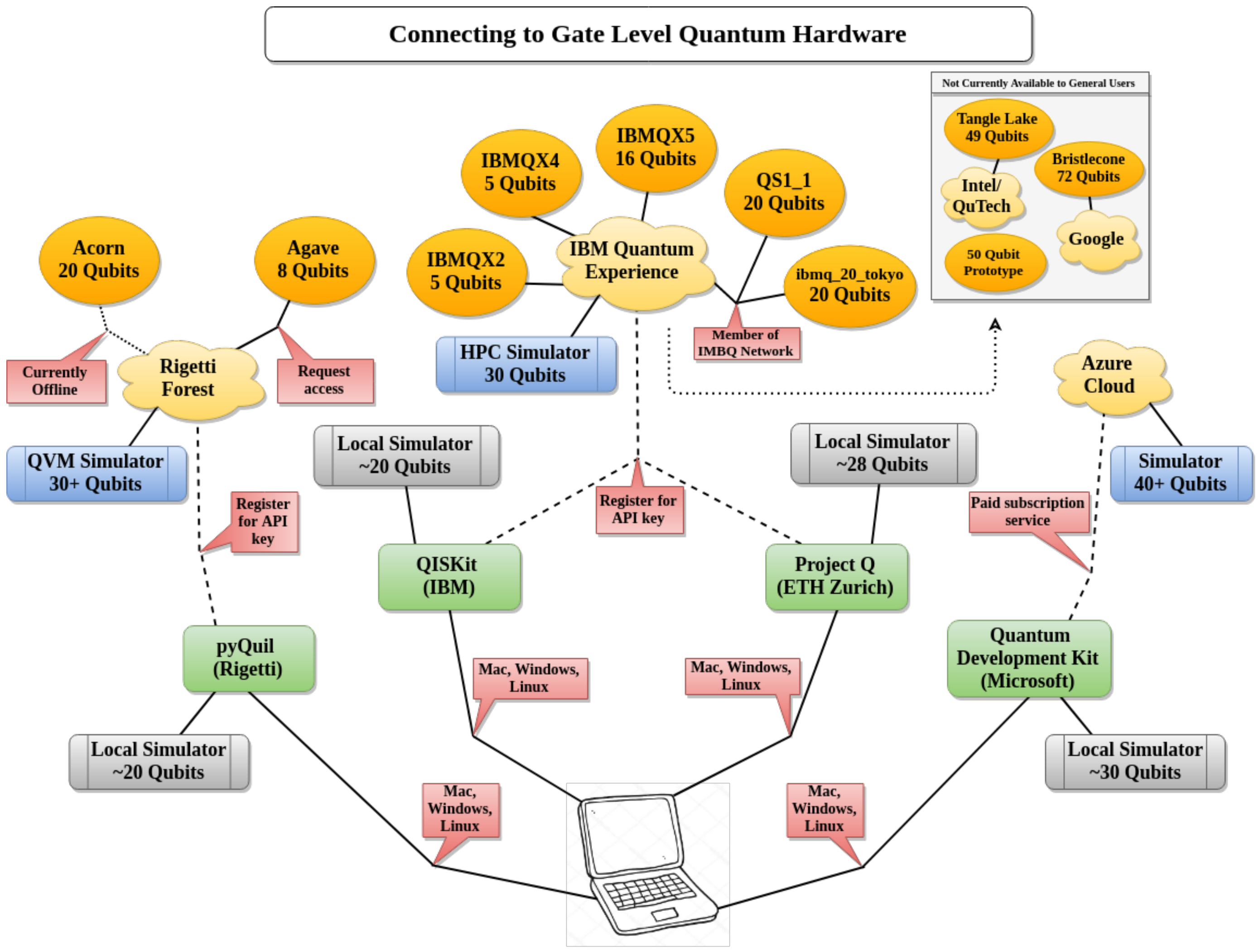

FIG. 1: A schematic diagram showing the paths to connecting a personal computer to a usable gate-level quantum computer. Starting from the personal computer (bottom center), nodes in green shows software that can be installed on the user's personal computer. Grey nodes show simulators run locally (i.e., on the user's computer). Dashed lines show API/cloud connections to company resources shown in yellow clouds. Quantum simulators and usable quantum computers provided by these cloud resources are shown in blue and gold, respectively. Red boxes show requirements along the way. For example, to connect to Rigetti Forest and use the Agave 8 qubit quantum computer, one must download and install pyQuil (available on macOS, Windows, and Linux), register on Rigetti's website to get an API key, then request access to the device via an online form. Notes: (i) Rigetti's Quantum Virtual Machine requires an upgrade for more than 30 qubits, (ii) local simulators depend on the user's computer so numbers given are approximates, and (iii) the grey box shows quantum computers that have been announced but are not currently available to general users.

This last method is recommended for any users who may wish to contribute to pyQuil. See the contribution guidelines on GitHub for more information.

b. Documentation and Tutorials pyQuil has excellent documentation hosted online with background information in quantum computing, instructions on installation, basic programs and gate operations, the simulator known as the quantum virtual machine (QVM), the actual quantum computer, and the Quil language and compiler. By downloading the source code of pyQuil from GitHub, one also gets an examples folder with Jupyter notebook tutorials, regular Python tutorials, and a program run_quil.py which can run text documents written in Quil using the quantum virtual machine. Last, we mention Grove, a collection of quantum algorithms built using pyQuil and the Rigetti Forest environment.

c. Syntax The syntax of pyQuil is very clean and efficient. The main element for writing quantum circuits is Program and can be imported from pyquil.quil. Gate operations can be found in pyquil.gates. The api module allows one to run quantum circuits on the virtual machine. One nice feature of pyQuil is that qubit registers and classical registers do not need to be defined a priori but can be rather allocated dynamically. Qubits in

pyQuil Overview

Institution	Rigetti
First Release	v0.0.2 on Jan 15, 2017
Current Version	v1.9.0 on June 6, 2018
Open Source?	✓
License	Apache-2.0
Homepage	Home
GitHub	Git
Documentation	Docs, Tutorials (Grove)
OS	Mac, Windows, Linux
Requirements	Python 3, Anaconda (recommended)
Classical Language	Python
Quantum Language	Quil
Quantum Hardware	8 qubits
Simulator	∼20 qubits locally, 26 qubits with most API keys to QVM, 30+ w/ private access
Features	Generate Quil code, example algorithms in Grove, topology-specific compiler, noise capabilities in simulator, community Slack channel

the qubit register are referred to by index (0, 1, 2, ...) and similarly for bits in the classical register. A random generator circuit can thus be written as follows:

```
1  # random number generator circuit in pyQuil
2  from pyquil.quil import Program
3  import pyquil.gates as gates
4  from pyquil import api
5
6  qprog = Program()
7  qprog += [gates.H(0),
8            gates.MEASURE(0, 0)]
9
10 qvm = api.QVMConnection()
11 print(qvm.run(qprog))
```

Listing 2: pyQuil code for a random number generator.

In the first three lines, we import the bare minimum needed to declare a quantum circuit/program (line 2), to perform gate operations on qubits (line 3) [44], and to execute the circuit (line 4). In line 6 we instantiate a quantum program, and in lines 7-8 we give it a list of instructions: first do the Hadamard gate H to the qubit indexed by 0, then measure the same qubit into a classical bit indexed by 0. In line 10 we establish a connection to the QVM, and in line 11 we run and display the output of our circuit. This program prints out, as is standard for pyQuil output, a list of lists of integers: in our case, either [[0]] or [[1]]. In general, the number of elements in the outer list is the number of trials performed. The integers in the inner lists are the final measurements into the classical register. Since we only did one trial (this is specified as an argument to api.QVMConnection.run, which is set as default to one), we only get one inner list. Since we only had one bit in the classical register, we only get one integer.

d. Quantum Language Quil is the quantum instruction language, or simply quantum language, that feeds quantum computers instructions. It is analogous to assembly language on classical computers. The general syntax of Quil is GATE index where GATE is the quantum gate to be applied to the qubit indexed by index (0, 1, 2, ...). pyQuil has a feature for generating Quil code from a given program. For instance, in the above quantum random number generator, we could add the line

```
print(qprog)
```

at the end to produce the Quil code for the circuit, which is shown below:

```
H 0
MEASURE 0 [0]
```

Listing 3: Quil code for a random number generator.

It is possible, if one becomes fluent in Quil, to write quantum circuits in a text editor in Quil and then execute the circuit on the QVM using the program run_quil.py. One could also modify run_quil.py to allow circuit execution on the QPU. We remark that the pyQuil compiler (also referred to as the Quil compiler in documentation) converts a given circuit into Quil code that the actual quantum computer can understand. We will discuss this more in Section III C.

e. Quantum Hardware Rigetti has a quantum processor that can be used by those who request access. To request access, one must visit the Rigetti website and provide a full name, email address, organization name, and description of the reason for QPU access. Once this is done, a company representative will reach out via email to schedule a time to grant the user QPU access. An advantage of this scheduling process, as opposed to the queue system of QISKit to be discussed shortly, is that many jobs can be run in the alloted time frame with deterministic runtimes, which is key for variational and hybrid algorithms. These types of algorithms send data back and forth between classical and quantum computers—having to wait in a queue makes this process significantly longer. A (perhaps) disadvantage is that jobs cannot be executed anytime when the QPU is available, but a specific time must be requested and granted.

In the author's experience, the staff is very helpful and the process is generally efficient. The actual device, the topology of which is shown in Figure 2, consists of 8 qubits with nearest neighbor connectivity. We will discuss this computer more in detail in Section III B.

f. Simulator The quantum virtual machine (QVM) is the main utility used to execute quantum circuits. It is a program written to run on a classical CPU that inputs Quil code and simulates the evolution of an actual quantum computer. To connect to the QVM, one must register for an API key for free on https://www.rigetti.com/forest by providing a name and email address. An email is then sent containing an API key and a user ID which must be set up by running

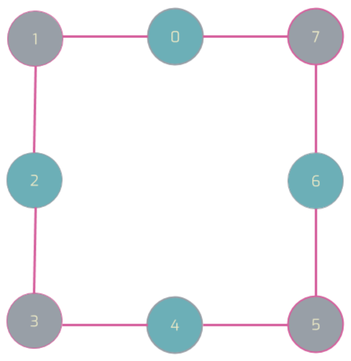

FIG. 2: Schematic diagram showing the topology (connectivity) of the 8 qubit Agave QPU by Rigetti. Qubits are labeled with integers 0, 1, ..., 7, and lines connecting qubits indicate that a two qubit gate can be performed between these qubits. For example, we can do Controlled-Z between qubits 0 and 1, but not between 0 and 2. To do the latter, the Quil compiler converts Controlled-Z (0, 2) into operations the QPU can perform. This diagram was taken from pyQuil's documentation.

```
pyquil-config-setup
```

at the command line (after installing pyQuil, of course). A prompt then appears to enter the emailed keys.

According to the documentation, most API keys give access to the QVM with up to 30 qubits, and access to more qubits can be requested. The author's API key gives access to 26 qubits (no upgrades were requested).

Additionally, the Forest library contains a local simulator written in Python and open-sourced, known as the Reference QVM. It is not as performant as the QVM, but users can run circuits with as many qubits as they have memory for on their local machines. As a general rule of thumb, circuits with qubits numbering in the low 20s are possible on commodity hardware. The reference QVM must be installed separately, which can be done with pip according to:

```
pip install referenceqvm
```

To use the Reference QVM instead of the QVM, one simply imports api from referenceqvm instead of from pyQuil:

```
import referenceapi.api as api
```

B. QISKit

The Quantum Information Software Kit, or QISKit, is an open-source software development kit (SDK) for working with the OpenQASM quantum language and quantum processors in the IBM Q experience. It is available in Python, JavaScript, and Swift, but here we only discuss the Python version.

a. Requirements and Installation QISKit is available on macOS, Windows, and Linux. To install QISKit, Python 3.5+ is required. Additional helpful, but not required, components are Jupyter notebooks for tutorials

QISKit Overview

Institution	IBM
First Release	0.1 on March 7, 2017
Current Version	0.5.4 on June 11, 2018
Open Source?	✓
License	Apache-2.0
Homepage	Home
Github	Git
Documentation	Docs, Tutorial Notebooks, Hardware
OS	Mac, Windows, Linux
Requirements	Python 3.5+, Jupyter Notebooks (for tutorials), Anaconda 3 (recommended)
Classical Language	Python
Quantum Language	OpenQASM
Quantum Hardware	IBMQX2 (5 qubits), IBMQX4 (5 qubits), IBMQX5 (16 qubits), QS1_1 (20 qubits)
Simulator	∼25 qubits locally, 30 through cloud
Features	Generate QASM code, topology specific compiler, community Slack channel, circuit drawer, ACQUA library

and the Anaconda 3 Python distribution, which comes with all the necessary dependencies pre-installed.

The easiest way to install QISKit is by using the Python package manager pip. At a command line, we install the software by typing:

```
pip install qiskit
```

Note that pip automatically handles all dependencies and will always install the latest version. Users who may be interested in contributing to QISKit can install the source code by entering the following at a command line, assuming git is installed:

```
git clone https://github.com/QISKit/qiskit-core
cd qiskit-core
python -m pip install -e .
```

For information on contributing, see the contribution guidelines in the online documentation on GitHub.

b. Documentation and Tutorials The documentation of QISKit can be found online at https://qiskit.org/documentation/. This contains instructions on installation and setup, example programs and connecting to real quantum devices, project organization, QISKit overview, and developer documentation. Background information on quantum computing can also be found for users who are new to the field. A very nice resource is the SDK reference where users can find information on the source code documentation.

QISKit also contains a large number of tutorial notebooks in a separate GitHub repository (similar to pyQuil and Grove). These include entangled states; standard algorithms like Deutsch-Josza, Grover's algorithm, phase estimation, and the quantum Fourier transform; more advanced algorithms like the variational quantum eigensolver and applications to fermionic Hamiltonians; and even some fun games like quantum battleships. Additionally, the ACQUA library (Algorithms and Circuits for QUantum Applications) contains cross-domain algorithms for chemistry and artificial intelligence with numerous examples.

There is also very detailed documentation for each of the four quantum backends containing information on connectivity, coherence times, and gate application time. Lastly, we mention the IBM Q experience website and user guides. The website contains a graphical quantum circuit interface where users can drag and drop gates onto the circuit, which is useful for learning about quantum circuits. The user guides contain more instruction on quantum computing and the QISKit language.

c. Syntax The syntax for QISKit can be seen in the following example program. In contrast to pyQuil, one has to explicitly allocate quantum and classical registers. We show below the program for the random number circuit in QISKit:

```
1  # random number generator circuit in QISKit
2  from qiskit import QuantumRegister,
       ClassicalRegister, QuantumCircuit, execute
3
4  qreg = QuantumRegister(1)
5  creg = ClassicalRegister(1)
6  qcircuit = QuantumCircuit(qreg, creg)
7
8  qcircuit.h(qreg[0])
9  qcircuit.measure(qreg[0], creg[0])
10
11 result = execute(qcircuit, 'local_qasm_simulator
       ').result()
12 print(result.get_counts())
```

Listing 4: QISKit code for a random number generator.

In line 2 we import the tools to create quantum and classical registers, a quantum circuit, and a function to execute that circuit. We then create a quantum register with one qubit (line 4), classical register with one bit (line 5), and a quantum circuit with both of these registers (line 6). Now that we have a circuit created, we begin providing instructions: in line 8, we do a Hadamard gate to the zeroth qubit in our quantum register (which is the only qubit in the quantum register); in line 9, we measure this qubit into the classical bit indexed by zero in our classical register (which is the only bit in the classical register) [45]. Now that we have built a quantum circuit, we execute it in line 11 and print out the result in line 12. By printing `result.get_counts()`, we print the "counts" of the circuit—that is, a dictionary of outputs and how many times we received each output. For our case, the only possible outputs are 0 or 1, and a sample output of the above program is {'0': 532, '1': 492}, indicating that we got 532 instances of 0 and 492 instances of 1. (The default number of times to run the circuit, called `shots` in QISKit, is 1024.)

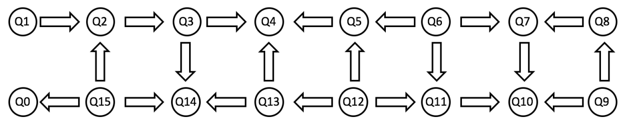

FIG. 3: A schematic diagram showing the topology of IBMQX5, taken from [30]. Directional arrows show entanglement capabilities. For example, we could perform the operation (in QASM) `cx Q1, Q2` but not the operation `cx Q2, Q1`. To do the latter, a compiler translates the instruction into equivalent gates that are performable in the topology and gate set.

d. Quantum Language OpenQASM (open quantum assembly language [25], which we may refer to simply as QASM) is the quantum language that provides instruction to the actual quantum devices, analogous to assembly language on classical computers. The general syntax of QASM is `gate qubit` where `gate` specifies a quantum gate operation and `qubit` labels a qubit. QISKit has a feature for generating QASM code from a circuit. In the above random number circuit example, we could add the line

```
print(qcircuit.qasm())
```

at the end to produce the QASM code for the circuit, shown below:

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q0[1];
4 creg c0[1];
5 h q0[0];
6 measure q0[0] -> c0[0];
```

Listing 5: OpenQASM code for a random number generator.

The first two lines are included in every QASM file. Line 3 (4) creates a quantum (classical) register, and lines 5 and 6 give the instructions for the circuit. It is possible to write small circuits like this directly in OpenQASM, but for larger circuits it is nice to have the tools in QISKit to cleanly and efficiently program quantum computers.

e. Quantum Hardware There is a vast amount of documentation for the quantum backends supported by QISKit. These devices include IBMQX2 (5 qubits), IBMQX4 (5 qubits), IBMQX5 (16 qubits), and QS1_1 (20 qubits, usable only by members of the IBM Q network). Documentation for each is available on GitHub. We discuss in detail IBMQX5 in Section III B, the topology of which is shown in Figure 3.

f. Simulator IBM includes several quantum circuit simulators that run locally or on cloud computing resources. These simulators include a local unitary simulator—which applies the entire unitary matrix of the circuit and is limited practically to about 12 qubits—and a state vector simulator—which performs the best locally and can simulate circuits of up to 25 qubits. For now we just quote qubit number, but we discuss the performance of the state vector simulator and compare it to other simulators in Section III D.

C. ProjectQ

ProjectQ is an open source software framework for quantum computing featuring connectivity to IBM's quantum backends, a high performance quantum computer simulator, and several library plug-ins. The first release of ProjectQ was developed by Thomas Häner and Damien S. Steiger in the group of Matthias Troyer at ETH Zürich, and it has since picked up more contributors.

ProjectQ Overview

Institution	ETH Zurich
First Release	v0.1.0 on Jan 3, 2017
Current Version	v0.3.6 on Feb 6, 2018
Open Source?	✓
License	Apache-2.0
Homepage	Home
Github	Git
Documentation	Docs, Example Programs, Paper
OS	Mac, Windows, Linux
Requirements	Python 2 or 3
Classical Language	Python
Quantum Language	none/hybrid
Quantum Hardware	no dedicated hardware, can connect to IBM backends
Simulator	∼28 qubits locally
Features	Draw circuits, connect to IBM backends, multiple library plug-ins

a. Requirements and Installation A current version of Python, either 2.7 or 3.4+, is required to install ProjectQ. The documentation contains detailed information on installation for each operating system. In our environment, we do the recommended pip install

```
python -m pip install --user projectq
```

to successfully install the software (as a user). To install via the source code, we can run the following at a command line:

```
git clone https://github.com/ProjectQ-Framework/
    ProjectQ
cd projectq
python -m pip install --user .
```

As with previous programs, this method is recommended for users who may want to contribute to the source code. For instructions on doing so, see the contribution guidelines on the ProjectQ GitHub page.

b. Documentation and Tutorials ProjectQ has very good documentation on installation. However, we find the remaining documentation to be a little sparse. The online tutorial provides instruction on basic syntax and example quantum programs (random numbers, teleportation, and Shor's factoring algorithm). The rest is the code documentation/reference with information on the structure of the code and each additional module, including functions and classes. The papers [18, 19] are a good reference and resource, but it is more likely that the online documentation will be more up to date.

c. Syntax The syntax of ProjectQ is clear and efficient, though it may take some getting used to. There is no quantum assembly language for ProjectQ (because there is no ProjectQ specific quantum backend), but the classical language is sort of a hybrid classical/quantum language. To elaborate, an example program to produce a random bit is shown below:

```
1  # random number generator circuit in ProjectQ
2  from projectq import MainEngine
3  import projectq.ops as ops
4
5  eng = MainEngine()
6  qbits = eng.allocate_qureg(1)
7
8  ops.H | qbits[0]
9  ops.Measure | qbits[0]
10
11 eng.flush()
12 print(int(qbits[0]))
```

Listing 6: ProjectQ code for a random number generator.

In line 2, we import the necessary module to make a quantum circuit, and in line 3 we import gate operations. In line 5 we allocate and engine from the MainEngine, and in line 6 we allocate a one qubit register. In lines 8 and 9 we provide the circuit instructions: first do a Hadamard gate on the qubit in the register indexed with a 0, then measure this qubit. This is where the "quantum syntax" appears within the classically scripted language. The general formulation is operation | qubit with the vertical line between the two resemblant of Dirac notation, $H|0\rangle$, for example. We then flush the engine which pushes it to a backend and ensures it gets evaluated/simulated. Unlike pyQuil and QISKit, in ProjectQ one does not specify a classical register when making a measurement. Instead, when we measure qbits[0] in line 9, we get it's value by converting it to an int when we print it out in line 12.

d. Quantum Language As mentioned, ProjectQ does not have its own dedicated quantum language. If one is using ProjectQ in conjunction with an IBM backend, the code will eventually get converted to OpenQASM, IBM's quantum assembly language.

e. Quantum Hardware ProjectQ does not have its own dedicated quantum computer. One is able to use IBM's quantum backends when using ProjectQ, however.

f. Simulator ProjectQ comes with a fast simulator written in C++, which will be installed by default unless an error occurs, in which case a slower Python simulator will be installed. Additionally, ProjectQ includes a ClassicalSimulator for efficiently simulating stabilizer circuits—i.e., circuits that consist of gates from the normalizer of the Pauli group, which can be generated from Hadmard, CNOT, and phase gates [26]. This simulator is able to handle thousands of qubits to check, e.g., Toffoli adder circuits for specific inputs. However, stabilizer circuits are not universal, so we focus our benchmark and testing on the C++ Simulator.

ProjectQ's C++ Simulator is sophisticated and fast. On the author's computer [23] (the maximum qubit number is limited by the user's local memory, as mentioned), it can handle circuits with 26 qubits of depth 5 in under a minute and circuits of 28 circuits of depth 20 in just under ten minutes. For full details, see section III D and Figure 6.

g. ProjectQ in other Platforms ProjectQ is well-tested, robust code and has been used for other platforms mentioned in this paper. Specifically, pyQuil contains ProjectQ code [27], and the kernels of Microsoft's QDK simulator are developed by Thomas Häner and Damian Steiger at ETH Zurich [28], the original authors of ProjectQ. (Note that this does not necessarily mean that the QDK simulator achieves the performance of the ProjectQ C++ simulator as the enveloping code could diminish performance.)

D. Quantum Development Kit

Unlike the superconducting qubit technology of Rigetti and IBM, Microsoft is betting highly on topological qubits based on Majorana fermions. These particles have recently been discovered [29] and promise long coherence times and other desirable properties, but no functional quantum computer using topological qubits currently exists. As such, Microsoft currently has no device that users can connect to via their Quantum Development Kit (QDK), the youngest of the four platforms featured in this paper. Nonetheless, the QDK features a new "quantum-focused" language called Q# that has strong integration with Visual Studio and Visual Studio Code and can simulate quantum circuits of up to 30 qubits locally. This pre-release software was first debuted in January of 2018 and, while still in alpha testing, is available on macOS, Windows, and Linux.

QDK Overview

Institution	Microsoft
First Release	0.1.1712.901 on Jan 4, 2018 (pre-release)
Current Version	0.2.1802.2202 on Feb 26, 2018 (pre-release)
Open Source?	✓
License	MIT
Homepage	Home
Github	Git
Documentation	Docs
OS	Mac, Windows, Linux
Requirements	Visual Studio Code (strongly recommended)
Classical Language	Q#
Quantum Language	Q#
Quantum Hardware	none
Simulator	30 qubits locally, 40 through Azure cloud
Features	Built-in algorithms, example algorithms

a. Requirements and Installation Although it is listed as optional in the documentation, installing Visual Studio Code is strongly recommended for all platforms. (In this paper, we only use VS Code, but Visual Studio is also an exceptional framework. We remain agnostic as to which is better and use VS Code as a matter of preference.) Once this is done, the current version of the QDK can be installed by entering the following at a Bash command line:

```
dotnet new -i "Microsoft.Quantum.
    ProjectTemplates::0.2-*"
```

To get QDK samples and libraries from the GitHub repository (strongly recommended for all and especially those who may wish to contribute to the QDK), one can additionally enter:

```
git clone https://github.com/Microsoft/Quantum.
    git
cd Quantum
code .
```

b. Documentation and Tutorials The above code samples and libraries are a great way to learn the Q# language, and the online documentation contains information on validating a successful install, running a first quantum program, the quantum simulator, and the Q# standard libraries and programming language. This documentation is verbose and contains a large amount of information; the reader can decide whether this is a plus or minus.

c. Syntax The syntax of Q# is rather different from the previous three languages. It closely resembles C#, is more verbose than Python, and may have a steeper learning curve for those not familiar with C#. Shown below is the same random number generator circuit that we have shown for all languages:

```
// random number generator circuit in QDK
operation random (count : Int, initial: Result)
    : (Int,Int)
    {
        body
        {
            mutable numOnes = 0;
            using (qubits = Qubit[1])
            {
                for (test in 1..count)
                {
                    Set (initial, qubits[0]);
                    H(qubits[0]);
                    let res = M (qubits[0]);

                    // count the number of ones
                    if (res == One)
                    {
                        set numOnes = numOnes+1;
                    }
                }
                Set(Zero, qubits[0]);
            }
            // return statistics
            return (count - numOnes, numOnes);
        }
    }
```

Listing 7: Q# code for a random number generator.

The use of brackets and keywords can perhaps make this language a little more difficult for new users to learn/read, but at its core the code is doing the same circuit as the previous three examples. For brevity we omit the code analysis. We will remark, however, that this is one of three files needed to run this code. Above is the .qs file for the Q# language, and we additionally need a Driver.cs file to run the .qs code as well as a .csproj containing meta-data. All in all, this example totals about 65 lines of code. Since this example can be found in the online documentation, we do not include these programs but refer the interested reader to the "Quickstart" tutorial in the documentation.

The author would like to note that the QDK is striving for a high-level language that abstracts from hardware and makes it easy for users to program quantum computers. As an analogy, one does not specifically write out the adder circuit when doing addition on a classical computer—this is done in a high level framework ($a+b$), and the software compiles this down to the hardware level. As the QDK is focused on developing such standards, measuring ease of writing code based on simple examples such as a random number generator and the teleportation circuit (see Appendix C) may not do justice to the overall language syntax and platform capabilities, but we include these programs to have some degree of consistency in our analysis.

d. Quantum Language/Hardware As mentioned, the QDK has no current capability to connect to a real quantum computer, and it does not have a dedicated quantum assembly language. The Q# language can be considered a hybrid classical/quantum language, however.

e. Simulator On the users local computer, the QDK includes a quantum simulator that can run circuits of up to 30 qubits. As mentioned above, kernels for QDK simulators were written by developers of ProjectQ, so performance can be expected to be similar to ProjectQ's simulator performance. (See Section III D.) Through a paid subscription service to Azure cloud, one can get access to high performance computing that enables simulation of more than 40 qubits. In the QDK documentation, however, there is currently little instruction on how to do this.

Additionally, the QDK provides a trace simulator that is very effective for debugging classical code that is part of a quantum program as well as estimating the resources required to run a given instance of a quantum program on a quantum computer. The trace simulator allows various performance metrics for quantum algorithms containing thousands of qubits. Circuits of this size are possible because the trace simulator executes a quantum program without actually simulating the state of a quantum computer. A broad spectrum of resource estimation is covered, including counts for Clifford gates, T-gates, arbitrarily-specified quantum operations, etc. It also allows specification of the circuit depth based on specified gate durations. Full details of the trace simulator can be found in the QDK documentation online.

III. COMPARISON

Now that the basics of each platform have been covered, in this section we compare each on additional aspects including library support, quantum hardware, and quantum compilers. We also enumerate some notable and useful features of each platform.

A. Library Support

We use the term "library support" to mean examples of quantum algorithms (in tutorial programs or in documentation) or a specific function for a quantum algorithm (e.g., language.DoQuantumFourierTransform(...)). We have already touched on some of these in the previous section. A more detailed table showing library support for the four software platforms is shown in Figure 4.

We remark that any algorithm, of course, can be implemented on any of these platforms. Here, we are highlighting existing functionality, which may be beneficial for users who are new to the field or even for experienced users who may not want to program everything themselves.

As can be seen from the table, pyQuil, QISKit, and the QDK have a relatively large library support. ProjectQ contains FermiLib, plugins for FermiLib, as well as compatibility with OpenFermion, all of which are open-source projects for quantum simulation algorithms. All examples that work with these frameworks naturally work with ProjectQ. Microsoft's QDK is notable for its number of built-in functions performing these algorithms automatically without the user having to explicitly program the quantum circuit. In particular, the QDK libraries offer detailed iterative phase estimation, an important procedure in many algorithms that can be easily realized on the QDK without sacrificing adaptivity. QISKit is notable for its large number of tutorial notebooks on a wide range of topics from fundamental quantum algorithms to didactic quantum games.

B. Quantum Hardware

In this section we discuss only pyQuil and QISKit, since these are the only platforms with their own dedicated quantum hardware. Qubit quantity is an important characterization in quantum computers, but equally important—if not more important—is the "qubit quality." By this, we mean coherence times (how long qubits live before collapsing to bits), gate application times, gate error rates, and the topology/connectivity of the qubits. Ideally, one would have infinite coherence times, zero gate application time, zero error rates, and all-to-all connectivity. In the following paragraphs we document some of

Algorithm	**pyQuil**	**QISKit**	**ProjectQ**	**QDK**
Random Number Generator	✓(T)	✓(T)	✓(T)	✓(T)
Teleportation	✓(T)	✓(T)	✓(T)	✓(T)
Swap Test	✓(T)			
Deutsch-Jozsa	✓(T)	✓(T)		✓(T)
Grover's Algorithm	✓(T)	✓(T)	✓(T)	✓(B)
Quantum Fourier Transform	✓(T)	✓(T)	✓(B)	✓(B)
Shor's Algorithm			✓(T)	✓(D)
Bernstein Vazirani	✓(T)	✓(T)		✓(T)
Phase Estimation	✓(T)	✓(T)		✓(B)
Optimization/QAOA	✓(T)	✓(T)		
Simon's Algorithm	✓(T)	✓(T)		
Variational Quantum Eigensolver	✓(T)	✓(T)	✓(P)	
Amplitude Amplification	✓(T)			✓(B)
Quantum Walk		✓(T)		
Ising Solver	✓(T)			✓(T)
Quantum Gradient Descent	✓(T)			
Five Qubit Code				✓(B)
Repetition Code		✓(T)		
Steane Code				✓(B)
Draper Adder			✓(T)	✓(D)
Beauregard Adder			✓(T)	✓(D)
Arithmetic			✓(B)	✓(D)
Fermion Transforms	✓(T)	✓(T)	✓(P)	
Trotter Simulation				✓(D)
Electronic Structure (FCI, MP2, HF, etc.)			✓(P)	
Process Tomography	✓(T)	✓(T)		✓(D)
Meyer-Penny Game	✓(D)			
Vaidman Detection Test		✓(T)		
Battleships Game		✓(T)		
Emoji Game		✓(T)		
Counterfeit Coin Game		✓(T)		

FIG. 4: A table showing the library support for each of the four software platforms. By "library support," we mean a tutorial notebook or program (T), an example in the documentation (D), a built-in function (B) to the language, or a supported plug-in library (P).

the parameters of IBMQX5 and Agave, the two largest publicly available quantum computers. For full details, please see the online documentation of each platform.

a. IBMQX5 IBMQX5 is a superconducting qubit quantum computer with nearest neighbor connectivity between its 16 qubits (see Figure 3). The minimum coherence (T2) time is 31 ± 5 microseconds on qubit 0 and the maximum is 89 ± 17 microseconds on qubit 15. A single qubit gate takes 80 nanoseconds to implement plus a 10 nanosecond buffer after each pulse. `CNOT` gates take about two to four times as long, ranging from 170 nanoseconds for `cx q[6], q[7]` to 348 nanoseconds for `cx q[3], q[14]`. Single qubit gate fidelity is very good at over 99.5% fidelity for all qubits (fidelity = 1 - error). Multi-qubit fidelity is above 94.9% for all qubit pairs in the topology. The largest readout error is rather large at about 12.4% with the average being around 6%. These statistics were obtained from [30].

Lastly, we mention that to use any available quantum computer by IBM, the user submits his/her job into a queue, which determines when the job gets run. This is in contrast to using Agave by Rigetti, in which users have to request access first via an online form, then schedule a time to get access to the device to run jobs. In the author's experience, this is done over email, and the staff is very helpful.

b. Agave The Agave quantum computer consists of 8 superconducting transmon qubits with fixed capacitive coupling and connectivity shown in Figure 2. The minimum coherence (T2) time is 9.2 microseconds on qubit 1 and the maximum is 15.52 microseconds on qubit 2. The time to implement a Controlled-Z gate is between 118 and 195 nanoseconds. Single qubit gate fidelity is at an average of 96.2% (again, fidelity = 1 - error) and minimum of 93.2%. Multi-qubit gate fidelity is on average 87% for all qubit-qubit pairs in the topology. Readout errors are unknown. These statistics can be found in the online documentation or through pyQuil.

C. Quantum Compilers

Platforms that provide connectivity to real quantum devices must necessarily have a means of translating a given circuit into operations the computer can understand. This process is known as *compilation*, or more verbosely *quantum circuit compilation/quantum compilation*. Each computer has a basis set of gates and a given connectivity—it is the compiler's job to input a given circuit and return an equivalent circuit obeying the basis set and connectivity requirements. In this section we only discuss QISKit and Rigetti, for these are the platforms with real quantum computers.

The IBMQX5 basis gates are u_1, u_2, u_3, and `CNOT` where

$$u_1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix},$$

$$u_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\lambda+\phi)} \end{bmatrix}, \qquad \text{and}$$

$$u_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix}.$$

Note that u_1 is equivalent to a frame change $R_z(\theta)$ up to a global phase and u_2 and u_3 are a sequence of frame changes and pulses $R_x(\pi/2)$

$$u_2(\phi, \lambda) = R_z(\phi + \pi/2) R_x(\pi/2) R_z(\lambda - \pi/2),$$
$$u_3(\theta, \phi, \lambda) = R_z(\phi + 3\pi) R_x(\pi/2) R_z(\theta + \pi) R_x(\pi/2) R_z(\lambda)$$

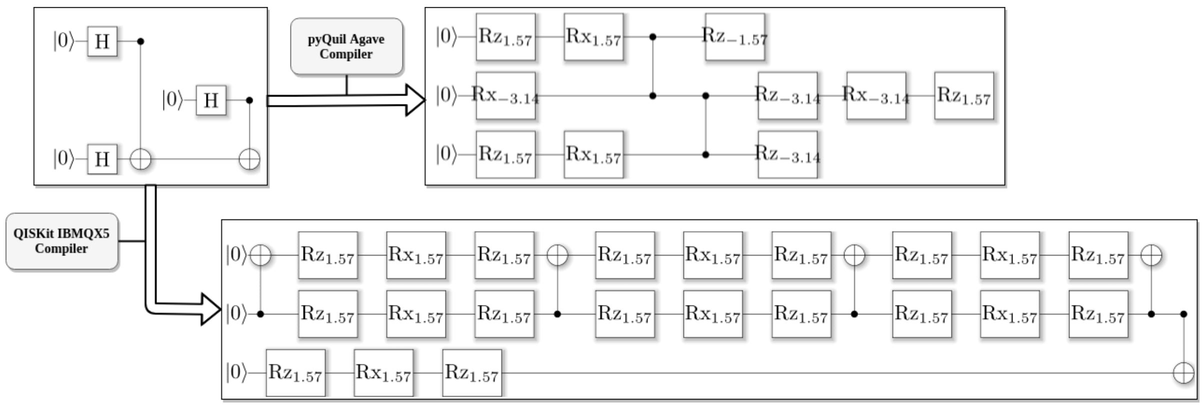

FIG. 5: An example of a quantum circuit (top left) compiled by pyQuil for Rigetti's 8 qubit Agave processor (top right), and the same circuit compiled by QISKit for IBM's 16 qubit IBMQX5. The qubits used on Agave are 0, 1, and 2 (see Figure 2), and the qubits used on IBMQX5 are 0, 1, and 2. Note that neither compiler can directly implement a Hadamard gate H but produces these via products of rotation gates R_x and R_z. A CNOT gate can be implement on IBMQX5, but not on Agave—here, pyQuil must express CNOT in terms of Controlled-Z and rotations. These circuits were made with ProjectQ.

with the rotation gates being the standard

$$R_x(\theta) := e^{-i\theta X/2} = \begin{bmatrix} \cos\theta/2 & -i\sin\theta/2 \\ -i\sin\theta/2 & \cos\theta/2 \end{bmatrix},$$

$$R_z(\theta) := e^{-i\theta Z/2} = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$$

where X and Z are the usual Pauli matrices. On the IBM quantum computers, $R_z(\theta)$ is a "virtual gate," meaning that nothing is actually done to the qubit physically. Instead, since the qubits are naturally rotating about the z-axis, doing a z rotation simply amounts to changing the clock, or frame, of the internal (classical) software keeping track of the qubit.

The topology of IBMQX5 is shown in Figure 3. This connectivity determines which qubits it is possible to natively perform CNOT gates, where a matrix representation of CNOT is given by

$$\mathsf{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Note that it is possible to perform CNOT between any qubits in QISKit, but when the program is compiled down to the hardware level, the QISKit compiler converts this into a sequence of CNOT gates allowed in the connectivity. The QISKit compiler allows one to specify an arbitrary basis gate set and topology, as well as providing a set of parameters such as noise.

For Rigetti's 8 qubit Agave processor, the basis gates are $R_x(k\pi/2)$ for $k \in \mathbb{Z}$, $R_z(\theta)$, and Controlled-Z. The single qubit rotation gates are as above, and the two qubit Controlled-Z (CZ) is given by

$$\mathsf{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

The topology of Agave is shown in Figure 2. Like QISKit, pyQuil's compiler also allows one to specify a target instruction set architecture (basis gate set and computer topology).

An example of the same quantum circuit compiled by both of these platforms is shown in Figure 5. Here, with pyQuil we compile to the Agave specifications and with QISKit we compile to the IBMQX5 specifications. As can be seen, QISKit produces a longer circuit (i.e., has greater depth) than pyQuil. It is not appropriate to claim one compiler is superior because of this example, however. Circuits that are in the language IBMQX5 understands would naturally produce a shorter depth circuit than pyQuil, and vice versa. It is known that any quantum circuit (unitary matrix) can be decomposed into a sequence of one and two qubit gates (see, e.g., [32]), but in general this takes exponentially many gates. It is currently a question of significant interest [46] to find an optimal compiler for a given topology.

D. Simulator Performance

Not all software platforms provide connectivity to real quantum computers, but any worthwhile program includes a quantum circuit simulator. This is a program that runs on a classical CPU that mimics (i.e., simulates) the evolution of a quantum computer. As with quantum hardware, it is important to look at not just how many qubits a simulator can handle but also how quickly it can process them, in addition to other parameters like adding noise to emulate quantum computers, etc. In this section, we evaluate the performance of QISKit's local state vector simulator and ProjectQ's local C++ simulator using the program listed in Appendix B. First, we mention the performance of pyQuil's QVM simulator.

a. pyQuil The Rigetti simulator, called the Quantum Virtual Machine (QVM), does not run on the users local computer but rather through computing resources in the cloud. As mentioned, this requires an API key to connect to. Most API keys give access to 30 qubits initially, and more can be requested. The author is able to simulate a 16 qubit circuit of depth 10 in 2.61 seconds on average. A circuit size of 23 qubits of depth 10 was simulated in 56.33 seconds, but no larger circuits could be simulated because the QVM terminates after one minute of processing with the author's current API access key. Because of this termination time, and because of the fact that the QVM does not run on the user's local computer, we do not test the performance of the QVM in the same way we test ProjectQ and QISKit's simulators.

The QVM contains sophisticated and flexible noise models to emulate the evolution of an actual quantum computer. This is key for developing short depth algorithms on near term quantum computers, as well as for predicting the output of a particular quantum chip. Users can define arbitrary noise models to test programs, in particular define noisy gates, add decoherence noise, and model readout noise. For full details and helpful example programs, see the Noise and Quantum Computation section of pyQuil's documentation.

b. QISKit QISKit has several quantum simulators available as backends: the local_qasm_simulator, the local_state_vector_simulator, the ibmq_qasm_simulator, the local_unitary_simulator, and the local_clifford_simulator. The differences in these simulators is the methodology of simulating quantum circuits. The unitary simulator implements basic (unitary) matrix multiplication and is limited quickly by memory. The state vector simulator does not store the full unitary matrix but only the state vector and single/multi qubit gate to apply. Both methods are discussed in [33], and [34–36] contains details on other techniques. Similar to the discussion of the ClassicalSimulator in ProjectQ, the local_clifford_simulator is able to efficiently simulate stabilizer circuits, which are not universal.

Using the local unitary simulator, a circuit of 10 qubits on depth 10 is simulated in 23.55 seconds. Adding one more qubit increases this time by approximately a factor of ten to 239.97 seconds, and at 12 qubits the simulator timed out after 1000 seconds (about 17 minutes). This simulator quickly reaches long simulation times and memory limitations because for n qubits, the unitary matrix of size $2^n \times 2^n$ has to be stored in memory.

The state vector simulator significantly outperforms the unitary simulator. We are able to simulate circuits of 25 qubits in just over three minutes. Circuits of up to 20 qubits with depth up to thirty are all simulated in under five seconds. See Figures 6 and 7 for complete details.

c. ProjectQ ProjectQ comes with a high performance C++ simulator that performed the best in our local testing. The maximum size circuit we were able to successfully simulate was 28 qubits, which took just under ten minutes (569.71 seconds) with a circuit of depth 20. For implementation details, see [18]. For the com-

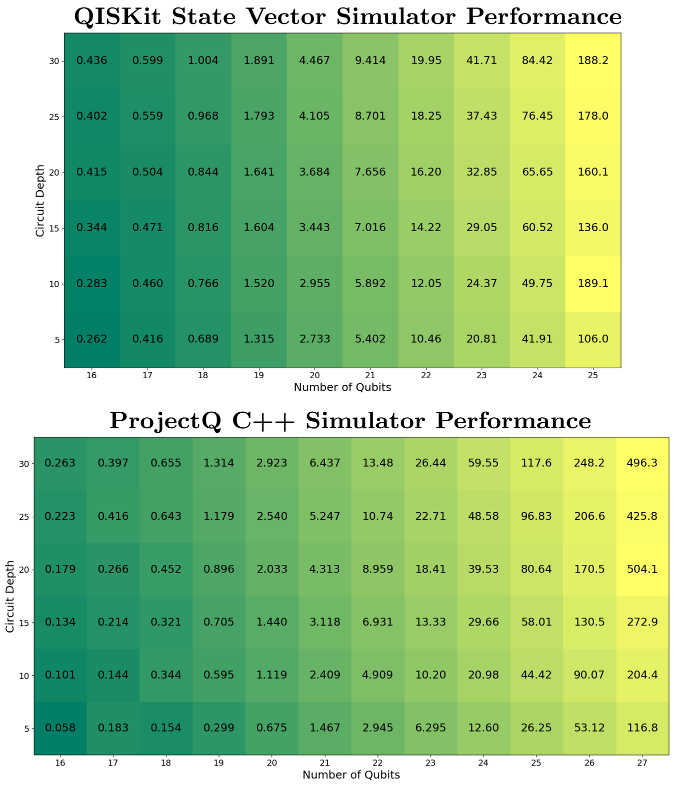

FIG. 6: Plots of the performances of QISKit's local state vector simulator (top) and ProjectQ's C++ simulator (bottom), showing runtime in seconds for a given number of qubits (horizontal axis) and circuit depth (vertical axis). Darker green shows shorter times and brighter yellow shows longer times (color scales are not the same for both plots). For details more details on the testing, see Appendix B.

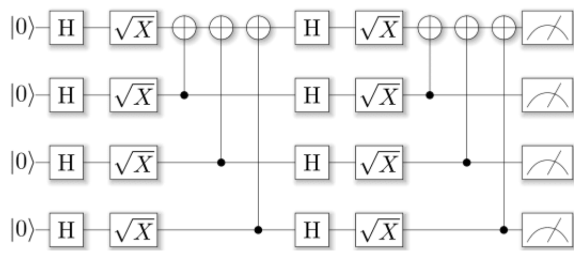

FIG. 7: The circuit used for testing the ProjectQ C++ simulator and QISKit local state vector simulator, shown here on four qubits. In the actual testing, the pattern of Hadamard gates, $\sqrt{X}$ gates, then the sequence of CNOT gates defines one level in the circuit. This pattern is repeated until the desired depth is reached. This image was produced using ProjectQ.

plete performance and testing, see Figures 6 and 7.

E. Features

A nice feature of pyQuil is Grove, a separate GitHub repository that can be installed containing tutorials and example algorithms using pyQuil. Rigetti is also building a solid community of users as exemplified by their dedicated Slack channel for Rigetti Forest. The Quil compiler and it's ability to compile for any given instruction set architecture (topology and gate basis) are also nice features. Lastly, pyQuil is compatible with OpenFermion [37], an open-source Python package for compiling and analyzing quantum algorithms to simulate fermionic systems, including quantum chemistry.

QISKit is also available in JavaScript and Swift for users who may have experience in these languages. For beginners, Python is a very good starter programming language because of its easy and intuitive syntax. Like Grove, QISKit also contains a dedicated repository of example algorithms and tutorials. Additionally, the ACQUA library in QISKit contains numerous algorithms for quantum chemistry and artificial intelligence. This library can be run through a graphical user interface or from a command line interface. IBM is second to none for building an active community of students and researchers using their platform. The company boasts of over 3 million remote executions on cloud quantum computing resources using QISKit run by more than 80,000 registered users, and there have been more than 60 research publications written using the technology [31]. QISKit also has a dedicated Slack channel with the ability to see jobs in the queue, a useful feature for determining how long a job submission will take to run. Additionally, the newest release of QISKit contains a built-in circuit drawer.

Likewise, ProjectQ contains a circuit drawer. By adding just a few lines of code to programs, one can generate TikZ code to produce high quality $\TeX$ images. All quantum circuit diagrams in this paper were made using ProjectQ. The local simulator of ProjectQ is also a great feature as it has very high performance capabilities. Although ProjectQ has no dedicated quantum hardware of its own, users are able to connect to IBM's quantum hardware. Additionally, ProjectQ has multiple library plug-ins including OpenFermion, as mentioned above.

The QDK was available exclusively on Windows until it received support on macOS and Linux in February 2018. The capability to implement quantum algorithms without explicitly programming the circuit is a nice feature of the QDK, and there are also many good tutorials in the documentation and examples folder for quantum algorithms. It is also notable that Q# provides auto-generation features for, e.g., the adjoint or controlled version of a quantum operation. In a more general sense, the QDK emphasizes and offers important tools for productive quantum algorithm development including the testing of quantum programs, estimating resource requirements, programming on different models of quantum computation targeted by different hardware, and ensuring the correctness of quantum programs at compile time. These aspects are key in moving towards high-level quantum programming languages.

IV. DISCUSSION AND CONCLUSIONS

At this point, we hope that the reader has enough information and understanding to make an informed decision of what quantum software platform(s) is (are) right for him/her. A next step is to begin reading the documentation of a platform, install it, and begin coding. In a short amount of time one can begin running algorithms on real quantum devices and begin researching/developing algorithms in their respective field.

For those who may be still undecided, we offer the following subjective suggestions:

- For those whose main objective is using quantum computers, QISKit (or ProjectQ) or pyQuil is the obvious choice.
- For those who are new to quantum computing, QISKit, pyQuil, or the QDK is a good choice.
- For those who have little programming experience, one of the Python platforms is a good choice.
- For those who are familiar with or prefer C/C# style syntax, the QDK is a good choice.
- For those wishing to develop, prototype, and test algorithms, ProjectQ is a good choice.
- For those who wish to run hybrid quantum-classical algorithms, pyQuil is a great choice for it's dedicated hardware time scheduling.
- For those who are interested in continuous variable quantum computing, see Strawberry Fields in Appendix A.

Again, these are simply suggestions and we encourage the reader to make his/her own choice. All platforms are significant achievements in the field of quantum computing and excellent utilities for students and researchers to program real quantum computers. As a final remark, we note that there are additional quantum software packages being developed, a few of which are mentioned in Appendix A.

V. ACKNOWLEDGEMENTS

We acknowledge use of the IBM Q experience for this work. The views expressed are those of the authors and do not reflect the official policy or position of IBM or the IBM Q experience team. RL thanks Doug Finke for many useful comments and edits on earlier drafts of this paper. We thank developers for feedback and helping us achieve a more accurate representation of each respective platform. Specifically, we thank Damian Steiger from ETH Zürich; Will Zeng from Rigetti; and Cathy Palmer, Julie Love, and the Microsoft Quantum Team from Microsoft. RL acknowledges support from Michigan State University through an Engineering Distinguished Fellowship.

[1] Bernhard Ömer, A procedural formalism for quantum computing, Master's thesis, Department of Theoretical Physics, Technical University of Vienna, 1998.

[2] S. Bettelli, L. Serafini, T. Calarco, Toward an architecture for quantum programming, Eur. Phys. J. D, Vol. 25, No. 2, pp. 181-200 (2003).

[3] Peter Selinger (2004), A brief survey of quantum programming languages, in: Kameyama Y., Stuckey P.J. (eds) Functional and Logic Programming. FLOPS 2004. Lecture Notes in Computer Science, vol 2998. Springer, Berlin, Heidelberg.

[4] Benjamin P. Lanyon, James D. Whitfield, Geoff G. Gillet, Michael E. Goggin, Marcelo P. Almeida, Ivan Kassal, Jacob D. Biamonte, Masoud Mohseni, Ben J. Powell, Marco Barbieri, Alán Aspuru-Guzik, Andrew G. White, Towards quantum chemistry on a quantum computer, *Nature Chemistry* **2**, pages 106-111 (2010), doi:10.1038/nchem.483.

[5] Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, Alán Aspuru-Guzik, Quantum information and computation for chemistry, NSF Workshop Report, 2017.

[6] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, Seth Lloyd, Quantum machine learning, *Nature* volume **549**, pages 195-202 (14 September 2017).

[7] Seth Lloyd, Masoud Mohseni, Patrick Rebentrost, Quantum principal component analysis, *Nature Physics* volume **10**, pages 631-633 (2014).

[8] Vadim N. Smelyanskiy, Davide Venturelli, Alejandro Perdomo-Ortiz, Sergey Knysh, and Mark I. Dykman, Quantum annealing via environment-mediated quantum diffusion, Phys. Rev. Lett. **118**, 066802, 2017.

[9] Patrick Rebentrost, Brajesh Gupt, Thomas R. Bromley, Quantum computational finance: Monte Carlo pricing of financial derivatives, arXiv preprint (arXiv:1805.00109v1), 2018.

[10] I. M. Georgescu, S. Ashhab, Franco Nori, Quantum simulation, Rev. Mod. Phys. 86, 154 (2014), DOI: 10.1103/RevModPhys.86.153.

[11] E. F. Dumitrescu, A. J. McCaskey, G. Hagen, G. R. Jansen, T. D. Morris, T. Papenbrock, R. C. Pooser, D. J. Dean, P. Lougovski, Cloud quantum computing of an atomic nucleus, Phys. Rev. Lett. **120**, 210501 (2018), DOI: 10.1103/PhysRevLett.120.210501.

[12] Lukasz Cincio, Yigit Subasi, Andrew T. Sornborger, and Patrick J. Coles, Learning the quantum algorithm for state overlap, arXiv preprint (arXiv:1803.04114v1), 2018

[13] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, et al., Quantum algorithm implementations for beginners, arXiv preprint (arXiv:1804.03719v1), 2018.

[14] Mark Fingerhuth, Open-Source Quantum Software Projects, accessed May 12, 2018.

[15] Quantiki: List of QC Simulators, accessed May 12, 2018.

[16] R. Smith, M. J. Curtis and W. J. Zeng, A practical quantum instruction set architecture, 2016.

[17] QISKit, originally authored by Luciano Bello, Jim Challenger, Andrew Cross, Ismael Faro, Jay Gambetta, Juan Gomez, Ali Javadi-Abhari, Paco Martin, Diego Moreda, Jesus Perez, Erick Winston, and Chris Wood, https://github.com/QISKit/qiskit-sdk-py.

[18] Damian S. Steiger, Thomas Häner, and Matthias Troyer ProjectQ: An open source software framework for quantum computing, 2016.

[19] Thomas Häner, Damian S. Steiger, Krysta M. Svore, and Matthias Troyer A software methodology for compiling quantum programs, 2016.

[20] The Quantum Development Kit by Microsoft, homepage: https://www.microsoft.com/en-us/quantum/development-kit, github: https://github.com/Microsoft/Quantum.

[21] Michael A. Nielsen and Isaac L. Chuang, Quantum Computation and Quantum Information 10th Anniversary Edition, Cambridge University Press, 2011.

[22] Doug Finke, Education, Quantum Computing Report, https://quantumcomputingreport.com/resources/education/, accessed May 26, 2018.

[23] All code in this paper was run and tested on a Dell XPS 13 Developer Edition laptop running 64 bit Ubuntu 16.04 LTS with 8 GB RAM and an Intel Core i7-8550U CPU at 1.80 GHz. Programs were run primarily from the command line but the Python developer environment Spyder was also used for Python programs and Visual Studio Code was used for C# (Q#) programs.

[24] Forest: An API for quantum computing in the cloud, https://www.rigetti.com/index.php/forest, accessed May 14, 2018.

[25] Andrew W. Cross, Lev S. Bishop, John A. Smolin, Jay M. Gambetta, Open quantum assembly language, 2017.

[26] Scott Aaronson, Daniel Gottesman, Improved Simulation of Stabilizer Circuits, Phys. Rev. A **70**, 052328, 2004.

[27] pyQuil Lisence, github.com/rigetticomputing/pyquil/blob/master/LICENSE#L204, accessed June 7, 2018.

[28] Microsoft Quantum Development Kit License, marketplace.visualstudio.com/items/quantum.DevKit/license, accessed June 7, 2018.

[29] Hao Zhang, Chun-Xiao Liu, Sasa Gazibegovic, et al. Quantized Majorana conductance, *Nature* **556**, 74-79 (05 April 2018).

[30] 16-qubit backend: IBM QX team, "ibmqx5 backend specification V1.1.0," (2018). Retrieved from https://ibm.biz/qiskit-ibmqx5 and https://quantumexperience.ng.bluemix.net/qx/devices on May 23, 2018.

[31] Talia Gershon, Celebrating the IBM Q Experience Community and Their Research, March 8, 2018.

[32] M. Reck, A. Zeilinger, H.J. Bernstein, and P. Bertani, Experimental realization of any discrete unitary operator, Physical Review Letters, 73, p. 58, 1994.

[33] Ryan LaRose, Distributed memory techniques for classical simulation of quantum circuits, arXiv preprint (arXiv:1801.01037v1), 2018.

[34] Thomas Haner, Damian S. Steiger, *0.5 petabyte simulation of a 45-qubit quantum circuit*, arXiv preprint (arXiv:1704.01127v2), September 18, 2017.

[35] Jianxin Chen, Fang Zhang, Cupjin Huang, Michael Newman, Yaoyun Shi, Classical simulation of intermediate-size quantum circuits, arXiv preprint (arXiv:1805.01450v2), 2018.

[36] Alwin Zulehner, Robert Wille, *Advanced simulation of quantum computations*, arXiv preprint (arXiv:1707.00865v2), November 7, 2017.
[37] Jarrod R. McClean, Ian D. Kivlichan, Kevin J. Sung, et al., OpenFermion: The electronic structure package for quantum computers, arXiv:1710.07629, 2017.
[38] Nathan Killoran, Josh Izaac, Nicols Quesada, Ville Bergholm, Matthew Amy, Christian Weedbrook, Strawberry Fields: A Software Platform for Photonic Quantum Computing, arXiv preprint (arXiv:1804.03159v1), 2018.
[39] IonQ website, https://ionq.co/, accessed June 15, 2018.
[40] D-Wave: The quantum computing company, https://www.dwavesys.com/home, accessed June 20, 2018.
[41] Up until June 4, 2018, Rigetti had a 20 qubit computer that was publicly available to use. According to email with support staff, the change to the 8 qubit chip was "due to performance degradation." It is not clear if this change is temporary or if the 20 qubit processor will be retired permanently (and perhaps replaced by a new chip).
[42] Current members of the IBMQ network include those announced in December 2017–JP Morgan Chase, Daimler, Samsung, Honda, Oak Ridge National Lab, and others– and those announced in April 2018–Zapata Computing, Strangeworks, QxBranch, Quantum Benchmark, QC Ware, Q-CTRL, Cambridge Quantum Computing (CQC), and 1QBit. North Carolina State University is the first American university to be a member of the IBM Q Hub, which also includes the University of Oxford and the University of Melbourne. For a complete and updated list, see https://www.research.ibm.com/ibm-q/network/.
[43] Indeed, all four of these platforms are currently alpha or beta software. In writing this paper, two new versions of QISKit were released, as well as the new ACQUA library for chemistry and artificial intelligence, and a new version of pyQuil was released.
[44] It is conventional pyQuil syntax to import only the gates to be used: e.g., `from pyquil.gates import H, MEASURE`. The author prefers importing the entire `pyquil.gates` for clarity of instruction and for comparison to other programming languages, but please note that the preferred developer method is the former, which can nominally help speed up code and keep programs cleaner.
[45] We could just declare a single qubit and a single classical bit for this program instead of having a register and referring to (qu)bits by index. For larger circuits, it is generally easier to specify registers and refer to (qu)bits by index than having individual names, though, so we stick to this practice here.
[46] IBM's contest ending May 31, 2018, the "quantum developer challenge," is for writing the best compiler code in Python or Cython that inputs a quantum circuit and outputs an optimal circuit for a given topology.

Appendix A: Other Software Platforms

As mentioned in the main text, it would be counterproductive to include an analysis of all software platforms or quantum computing companies. For an updated and current list, see the 'Players' page on Quantum Computing Report [22]. Our selections in this paper were largely guided by the ability for general users to connect to and use real quantum devices, as well as unavoidable subjective factors like the author's experience. Our omission of software/companies is not a statement to their disability—here, we briefly mention a few.

a. Strawberry Fields Developed by the Toronto-based startup Xanadu, Strawberry Fields is a full-stack Python library for designing, simulating, and optimizing quantum optical circuits [38]. Xanadu is developing photonic quantum computers with continuous variable qubits, or "qumodes" (as opposed to the discrete variable qubits), and though the company has not yet announced an available quantum chip for general users, one may be available in the near future. Strawberry Fields has a built in simulators using Numpy and TensorFlow, and a quantum programming language called Blackbird. One can download the source code from GitHub, and example tutorials can be found for quantum teleportation, boson sampling, and machine learning. Additionally, the Xanadu website https://www.xanadu.ai/ contains an interactive quantum circuit where users can drag and drop gates or choose from a library of sample algorithms.

b. IonQ IonQ is a recent startup company located in College Park, Maryland and headed by researchers from the University of Maryland and Duke University. IonQ uses a trapped ion approach to quantum computing architecture, which has some very desirable properties. Using the nuclear spin of ^{171}Yb as a qubit, IonQ has achieved T2 (decoherence) times of 15 minutes, though 1 second T2 times are more usual. Additionally, the T1 (excitation relaxation) time is on the order of 20,000 years, and single qubit gate errors are around 10^{-4} using Raman transitions. For smaller size circuits, all-to-all connectivity is available because of long-range Coulomb interactions between ions, and it is possible to "shuttle" qubits to different linear arrays to achieve all-to-all connectivity in larger circuits. More interesting effects such as nonlocal teleportation between qubits via photon emission have also been demonstrated.

The hardware of IonQ is not currently available to general users via a software interface, but it is possible to contact IonQ and request access. Several algorithms ranging from machine learning to game theory have been run by experimenters. To read more about the company, please visit their website [39].

c. D-Wave Systems D-Wave [40] is perhaps the oldest quantum computing company. Founded in 1999 in Vancouver, Canada, D-Wave makes special types of adiabatic quantum computers, known as a quantum annealers, which solve for the ground-state energy of the transverse-field Ising model

$$H = \sum_i h_i \sigma_i^x + \sum_{i,j} J_{ij} \sigma_i^z \sigma_j^z .$$

D-Wave has manufactured several computers, the most recent being 2048 qubits, and has extensive software in Matlab, C/C++, and Python for programming them.

Because these quantum computers do not operate on the gate/circuit model of quantum computing, we did not include D-Wave in the main body of the text. Omitting D-Wave altogether, however, would not give an accurate picture of the current quantum computing landscape.

Appendix B: Testing Simulator Performance

Below is the listing of the program for testing the ProjectQ C++ local simulator performance. These tests were performed on a Dell XPS 13 Developer Edition running 64 bit Ubuntu 16.04 LTS with 8 GB RAM and an Intel Core i7-8550U CPU at 1.80 GHz.

```
# ----------------------------------------------------------------
# imports
# ----------------------------------------------------------------

from projectq import MainEngine
import projectq.ops as ops
from projectq.backends import Simulator
import sys
import time

# ----------------------------------------------------------------
# number of qubits and depth
# ----------------------------------------------------------------

if len(sys.argv) > 1:
    n = int(sys.argv[1])
else:
    n = 16

if len(sys.argv) > 1:
    depth = int(sys.argv[2])
else:
    depth = 10

# ----------------------------------------------------------------
# engine and qubit register
# ----------------------------------------------------------------

eng = MainEngine(backend=Simulator(gate_fusion=
    True), engine_list=[])
qbits = eng.allocate_qureg(n)

# ----------------------------------------------------------------
# circuit
# ----------------------------------------------------------------

# timing -- get the start time
start = time.time()

# random circuit
for level in range(depth):
    for q in qbits:
        ops.H | q
        ops.SqrtX | q
        if q != qbits[0]:
            ops.CNOT | (q, qbits[0])

# measure
for q in qbits:
    ops.Measure | q

# flush the engine
eng.flush()

# timing -- get the end time
runtime = time.time() - start

# print out the runtime
print(n, depth, runtime)
```

The circuit, which was randomly selected, is shown in Figure 7. We remark that the QISKit simulator was tested on an identical circuit—we omit the code for brevity.

Appendix C: Example Programs: The Teleportation Circuit

In this section we show programs for the quantum teleportation circuit in each of the four languages for a side by side comparison. We remark that the QDK program shown is one of three programs needed to run the circuit, as discussed in the main body. The teleportation circuit is standard in quantum computing and sends an unknown state from one qubit—conventionally the first or top qubit in a circuit—to another—conventionally the last or bottom qubit in a circuit. Background information on this process can be found in any standard quantum computing or quantum mechanics resource. This quantum circuit is more involved than the very small programs shown in the main text and demonstrates some slightly more advanced features of each language—e.g., performing conditional operations.

For completeness, we include a circuit diagram to make it clearer what the programs are doing. Unlike the main body of the text, this figure was made using the new circuit_drawer released in the latest version of QISKit.

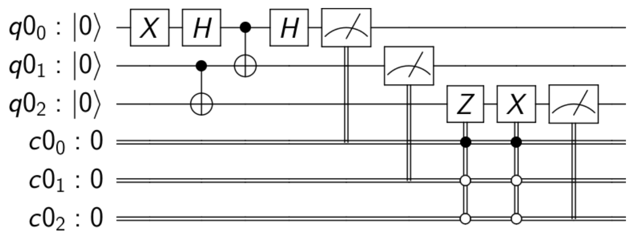

FIG. 8: The teleportation circuit produced with the circuit_drawer released in QISKit v0.5.4.

pyQuil

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ==========================================================
# teleport.py
#
# Teleportation circuit in pyQuil.
# ==========================================================

# ----------------------------------------------------------
# imports
# ----------------------------------------------------------

from pyquil.quil import Program
from pyquil import api
import pyquil.gates as gate

# ----------------------------------------------------------
# program and simulator
# ----------------------------------------------------------

qprog = Program()
qvm = api.QVMConnection()

# ----------------------------------------------------------
# teleportation circuit
# ----------------------------------------------------------

# teleport |1> to qubit three
qprog += gates.X(0)

# main circuit
qprog += [gates.H(1),
          gates.CNOT(1, 2),
          gates.CNOT(0, 1),
          gates.H(0),
          gates.MEASURE(0, 0),
          gates.MEASURE(1, 1)]

# conditional operations
qprog.if_then(0, gates.Z(2))
qprog.if_then(1, gates.X(2))

# measure qubit three
qprog.measure(2, 2)

# -------------------------------------------------------
# run the circuit and print the results
# -------------------------------------------------------

print(qvm.run(qprog))

# optionally print the quil code
print(qprog)
```

QISKit

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ==========================================================
# teleport.py
#
# Teleportation circuit in QISKit.
# ==========================================================

# ----------------------------------------------------------
# imports
# ----------------------------------------------------------

from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute

# ----------------------------------------------------------
# registers and quantum circuit
# ----------------------------------------------------------

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
qcircuit = QuantumCircuit(qreg, creg)

# ----------------------------------------------------------
# do the circuit
# ----------------------------------------------------------

# teleport |1> to qubit three
qcircuit.x(qreg[0])

# main circuit
qcircuit.h(qreg[0])
qcircuit.cx(qreg[1], qreg[2])
qcircuit.cx(qreg[0], qreg[1])
qcircuit.h(qreg[0])
qcircuit.measure(qreg[0], creg[0])
qcircuit.measure(qreg[1], creg[1])

# conditional operations
qcircuit.z(qreg[2]).c_if(creg[0][0], 1)
qcircuit.x(qreg[2]).c_if(creg[1][0], 1)

# measure qubit three
qcircuit.measure(qreg[2], creg[2])

# ------------------------------------------------------------
# run the circuit and print the results
# ------------------------------------------------------------
result = execute(qcircuit, 'local_qasm_simulator').result()
counts = result.get_counts()

print(counts)

# optionally print the qasm code
print(qcircuit.qasm())

# optionally draw the circuit
from qiskit.tools.visualization import circuit_drawer
circuit_drawer(qcircuit)
```

ProjectQ

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ==========================================================
# teleport.py
#
# Teleportation circuit in ProjectQ.
# ==========================================================

# ----------------------------------------------------------
# imports
# ----------------------------------------------------------
from projectq import MainEngine
from projectq.meta import Control
import projectq.ops as ops

# ----------------------------------------------------------
# engine and qubit register
# ----------------------------------------------------------

# engine
eng = MainEngine()

# allocate qubit register
qbits = eng.allocate_qureg(3)

# ----------------------------------------------------------
# teleportation circuit
# ----------------------------------------------------------

# teleport |1> to qubit three
ops.X | qbits[0]

# main circuit
ops.H | qbits[1]
ops.CNOT | (qbits[1], qbits[2])
ops.CNOT | (qbits[0], qbits[1])
ops.H | qbits[0]
ops.Measure | (qbits[0], qbits[1])

# conditional operations
with Control(eng, qbits[1]):
    ops.X | qbits[2]
with Control(eng, qbits[1]):
    ops.Z | qbits[2]

# measure qubit three
ops.Measure | qbits[2]

# ----------------------------------------------------------
# run the circuit and print the results
# ----------------------------------------------------------

eng.flush()
print("Measured:", int(qbits[2]))
```

Quantum Developer Kit

```
// ==========================================================
// teleport.qs
//
// Teleportation circuit in QDK.
// ==========================================================

operation Teleport(msg : Qubit, there : Qubit) : () {
        body {

            using (register = Qubit[1]) {
                // get auxiliary qubit to prepare for teleportation
                let here = register[0];

                // main circuit
                H(here);
                CNOT(here, there);
                CNOT(msg, here);
                H(msg);

                // conditional operations
                if (M(msg) == One)  { Z(there); }
                if (M(here) == One) { X(there); }

                // reset the "here" qubit
                Reset(here);
            }

        }
    }

    operation TeleportClassicalMessage(message : Bool) : Bool {
        body {
            mutable measurement = false;

            using (register = Qubit[2]) {
                // two qubits
                let msg = register[0];
                let there = register[1];

                // encode message to send
                if (message) { X(msg); }

                // do the teleportation
                Teleport(msg, there);

                // check what message was sent
                if (M(there) == One) { set measurement = true; }

                // reset all qubits
                ResetAll(register);
            }

            return measurement;
        }
    }
```