INSTITUTE FOR DEFENSE ANALYSES

Developing in the Commercial Quantum Cloud

Dale Visser, Project Leader
Arun S. Maiya

June 2022

Approved for public release;
distribution is unlimited.

IDA Document NS D-33113

INSTITUTE FOR DEFENSE ANALYSES
730 East Glebe Road
Alexandria, Virginia 22305

About This Publication

This work was conducted by the IDA Systems and Analyses Center under contract HQ0034-19-D-0001, Project C5211, "Quantum Software Development," for the IDA. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgements

Steven P. Wartik, David T. Chappell

For More Information

Dale Visser, Project Leader
dvisser@ida.org,

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Document NS D-33113

Developing in the Commercial Quantum Cloud

Dale Visser, Project Leader

Arun S. Maiya

Executive Summary

Quantum computing via cloud services is a fairly recent development. The major quantum cloud computing service providers (QCSPs) available in the United States are IBM Quantum, D-Wave Leap, Azure Quantum, Amazon Braket, and Google Quantum Computing Service. The quantum circuit model of computation, which currently applies to all the above services except D-Wave, relies on the presence of qubits, which are represented by a circuit line that interacts with boxes representing unitary gates. The ability of a set of qubits to represent all possible combinations of zeros and ones and the phenomenon of quantum entanglement within a multi-qubit state both contribute to the power of quantum computing to more efficiently compute many classes of problems. In many practical cases, there is a need to combine classical computing resources with the quantum processor in hybrid algorithmic workflows.

In this report, we demonstrate each QCSP in one or two configurations and describe how to set up a development environment for each. A critical piece of this is learning what quantum simulators are available for testing your code with smaller problems on classical hardware. In the case of D-Wave, the analog simulation resource is called a *sampling emulator*.

For the circuit model-based services, we present a quantum random number generator in code for each environment along with sample output. We selected this example to minimize the amount of quantum mechanical understanding needed by the reader while still demonstrating a truly quantum result produced by a hybrid algorithm.

The computing model for D-Wave is called *quantum annealing*, and it is able to set up larger numbers of coupled qubits, which are typically mapped to some optimization problem, and let the system “relax” into low energy states representing likely solutions. Typically, multiple runs are made, and the ground state properties are the results sampled from each run. In this paper, we use examples from Arun Maiya’s Quixotic framework to quickly and easily set up entire classes of optimization problems and submit them to D-Wave for a result.

Contents

1. Introduction

A. Scope

This report is meant to explain and explore what can currently be accomplished through the use of cloud providers for quantum computing. It provides a researcher with basic familiarity of the two commercially available quantum computing architectures, an understanding of the problems that may be solved with each, and instructions on how to get started with cloud offerings from Azure, IBM, Amazon, and D-Wave. We have assumed some familiarity with the basic concepts of quantum computing. *An Introduction to Quantum Computing* (Kaye, Laflamme, and Mosco 2007) is a good introductory textbook and is available electronically. Another good resource for software development practitioners is *Quantum Computing: Program Next-Gen Computers for Hard, Real-World Applications* (Mehta 2020), which takes the approach of initially hiding the usual mathematic formalism behind a simplified abstract graphical "Qubelet Model," in which qubits are represented as pictures called *qubelets*.

This report does not attempt to discuss quantum hardware or the future trajectory of quantum computing in any detail. That said, it is often said that we are in the era of Noisy Intermediate-Scale Quantum (NISQ) computers. Bharti et al. (2022) defines NISQ computers as being "composed of hundreds of noisy qubits, i.e., qubits that are not error corrected, and therefore perform imperfect operations within a limited coherence time." Bharti et al. is a literature review of what can be done with NISQ machines and of the future directions of quantum algorithms.

B. Setting Up a Development Environment

Although this report does show how to install and use the various quantum computing tools and environments, it was necessary to limit efforts to one or two environments per quantum cloud provider. The URLs given in the footnotes will lead to documentation that shows a variety of other valid alternative options.

We relied on the following personal preferences: (1) using a code editor like Visual Studio Code,[1] (2) using the popular high-level interpreted language, Python,[2] (3) working

[1] https://code.visualstudio.com/

[2] https://www.python.org/, "popular" according to the TIOBE index (https://www.tiobe.com/tiobe-index/) in February 2022

within a Linux environment when possible, and (4) leveraging containerization to define the environments. *Containerization* can refer to Open Container Initiative (OCI) containers, as provided by popular software like Docker or Podman.[3,4] We also recommend leveraging the Python virtual environment (venv) mechanism[5] when installing Python packages.

C. A Brief History of the Quantum Cloud

This paper does not provide an exhaustive list of quantum cloud service providers (QCSPs) that exist worldwide, but instead focuses on a few of the largest players based primarily in North America. Many companies are producing quantum computers and making them available to these QCSPs. This paper does not attempt to list these companies, but anyone exploring these cloud services will quickly discover their existence in the QCSP documentation of quantum compute options. For example, IonQ[6] and Rigetti[7] provide quantum compute resources to Amazon Web Services (AWS) Braket, Azure Quantum, and Google Quantum Computing Service.[8] Indeed, Google's initial QCSP offering had IonQ as its sole compute provider.

The timeline in Table 1 considers only the cloud services aspect of the companies' quantum computing efforts. D-Wave, for instance, has the most venerable quantum effort of the QCSPs, having been founded in 1999 to develop quantum hardware. D-Wave was also the first company to announce commercially available quantum computing hardware in May 2011 (Johnson et al. 2011). However, IBM began providing quantum cloud services over two years before D-Wave.

Table 1. A Timeline of Quantum Cloud Service Provider Launches

Provider Name	Public Launch Date
IBM Quantum (as IBM Quantum Experience)	May 2016
D-Wave Leap (v1)	October 4, 2018
Azure Quantum (Private Preview)	November 4, 2019

[3] https://www.docker.com/

[4] https://podman.io/

[5] https://docs.python.org/3/library/venv.html

[6] https://ionq.com/

[7] https://www.rigetti.com/

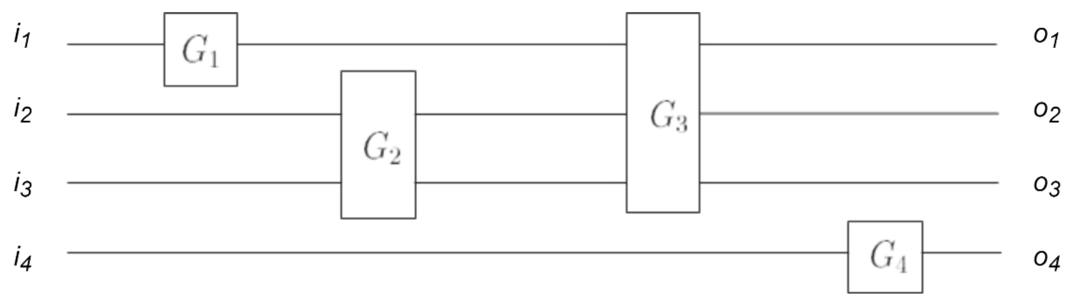
[8] As can be seen at https://aws.amazon.com/braket/, https://docs.microsoft.com/en-us/azure/quantum/overview-azure-quantum, https://quantumai.google/cirq/devices, and https://quantumai.google/cirq/rigetti/access

D-Wave Leap (v2)	February 26, 2020
Azure Quantum (Limited Preview)	May 19, 2020
AWS Braket	August 13, 2020
Azure Quantum (Public Preview)	February 1, 2021
IBM Quantum (redesigned and renamed)	March 2021
Google Quantum Computing Service	June 17, 2021

D. General Structure of Quantum Code

1. Circuit Model of Computation, and Reversibility

The opening chapter of Kaye, Laflamme, and Mosco (2007) introduces the circuit model of computation. This model (depicted in Figure 1) describes a form of Turing machine that takes its inputs on a finite set of *n* wires, each representing one classical bit (i.e., a value of 0 or 1). Algorithms are implemented by gates that take some number of wires, *k*, and generate outputs on the same number of wires, *k*. These gates are defined as *reversible*, which means that it must be possible to place a value on the output wires, run the gate in reverse, and reproduce the original input on the input wires.

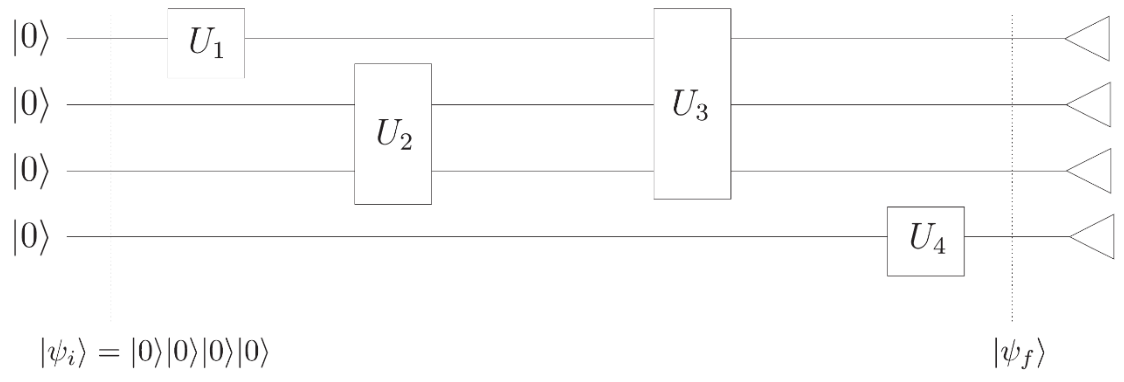


Note. Every gate has an equal number of input and output wires. This is necessary for the gates to be reversible. The inputs and outputs are classical bits (i.e., 0 or 1). Adapted from Kaye, Laflamme, and Mosco (2007).

Figure 1. Example of a Classical Circuit Model

The final output state of interest is represented by the bits measured from some set of output wires. The authors show it is always possible to map general non-reversible circuits to a reversible version that uses only reversible gates. To accomplish this, results of interest, whether intermediate or final, are copied to "spare" output wires that have no effect when running the circuit in reverse to see the inputs.

2. Quantum Circuit Model of Computation



Note. The triangles on the right represent measurement-basis measurement of the qubits in $|\psi_f\rangle$. Note that this circuit is only reversible back to $|\psi_i\rangle = |0000\rangle$[9] if one omits these final measurements (Kaye, Laflamme, and Mosco 2007).

Figure 2. A Quantum Circuit Corresponding to the Classical Circuit in Figure 1

This same reversible circuit model is often popular for defining quantum algorithms, where it is known as the *quantum circuit model of computation*, an example of which is shown in Figure 2. Each input bit of the prior circuit model is replaced by a qubit. A qubit can be represented in Dirac notation as $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$, showing its most general form as a superposition of the measurement-basis eigenstates. The coefficients, α and β, are

[9] The shorthand notation for four qubits, $|abcd\rangle$, is equivalent to writing $|a\rangle|b\rangle|c\rangle|d\rangle$.

called *amplitudes*. These amplitudes are complex numbers. Complex amplitudes are necessary in quantum mechanics to model wave mechanical interference effects. Reversible gates are replaced by unitary gates, which are the quantum equivalent. The gates alter the states of the qubits (possibly entangling qubit states together) such that a measurement of one qubit affects the subsequent measurement result of another.[10] Results are not simply the values of the output wires, as in the classical version previously described. Instead, one must make a measurement that will result in 0 or 1, with the probability of each depending on the qubit state. The asterisk operator in the following equation indicates the application of the complex conjugate:

$$(x + iy)^* \equiv x - iy.$$

For $|\varphi\rangle$ as defined earlier,

$$\alpha^*\alpha \ + \ \beta^*\beta = |\alpha|^2 + |\beta|^2 = 1,$$

$$P(0) = \ \alpha^*\alpha, \text{ and}$$

$$P(1) = \beta^*\beta \ .$$

P(x) means the probability of measuring the given measurement-basis state. Part of the power of quantum computing often lies in the ability to place sets of qubits in states that are admixtures of $|0\rangle$ and $|1\rangle$ and effectively compute against all possible states at once.

Applying gates to multiple qubits simultaneously often results in entanglement of the qubits (i.e., multi-qubit states that cannot be summarized simply as a set of separate one-qubit states). This second aspect of quantum mechanics is another oft-described feature that makes quantum computing powerful. The act of measurement can often project the portion of the solution space that solves a problem much quicker in, for example, $O(\log n)$ ("logarithmic time") rather than $O(n^k)$ ("polynomial time") or, worse yet, $O(e^n)$ ("exponential time").

While the circuit model defined above is the model most commonly implemented in quantum computing hardware and is the most studied in the theoretical/algorithmic literature, it is not the only model for quantum computing. D-Wave, described in more detail below, was early to market with *adiabatic* or *quantum annealing* (QA) computers. These rely much more on collective entanglement properties of many qubits at once, and they compute by preparing a state and then letting the system naturally seek the lowest, or

[10] Indeed, while outside the scope of this report, entanglement is often usefully leveraged in quantum algorithms.

ground, energy state. Repeating this process multiple times while measuring the ground state provides the desired result and is less sensitive to noisy qubits. The model is, however, more limited in the types of problems it can solve.

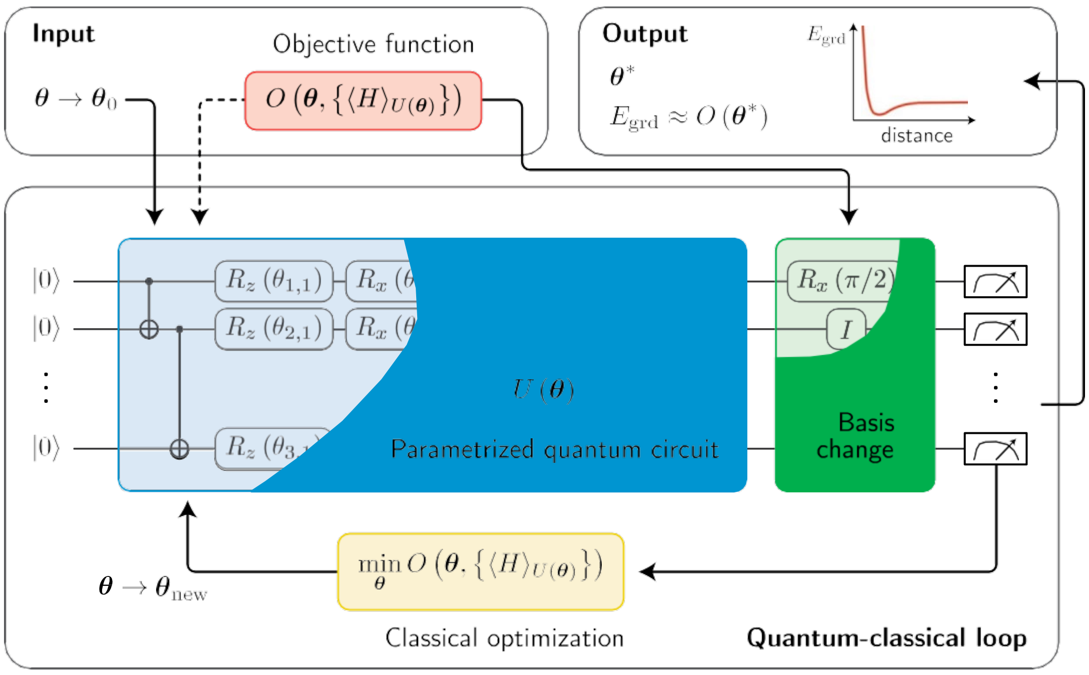
3. Classical Programs Are Still Needed

Classical here is defined in contrast with quantum — it really refers to standard electronic computers.[11] Despite being composed of semiconductor technology that leverages quantum material properties, standard computers perform their logic on standard classical electronic bits. The software frameworks described here typically have provision for both quantum and classical portions of computation, because it is quite typical for quantum software to have both a classical and quantum portion.

For example, consider the oft-mentioned Shor's algorithm (Shor 1994) for finding prime factors of large numbers in polynomial time, which will break the public key cryptography when quantum computers with enough sufficiently low-noise qubits become available. It consists of a classical algorithmic loop, where each iteration determines input for the quantum logic (which is based on the quantum Fourier transform). The output of the quantum computation is post-processed classically to determine if the factorization has succeeded or failed or if it needs additional iterations. In the last case, the results inform the next iteration of the loop.

Successfully executing algorithms like the one described earlier depends on the coordinated operation of a quantum computer and a classical computer. The latter drives the former, often using intermediate results to inform the next quantum execution (see Figure 3 for another example of this). Later in this document, the terms *quantum algorithm* and *quantum computation* refer to the strictly quantum portions of algorithms. *Classical algorithm* and *classical computation* will refer to the strictly classical portion. The terms *hybrid algorithm* and *hybrid computation* will be used to refer to computations such as Shor's algorithm described above. *Algorithm* or *computation* will be used without adjectives when the meaning is clear in context.

[11] This choice of words is an imperfect analogy with the contrast between classical Lagrangian/Newtonian mechanics and quantum mechanics.


Input
Objective function
$\theta \rightarrow \theta_0$
$O\left(\theta, \{\langle H \rangle_{U(\theta)}\}\right)$
Output
θ^*
$E_{\text{grd}} \approx O(\theta^*)$
E_{grd}
distance
$|0\rangle$
$R_z(\theta_{1,1})$
$R_z(\theta_{2,1})$
$U(\theta)$
Parametrized quantum circuit
$R_x(\pi/2)$
I
Basis change
$\min_{\theta} O\left(\theta, \{\langle H \rangle_{U(\theta)}\}\right)$
$\theta \rightarrow \theta_{\text{new}}$
Classical optimization
Quantum-classical loop

Note. $U(\vec{\theta})$[12] is a parameterized circuit that takes a set of angles and generates the initial state $|\overrightarrow{\theta_0}\rangle$ or subsequent $|\overrightarrow{\theta_{new}}\rangle$ states from the default all-$|0\rangle$ state. The angles are fed at each iteration into $U(\vec{\theta})$. After the basis-change circuit, which rotates to the energy-measuring Hamiltonion basis, the measurements indicate the evaluated ground state energy by classically evaluating the objective function. Well-known classical optimization techniques are used to arrive at a new estimate for $|\vec{\theta}\rangle$ or to determine that sufficiently precise convergence on the optimum has succeeded (Bharti et al. 2022).

Figure 3. Schematic Representation of a Hybrid Algorithm for Computing the Ground State of a Hamiltonian (H)

[12] The $\vec{\theta}$ notation is used to make explicit that θ in the figure represents a set of angles.

2. Quantum Cloud Providers Overview

A. Overview

Among the circuit-based QCSPs, Azure and IBM Quantum compete not only through their respective frameworks and libraries, but also by offering competing programming language visions. IBM's OpenQASM is an academic and industry-standard language that has evolved over several years. Azure's Q#, on the other hand, provides a familiar paradigm for .NET developers and has the benefit of having been designed from the ground up to be explicit about quantum operations versus classical algorithms. Every provider offers software frameworks and associated libraries with at least a Python binding, but they often offer bindings for other languages as well. Serious developers of quantum algorithms need the capability to simulate their quantum logic on classical hardware, and each QCSP makes provision for this as well, offering multiple simulators of varying capability, scale, and purpose. The analogs to simulators in annealing-based systems are sampling emulators.

All providers provide free trial usage options for learning purposes, with the exception of Google which is still in an early private preview stage. However, it is still possible to download Google's framework and run it using one of the provided local simulators. For comparison, Table 2 provides some details on the QCSP offerings.

Table 2. Summary of QCSPs Discussed in this Report

Provider	Circuit or Annealing-based	Available Simulators	Own Language?	Framework and Libraries	Web-Based Development[13]
Azure Quantum	Circuit	Full state, Sparse, Simple resource estimator, Trace-based resource estimator, Toffoli, Noise[14]	Q#	Microsoft QDK	No

[13] Not counting the ability to develop using Jupyter notebooks, which are supported by all providers in the table.

[14] https://docs.microsoft.com/en-us/azure/quantum/machines/

Provider	**Circuit or Annealing-based**	**Available Simulators**	**Own Language?**	**Framework and Libraries**	**Web-Based Development**[13]
IBM Quantum	Circuit	State vector, Stabilizer, Extended stabilizer, MPS, QASM[15]	OpenQASM	Qiskit	Composer GUI
Google Quantum Compute Service	Circuit	State vector or Density matrix[16]	No	Cirq, OpenFermion, TensorFlow Quantum[17]	No
AWS Braket	Both	Local (State Vector only), 3 Hosted options: State vector, Density matrix, Tensor network	No	AWS Braket software development kit (SDK)	No
D-Wave Leap	Annealing (with plans for circuit)	Optimizing and sampling emulator solvers[18]	No	Ocean SDK[19]	GitPod- and JupyterHub-based[20]

B. Azure Quantum

From a user perspective, the Azure Quantum offering is centered around the Microsoft Quantum Development Kit (QDK),[21] which can be used on Linux, Windows, or MacOS (see Figure 4). To have access to quantum hardware via the Azure Quantum

[15] https://quantum-computing.ibm.com/admin/docs/admin/manage/simulator/

[16] https://quantumai.google/cirq/simulation

[17] https://quantumai.google/cirq, https://quantumai.google/openfermion and https://www.tensorflow.org/quantum

[18] https://docs.dwavesys.com/docs/latest/c_solver_intro.html#emulators

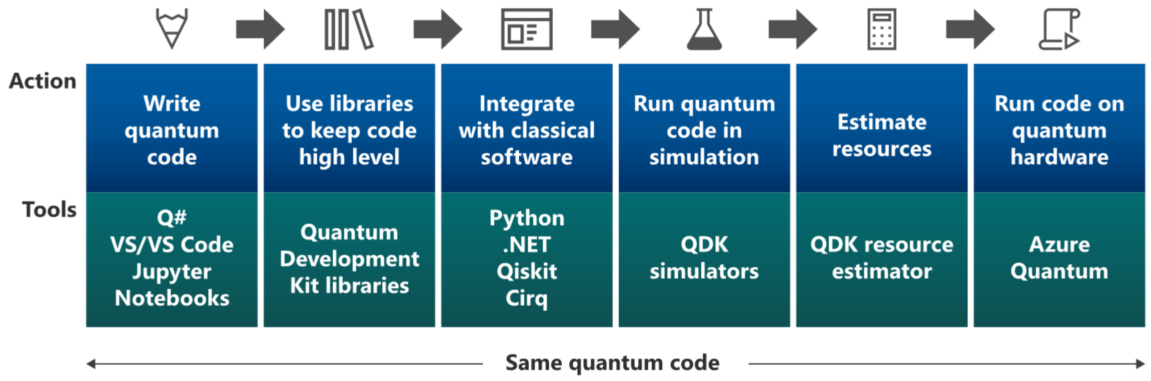
[19] https://docs.ocean.dwavesys.com/en/stable/

[20] https://www.gitpod.io/ and https://jupyter.org/hub

[21] https://www.microsoft.com/en-us/quantum/development-kit/

service, you will need login access to the Azure service. Use a Microsoft Account[22] to sign in at https://portal.azure.com/.

Q# is the QDK's language for expressing hybrid algorithms. It is also possible to code just the inherently quantum portions of algorithms in Q# and use a classical computing language like Python 3 or C# to drive the execution. For the example in Appendix A, the environment was installed using the ".NET CLI and pip" instructions at https://docs.microsoft.com/en-us/azure/quantum/install-python-qdk while installing the Python 3 *qsharp* package into a Python venv. When using an activated venv, the *python* command always invokes the venv's version of Python, which is why the *python3* command is not seen in this section.



Action
Tools
Write quantum code
Use libraries to keep code high level
Integrate with classical software
Run quantum code in simulation
Estimate resources
Run code on quantum hardware
Q# VS/VS Code Jupyter Notebooks
Quantum Development Kit libraries
Python .NET Qiskit Cirq
QDK simulators
QDK resource estimator
Azure Quantum
Same quantum code

Note. Image source: https://docs.microsoft.com/en-us/azure/quantum/overview-azure-quantum

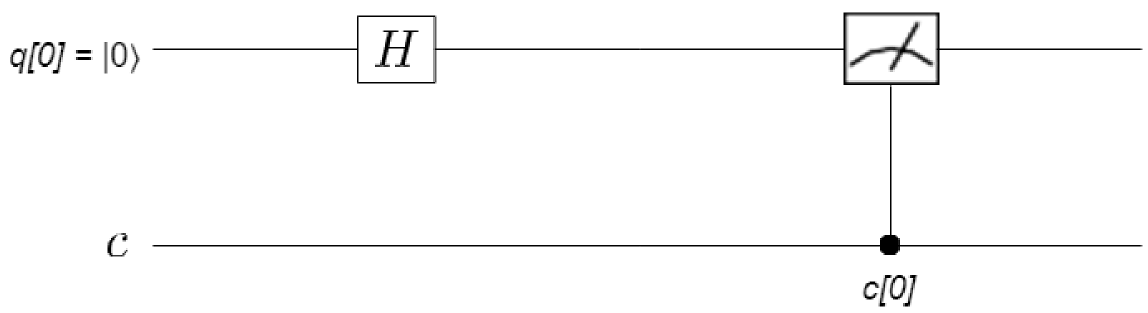
Figure 4. Stages of Quantum Code Development with the Microsoft QDK

Appendix A contains an example adapted from the QDK tutorials that leverages a qubit to generate random integers as sequences of random bits. It contains two source files. The first file, *qrng.qs*, contains Q# source code defining both quantum code and all the necessary classical code to drive it. As will be shown, it can be launched directly using .NET command-line tools. The second file, *qrng_host.py*, shows that the classical parts of the code can also be implemented in Python, and the quantum code in *qrng.qs* can be driven from a Python process.

In *qrng.qs*, a quantum function,[23] SampleQuantumRandomNumberGenerator, is defined. This function generates a qubit measurement result, with coin flip odds of returning the special measurement result values *Zero* or *One*. This function is purely quantum, and it defines the circuit depicted in Figure 5:

[22] A Microsoft login account can be created at https://signup.live.com/.

[23] Functions are called operations in Q#.



Note. Adapted from Mehta (2020). This figure is very close to the diagrams generated by IBM's Qiskit (described later) and to the appearance of the same circuit in the Qiskit graphical coding environment. The top line represents the single qubit. The boxed H represents the Hadamard gate, which places the qubit into a 50-50 admixture of $|0\rangle$ and $|1\rangle$ (specifically $(|0\rangle + |1\rangle)/\sqrt{2}$). The analog gauge symbol represents a measurement in the $|0\rangle$ and $|1\rangle$ measurement bases, with the 0 or 1 result being placed in the classical register, *c*.

Figure 5. Quantum Circuit for Generating a Random Classical Bit

Next, *qrng.qs* defines a classical function, SampleRandomNumberInRange, that invokes the first function enough times to generate a bit string that can range from zero to a given maximum value. If the bit string represents an integer greater than the maximum, it preserves a uniform probability distribution by starting over and trying again until a valid result is obtained.

The final function in *qrng.qs*, RunIt, is classical and is tagged as the entry point, accepting an argument from the command line. It invokes the second function and prints human-readable output. The code is run from source folder by invoking, for example:

```
dotnet run --max-result=100
```

If the --max-result argument isn't provided, a useful help message will be output explaining how to provide the missing input.

The Python 3 script *qrnh_host.py* leverages a Python package provided in the QDK called *qsharp*.[24] This package is never explicitly invoked in the script, but the import qsharp statement makes it possible to then import the quantum operation from our Q# file and call it. From the command line, it is invoked similarly, for example:

```
python qrng_host.py 100
```

To run this on actual Azure-accessible quantum hardware, the Python script needs the following lines included after the import section:

```
qsharp.azure.connect(
  resourceId="/subscriptions/.../Microsoft.Quantum/Workspaces/WORKSPACE_NAME",
  location="West US")
```

[24] https://docs.microsoft.com/en-us/python/qsharp-core/qsharp

```
qsharp.azure.target("ionq.simulator")
```

where the resourceId and location strings, and the target argument string are replaced by appropriate values for your credentials and preferred processor. The line invoking the local quantum simulator is replaced with this:

```
yield qsharp.azure.execute(SampleQuantumRandomNumberGenerator, shots=1,
                 jobName="Generate random bit")
```

If you attempt to run this code, you will likely discover that each execution of the operation on the cloud's quantum hardware is subject to high demand and time-consuming queuing.[25] With an execution request for each randomly generated bit, it could take quite a while to get a result. One could certainly go back to the drawing board and improve the algorithm to simultaneously prepare and measure multiple independent qubits at once up to the number of qubits provided by that quantum machine. However, the goal is not to instruct on how to create the best or fastest quantum random number generator. Quantum computers are unlikely to make current hardware random number generators obsolete in the coming decades. Rather, the goal is to illustrate the cooperation of classical and quantum computing hardware with an easily understood quantum operation.

C. IBM Quantum

1. Description

IBM has been offering cloud access to quantum computers since 2016 and claims to be the first company to do so. The company has steadily been expanding the capabilities of its quantum hardware, regularly graduating research hardware to production use in its cloud. As of this writing, there are 23 different systems available on IBM's cloud, computing with quantities of qubits ranging from 1 to 65. IBM's current research system has 127 qubits, and there is a public roadmap[26] with plans for systems with more than 1,000 qubits by 2024.

IBM's own language for quantum development is called OpenQASM.[27] The author of this report has, in the past, used IBM's online resources to learn about quantum programming with OpenQASM 1.0 and OpenQASM 2.0, which had limited capabilities for the classical portion of hybrid algorithm. The most recent revision, OpenQASM 3.0, has added more support for the classical side, enabling full hybrid algorithms running on cloud systems that support it.

[25] Typically lasting a few minutes.

[26] https://research.ibm.com/blog/quantum-development-roadmap

[27] QASM is a generally used abbreviation standing for Quantum Assembly/Assembler

IBM has a very approachable interface for newcomers to quantum circuits, called the IBM Quantum Composer, which is a graphical interface for directly manipulating a quantum circuit diagram to build up circuits. Graphical manipulations are translated into the OpenQASM language; the OpenQASM representation may be directly edited and the changes are reflected in the graphical representation. Circuits may be run on a simulator or, if an IBM Quantum account is obtained, run on one of the available quantum computers in IBM's cloud offering.

2. Installation

We found that leveraging IBM Quantum's Python 3 package, *Qiskit*,[28] and Jupyter was a very quick way to approach using IBM Quantum. The following instructions detail how to do this in any typical Linux environment:

1. Create a working folder and activate a venv within.
   a. mkdir -p qiskit/venv && cd qiskit
   b. python3 -m venv
   c. source venv/bin/activate
2. Upgrade installation tools to the latest versions:[29] pip install --upgrade pip setuptools
3. Install Qiskit and, optionally, Jupyter and associated packages.
   a. pip install qiskit
   b. pip install jupyter jupyterlab matplotlib
4. If you wish to work with the Qiskit textbook at https://qiskit.org/textbook,
   a. pip install --upgrade wheel
   b. pip install seaborn

As with other quantum SDKs, developers usually start with a quantum simulator that runs locally before running their algorithms on actual quantum hardware. IBM Quantum is the default quantum backend; IonQ and Azure Quantum, among others, are supported.[30] Qiskit Runtime[31] is a cloud offering that can run full hybrid algorithms written using the qiskit-ibm-runtime Python package.

[28] https://qiskit.org/

[29] Because of the venv, the correct Python 3 *pip* executable is aliased.

[30] https://qiskit.org/documentation/partners/

[31] https://github.com/qiskit/qiskit-ibm-runtime

See Appendix B for a Jupyter notebook that implements a version of the quantum random number generation example introduced in the Azure Quantum section. The notebook uses the Aer simulator,[32] a local simulator capable of simulating noisy qubits. If you wish to run code on an actual quantum computer, obtain an API token from your user profile page on IBM Quantum and run the following short Python program on your system:

```
from qiskit import IBMQ
IBMQ.save_account('«API Token»')
```

This will save your credential to a local file. To use your credential in your programs, execute the following commands first:

```
from qiskit import IMBQ
provider = IBMQ.load_account()
```

If you want to connect to any real quantum device, you can modify the first section of the notebook where the *qiskit.providers* package is imported from:

```
from qiskit.providers.ibmq import least_busy
real_devices = provider.backends(simulator=False, operational=True)
backend = least_busy(real_devices)
```

In general, there will be a queue of jobs in front of yours. Even if there is no wait, there is latency associated with submission into the job queue, compilation into control commands for the quantum hardware, and execution. Now, when qiskit.execute() is invoked, it will be necessary to wait for the job result to be ready before accessing it. It is possible to view your submitted jobs on the IBM Quantum web interface. A simple way to wait on the result in your code is to insert the following command:

```
job.wait_for_final_state()
```

In a real application, it would be wise to add a timeout parameter to the call and check for possible exceptions.

D. Google Quantum Computing Service

Google's Quantum Computing Service[33] is, at present, only open to those on an early-access list. There is a questionnaire[34] that research projects can use to request access. As with the other cloud services, Google offers a Python package that allows for coding the quantum circuit model, called Cirq.[35] For completeness, Appendix D includes a version of

[32] https://github.com/Qiskit/qiskit-aer

[33] https://quantumai.google/quantum-computing-service

[34] https://docs.google.com/forms/d/1DfUWu4zUAJ87GKy-ZoTHrFri5IwIteKtMxKfsy3lmHE

[35] https://pypi.org/project/cirq/

the quantum random number generator example that can be run locally using Cirq's built-in circuit simulator. A list of available quantum processors and simulators is also available.[36]

E. Amazon Braket

Introductory reading material on AWS focuses on keeping the entire quantum development workflow inside the AWS infrastructure. When doing this, the simulation of quantum circuits is performed on the same AWS Elastic Compute Cloud (EC2) VM instance as your Jupyter notebook or by a managed quantum simulator hosted by AWS. Like the other cloud providers, an SDK is provided as a Python package, amazon-braket-sdk.[37] We will also describe how to set up a local environment that uses the SDK, which can use a local simulator, or to invoke actual quantum hardware via AWS. Complex simulations or actual quantum processing are accomplished via properly authenticated calls into AWS infrastructure.

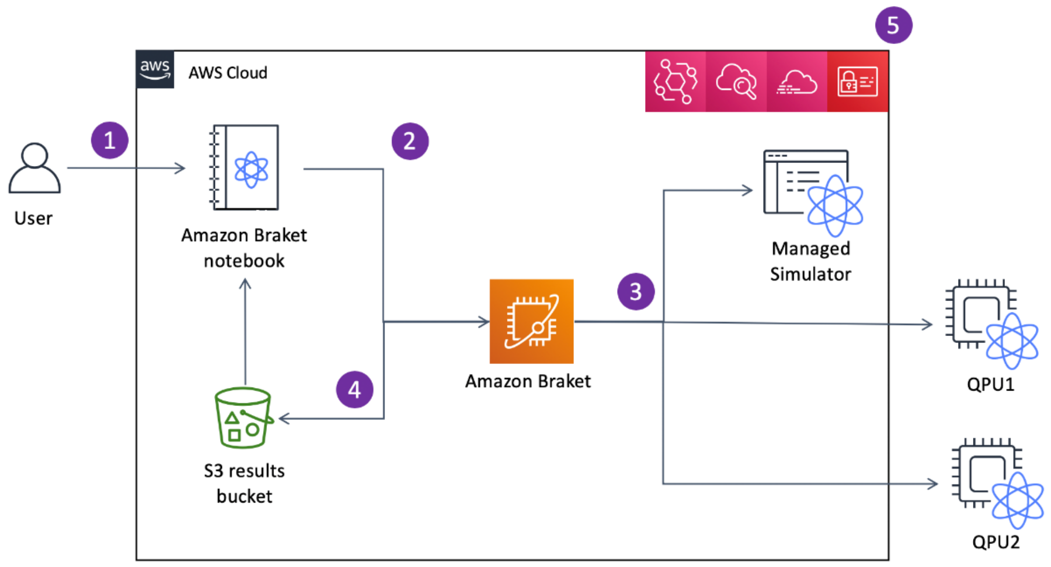
To work within an AWS-provisioned Jupyter notebook requires spinning up a specialized Sagemaker EC2 instance that is at least an ml.t3.medium instance (250 free hours for the first two months after creating your first instance, $0.05/hour thereafter).[38] To save money in initial experimentation, it is possible to run simulations locally on your notebook instance. To accomplish this, launch jobs in your notebooks using the create() method of the class braket.jobs.local.local_job.LocalJob. This will spin up a Docker container for running the type of simulator you specified. Even more simply, you can use the LocalSimulator class.[39] Charges for managed simulator (typically $/minute of processing) or quantum processing unit (QPU) usage (typically billed in USD/task plus USD/shot) are easily browsed once you have enabled Braket on your AWS account.

[36] https://quantumai.google/cirq/ecosystem#supported_quantum_cloud_services

[37] https://github.com/aws/amazon-braket-sdk-python and https://amazon-braket-sdk-python.readthedocs.io/en/latest/

[38] https://aws.amazon.com/sagemaker/pricing

[39] https://docs.aws.amazon.com/braket/latest/developerguide/braket-get-started-run-circuit.html



Note. The developer (1) authors circuits and other code on a Jupyter notebook in AWS, (2) runs simulations on an AWS-hosted quantum simulator, and when ready, and (3) executes on actual quantum hardware, termed Quantum Processing Units (QPUs). (4) Results are made available in S3 buckets, and (5) integration with other AWS services like Identity, IAM, CloudWatch, CloudTrail and EventWatch is available. (Source: https://docs.aws.amazon.com/braket/latest/developerguide/braket-how-it-works.html.)

Figure 6. Amazon Braket Workflow

Braket also offers support for the hybrid quantum programs — that is, standard compute resources relatively local to the QPU can work in a feedback loop iterating quantum jobs based on prior quantum results. The GitHub repo cited at the beginning of this section includes a README with instructions for using the AwsQuantumJob class to execute *examples/job.py*.

Amazon offers three different managed quantum simulators: a state vector simulator (SV1), a tensor network simulator (TN1), and a density matrix simulator (DM1).[40] Thus far, Amazon does not natively produce its own QPU hardware. Rather, it relies on third-party providers IonQ and Rigetti. It even allows you to run on QA hardware at D-Wave, which is discussed in the next section. As of March 7, 2022, Braket supports IBM's OpenQASM 3.0 for applicable gate-based QPUs. When enabling Braket on an AWS account, it is made clear that your code, data, and results transit to/from outside of AWS when executing on these third-party resources (also see Figure 6).

Braket's documentation is not explicit about the fact that that you can install its SDK locally and play with it using a local quantum simulator.[41] To run on one of the managed simulators or an actual QPU from your own installation, set up credentialed connections as

[40] https://docs.aws.amazon.com/braket/latest/developerguide/braket-devices.html#choose-a-simulator

[41] See https://github.com/aws/amazon-braket-sdk-python#available-simulators and examine the source code at *examples/local_bell.py* in the repository.

described at https://github.com/aws/amazon-braket-sdk-python#boto3-and-setting-up-aws-credentials Then, as shown in the example at *examples/bell.py*, replace your LocalSimulator device with a device instance leveraging an AWS resource:

```
from braket.aws import AwsDevice
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

Braket has mature support for prioritized hybrid algorithm execution.[42] As detailed at the resource in the footnote, it is expected that the hybrid code obtains the device Amazon Resource Name (ARN)[43] from its environment. There are multiple ways to set the environment variables; the most straightforward is to use braket.aws.AwsQuantumJob.create(...) when submitting the job from code. It takes several arguments. The first positional argument is the ARN of the QPU (or simulator) you want to use. Additional arguments specify source file, code entry point, continuation form a prior incomplete job, and S3[44] bucket to store results in.

F. D-Wave

1. Leap

D-Wave may be the oldest quantum computing company, having been founded in 1999. To date, the company has focused on QA, in which qubits and the couplings between them naturally seek their lowest energy state(s). This contrasts with the quantum circuit computation model described earlier and used by all the other services. Annealing has provided an advantage of being much less susceptible to qubit noisiness, making it easier to build systems that can make use large numbers of qubits. As described below, a large class of optimization problems of business and logistics interest lend themselves to being computed this way.

QA lacks the general applicability of the quantum circuit model of computation (often called *gate-model computation* in D-Wave's literature). However, for the subset of optimization problems it is suited for, it works well, and has been finding practical application for years already. These are typically tasks that can be expressed using a binary quadratic model. D-Wave's libraries allow these to be defined as quadratic unconstrained binary optimization (QUBO) problems or through the use of an Ising model, which is more familiar to physicists and material scientists. QA is best used to solve such issues as logistics with complex constraints, modeling molecular interactions, and financial portfolio optimization (e.g., optimizing between yield and risk).

[42] https://docs.aws.amazon.com/braket/latest/developerguide/braket-jobs.html

[43] https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

[44] Simple storage service, see https://docs.aws.amazon.com/s3/

D-Wave Leap provides two easy ways to start learning and using its system within your web browser. Both require you to first set up a developer account at https://cloud.dwavesys.com/leap/signup.

For a Visual Studio Code-like experience, working with Python source code files, D-Wave offers a GitPod-based[45] cloud IDE called D-Wave Leap.[46] With it, one can directly browse and open the various code examples offered at https://github.com/dwave-examples/. As with the gitpod.io service, it is also possible to enter any GitHub project URL and access it directly with a prepared D-Wave environment.

There are also Jupyter notebook projects in the *dwave-examples* GitHub organization page. There is a prepared JupyterHub environment[47] for these that allows one to easily open and work with any of the examples.

It is relatively simple to set up a local environment for any of the example projects as well. The following steps should work in most Linux environments:

- git clone https://github.com/dwavesys-examples/«name-of-project»
- cd «name-of-project»
- python3 -m venv .venv
- source .venv/bin/activate
- pip install --upgrade pip setuptools wheel
- pip install -r requirements.txt
- (First project only) Install your D-Wave Leap credentials using the steps described at https://docs.ocean.dwavesys.com/en/stable/overview/sapi.html.

After these steps, peruse the example project README, which will instruct you on running the code or starting the notebook server, as appropriate.

2. Quixotic Framework

Arun Maiya has produced a Python framework, called Quixotic,[48] that further simplifies the generation and running of these types of problems. Quixotic puts various popular graph algorithms within easier reach for defining and submitting to D-Wave's quantum processors or to AWS (which, as of this writing, partners with mostly gate-based quantum processing providers; D-Wave is its sole annealing-based processing provider). In addition to the written guidance on the Quixotic website, we have provided a Jupyter

[45] https://www.gitpod.io/#get-started

[46] Accessible at https://cloud.dwavesys.com/leap/

[47] Accessible at https://cloud.dwavesys.com/learning/hub/home

[48] https://amaiya.github.io/quixotic/

notebook in Appendix E that demonstrates querying the framework for supported algorithms (or tasks) and sampling a couple of them both locally and on a D-Wave QPU. Just as with running the D-Wave example projects, it is necessary to first place your API token in your local environment. To install the latest Quixotic version 0.0.6, it is very important to use Python 3.7 or Python 3.8 so that pre-built Python Package Index (PyPI) binary wheels can be downloaded. You can mostly follow the same instructions used for the D-Wave environment above. However, instead of the

```
pip install -r requirements.txt
```

step, do

```
pip install quixotic jupyter
```

Of course, Jupyter is unnecessary if you are not using notebooks.

Appendix A. Azure Code

A. qrng.qs

The source code below is written in Q#, Microsoft's language, which can implement hybrid quantum algorithms.

```
namespace Qrng {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Arrays;

    // Quantum operation that generates a single random classical
    // bit using a simple quantum circuit.
    operation SampleQuantumRandomNumberGenerator() : Result {
        use q = Qubit();  // Allocated by default to |0⟩

        // Put the qubit to superposition
        H(q);  // H|0⟩ = |+⟩ = (|0⟩ + |1⟩)/sqrt(2)

        // Measure the qubit value, with probabilities given by:
        //   P(Zero) = ⟨+|0⟩⟨0|+⟩ = 50%
        //   P(One)  = ⟨+|1⟩⟨1|+⟩ = 50%
        return M(q);
    }

    operation SampleRandomNumberInRange(max : Int) : Int {
        mutable output = 0;  // Q# requires binding a value at declaration
        repeat {
            mutable bits = EmptyArray<Result>();
            for idxBit in 1..BitSizeI(max) {
                set bits += [SampleQuantumRandomNumberGenerator()];
            }
            set output = ResultArrayAsInt(bits);
        } until (output <= max);  // # i.e., redo QRNG if answer is too big
        return output;
    }

    @EntryPoint()
    operation RunIt(max_result: Int) : Unit {
        Message("Using quantum RNG to generate a number from 0 to " +
              $"{max_result}.");
        mutable output = SampleRandomNumberInRange(max_result);
        Message($"The generated number is {output}.");
    }
}
```

B. qrng_host.py

```
"""
Script that invokes quantum code in Qrng.qs for a silly bit-by-bit
quantum random number generation.
"""
import sys
from typing import Iterable

import qsharp
from Qrng import SampleQuantumRandomNumberGenerator


# generate random numbers from 0..max, which may be provided as an argument
MAX_RESULT = 50 if len(sys.argv) < 2 else int(sys.argv[1])

def generate_random_bits() -> Iterable[int]:
    """
    Call the quantum operation as many times as there are bits needed to
    define the maximum of the range. For example, if max == 7, you need three
    bits to generate all the numbers from 0 to 7.
    """
    for _ in range(0, MAX_RESULT.bit_length()):
        # Call the quantum operation and store the random bit in the list
        yield SampleQuantumRandomNumberGenerator.simulate()


def convert_to_int(bit_string: Iterable[int]) -> int:
    """
    Interpret the bit string as a binary literal.
    """
    return int("".join(str(x) for x in bit_string), 2)

print(f"Using quantum RNG to generate a number from 0 to {MAX_RESULT}.")

# Variable to store the output
RESULT = MAX_RESULT + 1
while RESULT > MAX_RESULT:  # i.e., redo QRNG if answer is too big
    RESULT = convert_to_int(generate_random_bits())

print(f"The generated number is {RESULT}.")
```

Appendix B. IBM Qiskit Jupyter Notebook

A. In [1]

```
import qiskit
from qiskit import (__qiskit_version__, QuantumCircuit, QuantumRegister,
                ClassicalRegister, execute)
from qiskit.providers.aer import AerSimulator
qiskit.__qiskit_version__
```

B. Out[1]

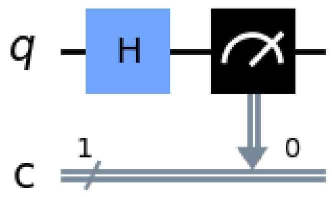
```
{'qiskit-terra': '0.19.2', 'qiskit-aer': '0.10.3', 'qiskit-ignis': '0.7.0', 'qiskit-ibmq-provider': '0.18.3', 'qiskit-aqua':
None, 'qiskit': '0.34.2', 'qiskit-nature': None, 'qiskit-finance': None, 'qiskit-optimization': None, 'qiskit-
machine-learning': None}
```

C. In[2]

```
%matplotlib
import numpy as np
import math
from qiskit.visualization import plot_histogram

# Define circuit, including measurements
circuit = QuantumCircuit(1, 1)
circuit.h(0)
circuit.measure(0, 0)
circuit.draw(output='mpl')
```

D. Out[2]

```
Using matplotlib backend: agg
```



E. In[3]

```
backend = AerSimulator()

# Leverage a qubit to calculate a single random bit
```

```
def quantum_random_bit():
    job = execute(circuit, backend, shots=1)
    hist = job.result().get_counts()
    return 0 if '0' in hist else 1
```

F. In[4]

```
from typing import Iterable

def generate_random_bits(upper_limit: int) -> Iterable[int]:
    """
    Call the quantum operation as many times as there are bits needed to
    define the maximum of the range. For example, if max == 7, you need three
    bits to generate all the numbers from 0 to 7.
    """
    for _ in range(0, upper_limit.bit_length()):
        # Call the quantum operation and store the random bit in the list
        yield quantum_random_bit()

def convert_to_int(bit_string: Iterable[int]) -> int:
    """
    Interpret the bit string as a binary literal.
    """
    return int("".join(str(x) for x in bit_string), 2)

def quantum_random_int(upper_limit: int):
    """
    Using quantum RNG to generate a number from 0 to upper_limit
    """
    result = upper_limit + 1
    while result > upper_limit:  # i.e., redo QRNG if answer is too big
        result = convert_to_int(generate_random_bits(upper_limit))
    return result
```

G. In[5]

```
# E.g., throw 100-sided quantum die 5 times
for _ in range(5):
    print(1 + quantum_random_int(99))
```

H. Out[5]

```
18
60
52
54
79
```

Appendix C. Braket Code

Below is the Quantum Random Number Generator example as entered into a Braket/Sagemaker Jupyter notebook.

```
# general imports
import matplotlib.pyplot as plt
%matplotlib inline

# AWS imports: Import Braket SDK modules
from braket.circuits import Circuit
from braket.devices import LocalSimulator
```

A. Build a random bit circuit

```
# Define coin-flip circuit (measurement is automatic when run)
circuit = Circuit().h(0)

# This shows a circuit diagram representation. The T lines
# show time steps, of which there is only one in in this
# particular circuit.
print(circuit)
T  : |0|
q0 : -H-
T  : |0|
```

B. Use a local simulator to run on a Sagemaker notebook machine

This is more limited in capability than the AWS managed simulators.

```
# set up device
device = LocalSimulator()

# Leverage a qubit to calculate a single random bit
def quantum_random_bit():
    result = device.run(circuit, shots=1).result()
    counts = result.measurement_counts
    return 0 if '0' in counts else 1
```

C. Implement classical logic side of hybrid code

This code is identical to that in the Qiskit example.

```
from typing import Iterable
```

```
def generate_random_bits(upper_limit: int) -> Iterable[int]:
    """
    Call the quantum operation as many times as there are bits needed to
    define the maximum of the range. For example, if max == 7, you need three
    bits to generate all the numbers from 0 to 7.
    """
    for _ in range(0, upper_limit.bit_length()):
        # Call the quantum operation and store the random bit in the list
        yield quantum_random_bit()

def convert_to_int(bit_string: Iterable[int]) -> int:
    """
    Interpret the bit string as a binary literal.
    """
    return int("".join(str(x) for x in bit_string), 2)

def quantum_random_int(upper_limit: int):
    """
    Using quantum RNG to generate a number from 0 to upper_limit
    """
    result = upper_limit + 1
    while result > upper_limit:  # i.e., redo QRNG if answer is too big
        result = convert_to_int(generate_random_bits(upper_limit))
    return result


# E.g., throw 100-sided quantum five times
for _ in range(5):
    print(1 + quantum_random_int(99))
35
38
70
82
96
```

Appendix D. Google Cirq Code

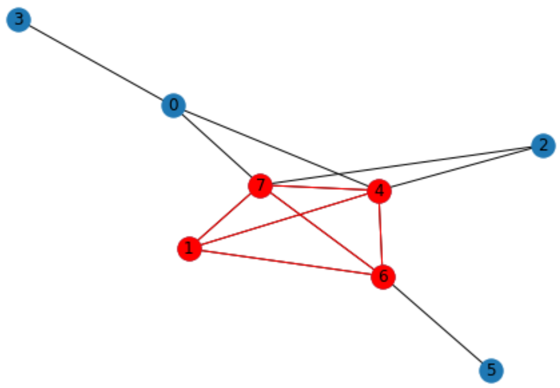
A. Python source file qrng.py

This is the Quantum Random Number Generator example coded against Google's Python package, *cirq*. It borrows heavily from earlier Python codes.

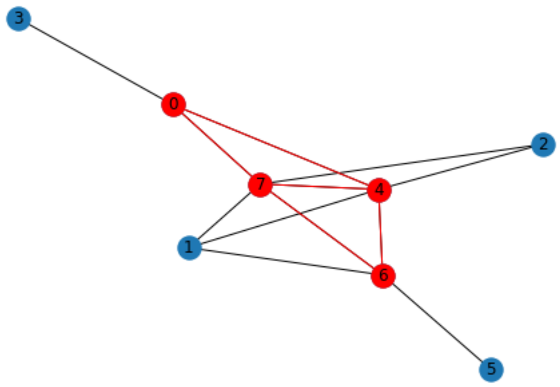
```
import cirq
from typing import Iterable

# 1. Define coin-flip circuit (measurement is automatic when run)

qubit = cirq.GridQubit(0,0)
circuit = cirq.Circuit(cirq.H(qubit),
                       cirq.measure(qubit, key='m'))
print(circuit)

# 2. Use a local simulator, and leverage a qubit to calculate a
#    single random bit
device = cirq.Simulator()
def quantum_random_bit():
    result = device.run(circuit, repetitions=1)
    return result.measurements['m'][0][0]


#3.Implement classical logic side of hybrid code
# This code is identical to that in the Qiskit example.


def generate_random_bits(upper_limit: int) -> Iterable[int]:
    """
    Call the quantum operation as many times as there are bits needed to
    define the maximum of the range. For example, if max == 7, you need three
    bits to generate all the numbers from 0 to 7.
    """
    for _ in range(0, upper_limit.bit_length()):
        # Call the quantum operation and store the random bit in the list
        yield quantum_random_bit()

def convert_to_int(bit_string: Iterable[int]) -> int:
    """
    Interpret the bit string as a binary literal.
    """
    return int("".join(str(x) for x in bit_string), 2)

def quantum_random_int(upper_limit: int):
    """
    Using quantum RNG to generate a number from 0 to upper_limit
    """
    result = upper_limit + 1
    while result > upper_limit:  # i.e., redo QRNG if answer is too big
        result = convert_to_int(generate_random_bits(upper_limit))
    return result


# E.g., throw 100-sided quantum five times
for _ in range(5):
    print(1 + quantum_random_int(99))
```

Appendix E. Quixotic Jupyter Notebook

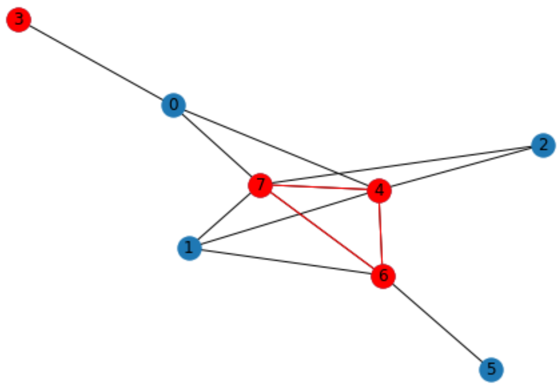
```
from quixotic.core import QuantumAnnealer

QuantumAnnealer.supported_tasks()
maximum_clique
minimum_vertex_cover
minimum_weighted_vertex_cover
maximum_independent_set
maximum_weighted_independent_set
maximum_cut
weighted_maximum_cut
structural_imbalance
traveling_salesperson

import networkx as nx

GRAPH_SEED = 1334
LAYOUT_SEED = 1971

def generate_graph(size):
    return nx.erdos_renyi_graph(size, p=0.5, seed=GRAPH_SEED)

def draw_graph(graph):
    positions = nx.spring_layout(graph, seed=LAYOUT_SEED)
    nx.draw(g, with_labels=True, pos=positions)

# defaults to local annealing simulator
def max_clique(graph, task, backend='local'):
    qo = QuantumAnnealer(graph, task=task, backend=backend).execute()
    return qo.results()

def draw_subgraph(graph, nodes):
    positions = nx.spring_layout(graph, seed=LAYOUT_SEED)
    sub = graph.subgraph(nodes)
    nx.draw(graph, pos=positions, with_labels=True)
    nx.draw(sub, pos=positions, node_color="r", edge_color="r")

def show_task_subgraph_for_random_graph(size, task, backend='local'):
    g = generate_graph(size)
    draw_subgraph(g, max_clique(g, task, backend=backend))

show_task_subgraph_for_random_graph(size=8, task='maximum_clique')
Executing locally.
```



```
show_task_subgraph_for_random_graph(size=8, task='minimum_vertex_cover')
        Executing locally.
```



```
show_task_subgraph_for_random_graph(size=8, task='minimum_vertex_cover', backend='dwave')
        Executing on D-Wave LEAP.
```



References

Bharti, Kishor, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alberin-Lea, Abhinav Anand, Matthias Degroote, et al. 2022. “Noisy intermediate-scale quantum algorithms.” *Reviews of Modern Physics* 94. doi:10.1103/RevModPhys.94.015004.

Johnson, M W, M H.S. Amin, S Gildert, T Lanting, F Hamze, and N Dickson. 2011. “Qunatum annealing with manufactured spins.” *Nature* 194-198. doi:10.1038/nature10012.

Kaye, Phillip, Raymond Laflamme, and Michele Mosco. 2007. *An Introduction to Quantum Computing.* Oxford University Press. doi:10.1093/oso/9780198570004.003.0004.

Mehta, Nihal. 2020. *Quantum Computing: Program Next-Gen Computers for Hard, Real-World Applications.* Raleigh, NC: The Pragmatic Bookshelf. https://pragprog.com/titles/nmquantum/quantum-computing/.

Shor, P.W. 1994. “Algorithms for quantum computation: discrete logarithms and factoring.” *Proceedings 35th Annual Symposium on Foundations of Computer Science.* IEE Comput. Soc. Press. 124-134. doi:10.1109/sfcs.1994.365700.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. REPORT DATE (DD-MM-YY) 00-06-22	2. REPORT TYPE Non-Standard	3. DATES COVERED (From – To)
4. TITLE AND SUBTITLE Developing in the Commercial Quantum Cloud		5a. CONTRACT NUMBER HQ0034-19-D-0001
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBERS
6. AUTHOR(S) Dale Visser, Arun S. Maiya		5d. PROJECT NUMBER C5211
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Institute for Defense Analyses 730 East Glebe Road Alexandria, VA 22305		8. PERFORMING ORGANIZATION REPORT NUMBER NS D-33113
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Institute for Defense Analyses 730 East Glebe Road, Alexandria, VA 22305		10. SPONSOR'S / MONITOR'S ACRONYM IDA
		11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: Dale Visser

14. ABSTRACT

There are several quantum computing cloud service providers available in the United States. This report gives a brief overview of what quantum computing is, and shows how to get started developing on the services available from IBM, D-Wave, Azure, Amazon and Google. An IDA-developed framework for leveraging quantum annealing systems for graph and optimization solutions is also described.

15. SUBJECT TERMS

Quantum Computing, Cloud Services, Software Development

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Institute for Defense Analyses
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	Unlimited	30	19b. TELEPHONE NUMBER (Include Area Code)