Quantum Software Development Tools

Table of contents

1. Introduction

This article provides an overview of currently available quantum development software and tools for Quantum Annealers (QAs), for gate-based quantum computers, comprising both Noisy Intermediate-Scale (NISQ) quantum computers and Fault-Tolerant Quantum Computers (FTQCs), and for Measurement-Based Quantum Computing (MBQC) quantum computers. This overview is neither complete nor very detailed in the descriptions of the various component types and associated products from quantum computing vendors and other sources, given the very large number of such artifacts currently available and the fact that these artefacts are continuously evolving (mostly by adding new features or supporting new quantum computing platforms). Furthermore, in many instances, the descriptions provided in this article only relate to gate-based quantum computing.

Note

It is assumed that the reader has reasonable knowledge of classical computing and some basic knowledge of quantum computing (as for example included in the article 'Quantum Computing Explained' published by the NOREA Taskforce Quantum Computing).

Quantum computers are complex systems that should typically not be directly handled by quantum software developers. Instead, to take advantage of the power of these devices, a stack of software layers that approximates the quantum computer to these developers is required (Figure 1.1 and Figure 1.2).

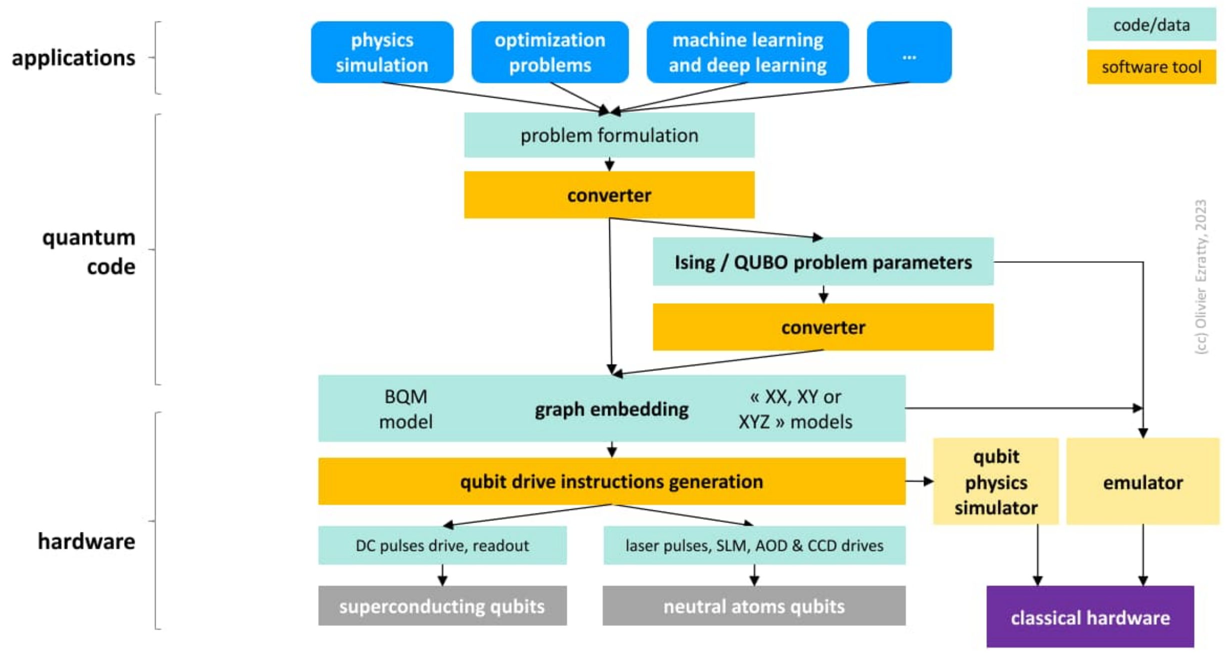
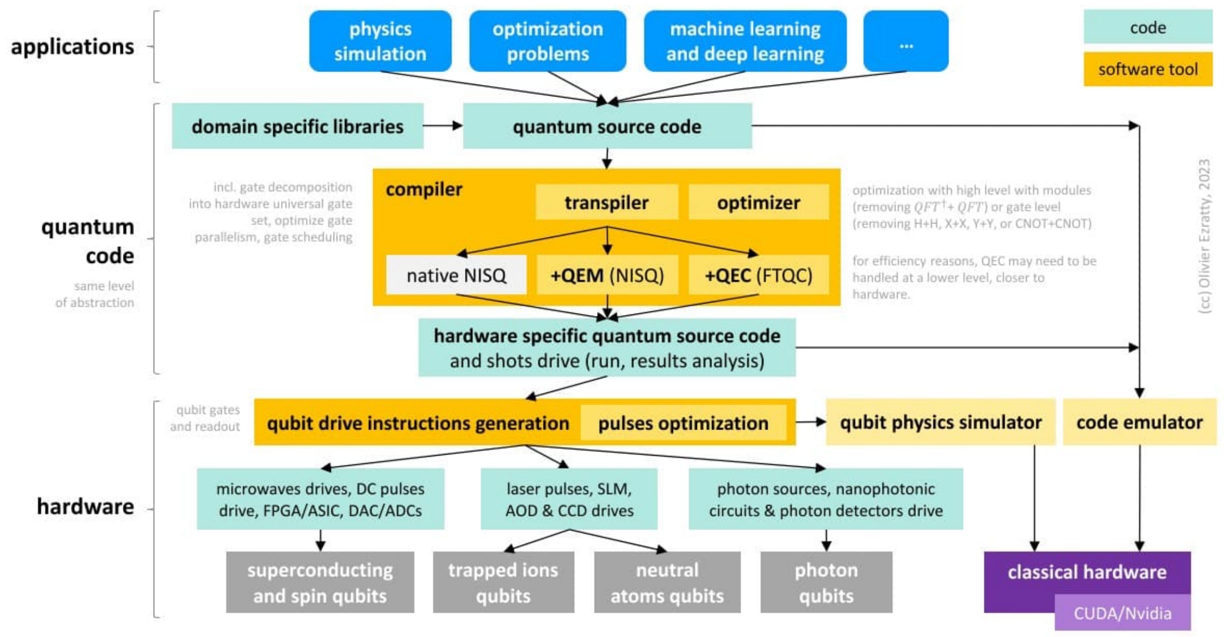


Figure 1.1: Analogue quantum computing stack



applications
physics simulation
optimization problems
machine learning and deep learning
...
code
software tool
domain specific libraries
quantum source code
quantum code
same level of abstraction
incl. gate decomposition into hardware universal gate set, optimize gate parallelism, gate scheduling
compiler
transpiler
optimizer
native NISQ
+QEM (NISQ)
+QEC (FTQC)
optimization with high level with modules (removing $QFT^{\dagger}+QFT$) or gate level (removing H+H, X+X, Y+Y, or CNOT+CNOT)
for efficiency reasons, QEC may need to be handled at a lower level, closer to hardware.
(cc) Olivier Ezratty, 2023
hardware specific quantum source code and shots drive (run, results analysis)
qubit gates and readout
qubit drive instructions generation
pulses optimization
qubit physics simulator
code emulator
hardware
microwaves drives, DC pulses drive, FPGA/ASIC, DAC/ADCs
laser pulses, SLM, AOD & CCD drives
photon sources, nanophotonic circuits & photon detectors drive
superconducting and spin qubits
trapped ions qubits
neutral atoms qubits
photon qubits
classical hardware
CUDA/Nvidia

Figure 1.2: Gate-based quantum computing stack

Programming for quantum computers is based on very different concepts than programming for classical computers, and as such requires new computing languages and associated development software and tools[1]. It is however important to note that most of these are in fact hybrid classical-quantum artefacts, where the quantum computer is considered to constitute a so-called “quantum accelerator” (a concept similar to GPUs and AI accelerators). This in particular the case for cloud-based quantum computing and for hybrid classical-hybrid HPC computing environments.

When developing quantum applications, the problem to be solved by the use of quantum computing must be determined first. After the problem has been analysed and properly understood, the next step is to design the quantum algorithm and then design and specify the corresponding quantum circuit, i.e. the set of qubits, the sequence of operations (quantum gates) to be performed on these qubits, and the qubit measurements yielding the (classical) outcome of the quantum computation. Many different quantum software development tools can be used for this purpose. Most of these tools can produce the source code of the quantum circuit according to one of the available quantum programming languages (see Chapter 2).

After the quantum source code has been obtained, it is usually integrated into a classical program by embedding it in a program developed with a quantum programming language derived from a classical programming language.

[1] In contrast to the early stages of classical computing, software for quantum computers has already considerably evolved, because of the existence of personal computers, remote (cloud) access, open-source communities etc., none of which were available during the early stages of classical computing.

The resulting program source code must then be compiled into machine language for execution on the physical classical/quantum computing platform. There are two main options for doing this:

1. Execute the compiled code on a local (on-premises) classical/quantum computer system or on a quantum emulator running on a local classical computer (when a quantum circuit has been designed, it is usually first evaluated on a quantum circuit emulator, which provides the quantum computing results in different formats, e.g. histograms with the probabilities of measuring each of the possible states of the qubits).

2. Execute the compiled code using the Quantum Computing-as-a-Service (QCaaS) service provided by a quantum computing vendor or service provider. In this case, the most common option is sending the source code or the compiled quantum machine code to a remote quantum service, where it is enqueued for compilation and/or execution on the classical/quantum computing infrastructure or on a quantum emulator provided by the vendor or service provider. After execution of the quantum program, a report with the results of the computation is sent back to the requestor.

The verification and certification of quantum algorithms and the result of their execvution is an important topic. Verification deals with verifying that the code will run as expected (it responds to the question: are we building product the product right?). Validation concerns the program output and making sure it works as planned (it responds to the question: are we building the right product?).

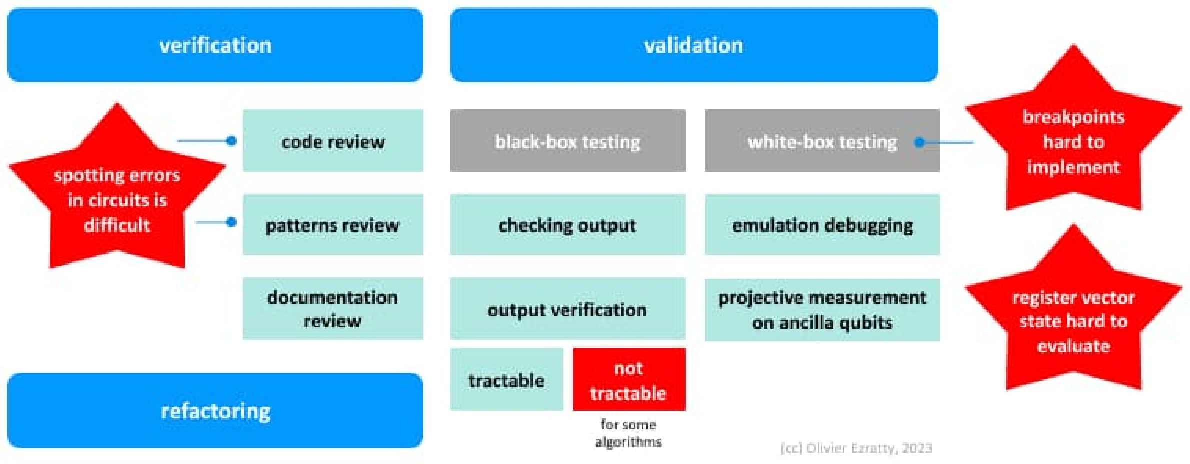


Figure 1.3: Quantum software quality assurance

A quantum circuit is not easy to debug and it will certainly require new debugging tools and approaches because many quantum code bugs are “quantum” in nature and are not easy to spot with traditional methods.

For the moment, simple quantum circuits can be analysed and debugged with a quantum emulator running on a classical computer, to understand how the qubit register vector state evolves step-by-step. But when quantum computers will be available with a large number of qubits[2], beyond any classical emulation capacity, other means will have to be used.

Generally, quantum developers work in a user-friendly environment, in which a quantum circuit is coded or designed with drag-and-drop visual tools. Such high-level quantum source code is not yet directly executable by the quantum processor and must first be compiled through a compiler that adapts and optimises the quantum circuit to the quantum processor's hardware characteristics and its supported set of qubits, quantum gates (operations) and qubit measurements. This is sometimes done by first converting the quantum circuit's high-level source code to an equivalent intermediate representation in a low-level quantum assembly language, and then compile it into an executable quantum program.

The current state of play in quantum computer software includes many products in development both commercially and academically, several of them being open-source efforts. With the recent industry push toward larger quantum computers and prototypes (including availability on public clouds for broad use), there is an increased awareness of the need for full-stack quantum computing software in order to encourage usage and nurture an ever increasing quantum developer community. Thus, it is reasonable to expect that quantum programming languages and software ecosystems will receive considerable attention and may see significant changes in coming years.

[2] Current "guestimates": 50 to 100 qubits.

2. Quantum programming languages and associated tools

Several levels of abstraction are involved in a typical full quantum software/hardware stack (see Figure 2.1 for an example) and therefore, several layers of quantum development software and tools can be distinguished.

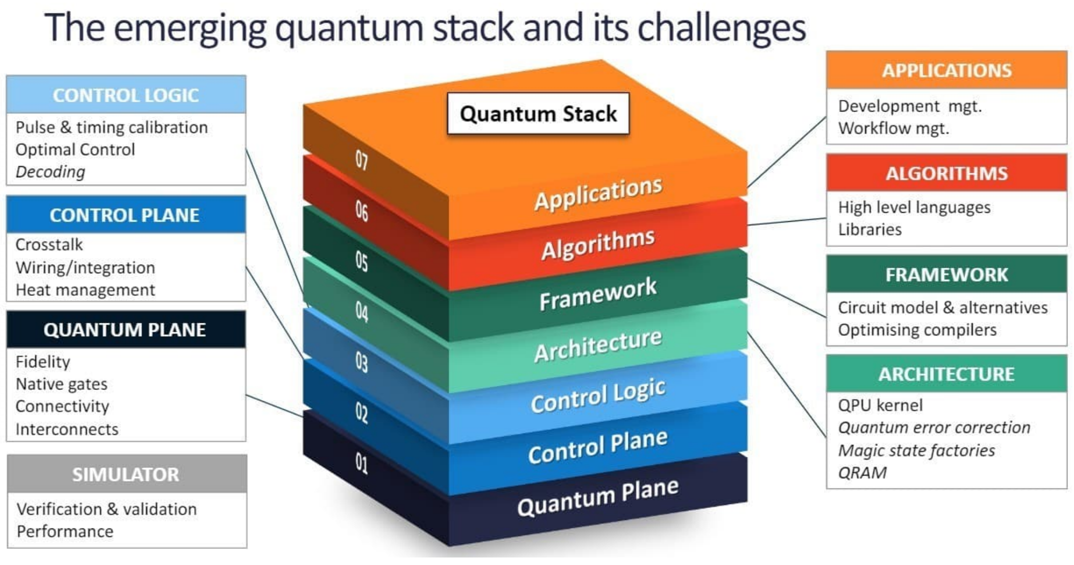


Figure 2.1: Quantum software/hardware stack (source: Fact Based Insight)

High-level quantum programming languages and high-level quantum compilers enable a developer to program quantum algorithms or quantum circuits, while shielding the developer from the details of the underlying quantum hardware platform. A distinction can be made between *quantum algorithm programming languages* and *quantum circuit programming languages* (several high-level quantum programming languages support both).

Note

In stark contrast with classical computing where bits are only used at the lowest levels of abstraction, quantum computing is based on qubits and operations on qubits (quantum gates) at all levels of abstraction (see Figure 2.2).

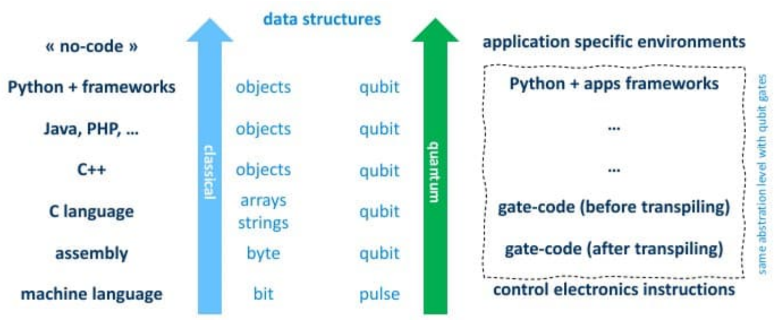


Figure 2.2: Computing language abstraction levels (source: Olivier Ezratty 2023)

Low-level quantum programming languages (aka quantum assembly languages) and low-level quantum compilers (aka quantum assemblers) enable a developer to specify quantum circuits in detail, interacting directly with a specific quantum hardware platform by providing the means to specify the physical instructions necessary to execute the quantum circuit.

<u>Note</u>
It is not always obvious whether a quantum computing language is to be classified as high-level or low-level. Several quantum programming languages could fit in both categories, depending on one's definition of what is "low" and what is "high"[3].

Quantum programming languages are either imperative languages, declarative languages or functional languages:

- *Imperative programming* is a programming paradigm that specifies statements to change a program's state. In much the same way that the imperative mode in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates step-by-step, rather than on high-level descriptions of its expected results.

- *Declarative programming* is a programming paradigm that expresses the logic of a computation without describing its control flow. Many languages that apply this style attempt to minimise or eliminate side effects by describing what the program must accomplish in terms of the problem domain, rather than describing how to accomplish it as a sequence of the programming language primitives (the "how" being left up to the language's implementation).

- *Functional programming* is a programming paradigm where a program is constructed by applying and composing functions without describing the program's control flow. Function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

Quantum compilers translate quantum program source code into some kind of "quantum machine instructions". These compilers first transform the universal quantum gates supported by the quantum programming language into the particular physical quantum gates supported by the quantum computer (the "basis gates" aka "native gates")[4] and then into the sequences of (electronic or photonic) control pulses operating on the physical qubits of the quantum processor (Figure 2.3).

[3] This is not unlike the geographical notions of "high" and "low". Very few Europeans, except maybe the Danish, would for example classify the Dutch Cauberg hill as a mountain like Dutchmen do.

[4] This process is often called "transpilation".

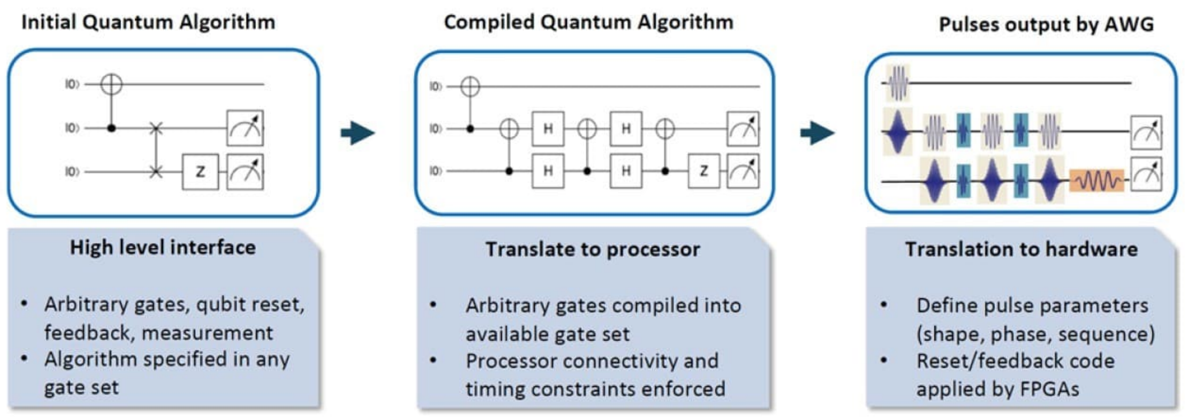

Initial Quantum Algorithm
Compiled Quantum Algorithm
Pulses output by AWG
High level interface
• Arbitrary gates, qubit reset, feedback, measurement
• Algorithm specified in any gate set
Translate to processor
• Arbitrary gates compiled into available gate set
• Processor connectivity and timing constraints enforced
Translation to hardware
• Define pulse parameters (shape, phase, sequence)
• Reset/feedback code applied by FPGAs

Figure 2.3: Quantum program compilation (source: Bert de Jong 2019)

Quantum compilers typically provide for optimisation of the generated quantum program source code. They often also provide advanced features such as reducing the number of quantum gates to create shallower quantum circuits, integration of Quantum Error Correction (QEC) code or distribution of the quantum computations across multiple QPUs.

Note
In contrast to classical computing, there is no such thing as an Operating System (OS) for quantum computers, though a few vendors use this term to denote the qubits control subsystem, which is however very different from a classical OS since it is inseparable from the QPU subsystem (the quantum computer being the combination of both subsystems). On the other hand, one could argue that the "orchestration" part of the qubits control software (which generally runs on a classical computer) is the quantum computer's operating system, because it has a few things in common with a classical OS.

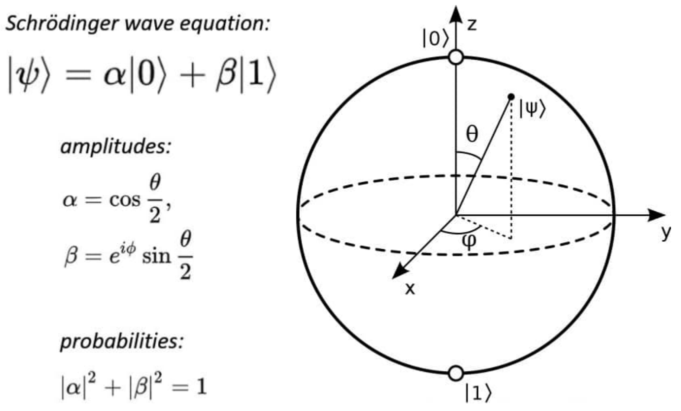
Some high-level quantum compilers generate intermediate quantum program source code (aka quantum assembly code) that is then input to a low-level quantum compiler (aka quantum assembler) to generate executable quantum machine code.

Quantum scripting languages are used to program quantum algorithms or quantum circuits in "text mode". Most quantum scripting languages are able to combine classical computing programming with quantum computing programming. Many of them are implemented as quantum extensions of classical computer languages such as C, C++, C#, Java, PHP and Pyhton.

Quantum graphical programming tools provide the means to visually define the sequence of quantum gates and qubit measurements for the specification of a quantum circuit (this functionality is embedded in many quantum scripting languages). Such tools can be used to specify relatively simple quantum circuits, which should fine for the current NISQ generation of quantum computers.

Quantum graphical programming tools can sometimes emulate a quantum computer and visualise the status of its qubits with either Bloch spheres (Box 2.1), qubit register states (Box 2.2) or Density Matrixes (DMs, see Box 2.3).

The Bloch sphere is a geometrical representation of the pure quantum state space of a two-level quantum mechanical system, e.g. a qubit. The Bloch sphere has antipodal points corresponding to a pair of mutually orthogonal quantum state vectors. The north and south poles of the Bloch sphere correspond to the standard basis vectors $|0\rangle$ and $|1\rangle$ of the qubit.


Schrödinger wave equation:
$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$
amplitudes:
$\alpha = \cos\frac{\theta}{2}$,
$\beta = e^{i\phi}\sin\frac{\theta}{2}$
probabilities:
$|\alpha|^2 + |\beta|^2 = 1$
|0⟩
z
|ψ⟩
θ
φ
y
x
|1⟩

Box 2.1: Bloch sphere

In a quantum computer, qubits are organised in registers, like the bit registers in today's classical processors but not quite the same though. One key difference is that a quantum computer has only one register and not many as current classical processors.

The most important difference between a qubit register and a classical bit register is the amount of information that can be manipulated simultaneously. In classical computers, the bit registers store bitstrings, integers or floating-point numbers on which elementary logical or arithmetic operations are performed. In contrast, a register of n qubits is a vector in a 2^n dimensional space of complex numbers. These complex numbers are the amplitude of each computational quantum state and the total of their squared norms equals 1 since these are probabilities. Hence the dimensionality of a n-qubit register is exponentially larger than that of a n-bit register.

Box 2.2: Qubit register

A Density Matrix (DM) is a matrix that describes the quantum state of a quantum system. It allows for the calculation of the probabilities of the outcomes of any measurement performed upon this system.

A DM is a generalisation of the more usual quantum state vector or quantum wavefunction: while these can only describe a “pure” quantum state, a DM can also describe a “mixed” quantum state, into which the pure quantum state “decoheres” by interactions with the environment (this is for example the case with “noisy” qubits).

Box 2.3: Density Matrix (DM)

See Appendix A for a brief description of a representative selection of quantum programming languages.

Due to the difficulty of building quantum computers and the limited access to the few real quantum computers currently available, a growing body of *quantum emulators* has emerged to assist in the tasks of designing quantum algorithms and associated quantum circuits. When new quantum algorithms are being developed, it is generally advisable to conduct the initial proof-of-concept validation with a quantum emulator.

Note

Quantum emulators are often called "quantum simulators" but this is not the right term (Box 2.4) and it creates confusion with quantum simulator systems, i.e. analogue quantum computers used for simulating quantum mechanical systems[5] (Figure 2.4). In this article, the term "quantum emulator" is used, except in cases where the word "simulator" is part of the name of a software artefact.

Simulation is the imitation of the operation of a system over time and is based on a model which represents its key characteristics and behaviours. Simulation is used for various purposes, e.g. testing, optimising, performance tuning, etc. of technology being designed, the study of physical systems based on scientific modelling of these systems, etc.

Emulation is a technique that enables one system (the emulator) to behave (almost) exactly like another system (the target). The Church-Turing thesis implies that, in theory, any computing environment can be emulated within any other computing environment, assuming memory limitations are ignored.

Box 2.4: Simulation versus emulation

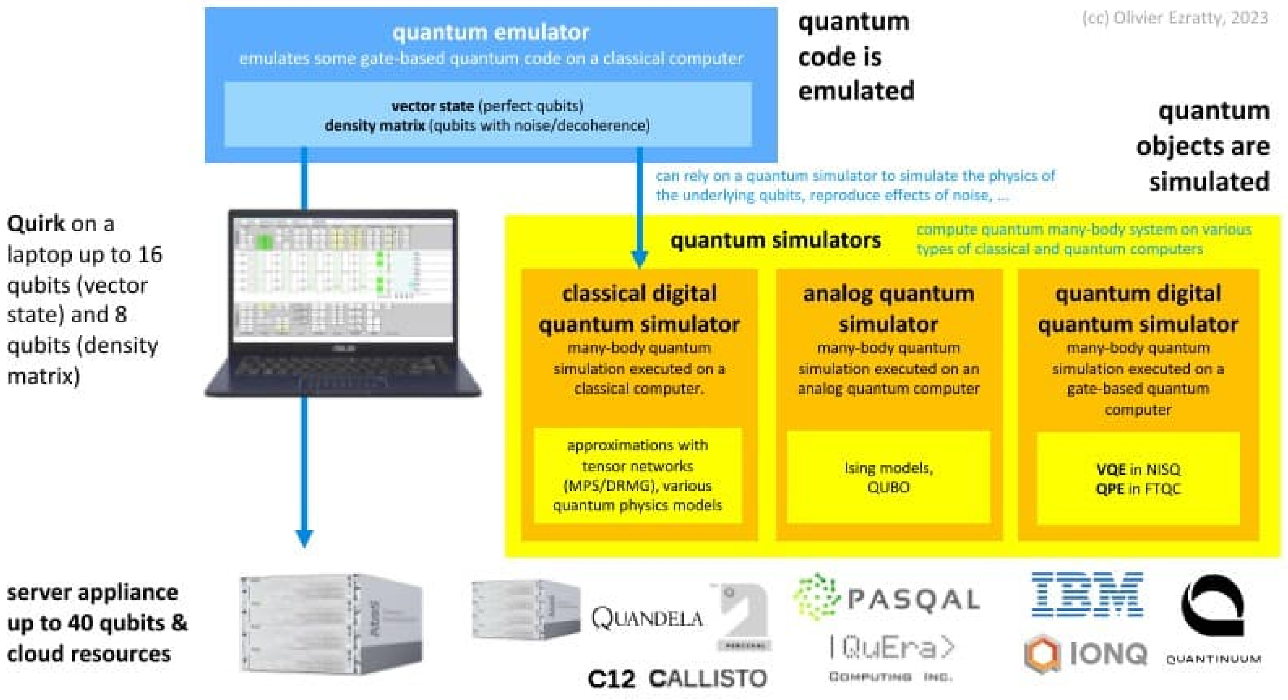

quantum emulator
emulates some gate-based quantum code on a classical computer
vector state (perfect qubits)
density matrix (qubits with noise/decoherence)
quantum code is emulated
(cc) Olivier Ezratty, 2023
quantum objects are simulated
can rely on a quantum simulator to simulate the physics of the underlying qubits, reproduce effects of noise, ...
Quirk on a laptop up to 16 qubits (vector state) and 8 qubits (density matrix)
quantum simulators
compute quantum many-body system on various types of classical and quantum computers
classical digital quantum simulator
many-body quantum simulation executed on a classical computer.
approximations with tensor networks (MPS/DRMG), various quantum physics models
analog quantum simulator
many-body quantum simulation executed on an analog quantum computer
Ising models, QUBO
quantum digital quantum simulator
many-body quantum simulation executed on a gate-based quantum computer
VQE in NISQ
QPE in FTQC
server appliance up to 40 qubits & cloud resources
QUANDELA
C12 CALLISTO
PASQAL
QuEra COMPUTING INC.
IBM
IONQ
QUANTINUUM

Figure 2.4: Quantum emulation vs. quantum simulation

[5] These are the quantum computers that Richard Feynman had in mind when he introduced the term "quantum computer" in 1981.

There are several types of quantum emulators, including:

- Density Matrix (DM)-based: using DMs for representing mixed quantum states (see Box 2.3), which allows for the emulation of imperfect qubits (i.e. qubits affected by noise and decoherence) but is also the most resources-hungry emulation method.

- State Vector (SV)-based: representing pure quantum states by quantum state vectors, which only allows for the emulation of perfect qubits (i.e. qubits that are not affected by noise and decoherence) but is far less resources-hungry.

- Tensor Network (TN)-based: using TN compression techniques, which significantly reduces the complexity of the emulation software and also allows for the distribution of the emulation program execution over a cluster of classical computing nodes.

See Appendix B for a brief description of a representative selection of quantum emulators.

The amount of computing resources required for the emulation of a quantum circuit heavily depends on the number of qubits to be emulated and also on the depth of the quantum circuit (i.e. the number of quantum gates). The main limitation of classical computers for quantum circuit emulation is the amount of available computer memory (RAM) rather than CPU capacity.

Resource estimators are software tools designed to estimate the quantum computing hardware resources using as inputs a given algorithm and the various hardware characteristics. At the moment, there are only a few of these tools available:

- Google developed the Cirq-FT tool;

- Microsoft developed a tool to estimate the qubit numbers, T-gate count and execution time for a given quantum algorithm targeting an FTQC platform;

- USC, UCSB and Berkeley developed the QuRE tool;

- Zapata Computing, Aalto University, IonQ, the University of Technology Sydney and the University of Texas at Dallas developed BenchQ, an open-source resource estimation tool for chemistry industry quantum computing applications (part of DARPA Quantum Benchmarking Program Phase II).

Quantum computer benchmarking tools, which are used to evaluate and compare existing quantum computers. These benchmarking tools already abound and are very diverse; they are not described in this article.

Quantum Software Development Kits (QSDKs) provide collections of tools to develop, test and execute quantum programs. They provide the means to prepare the quantum circuits to be run using either on-premise or cloud-based (QCaaS) quantum computers and quantum emulators.

See Appendix C for a representative selection of prevalent QSDKs.

Many quantum software development tools are open-sourced and free to install and use, their differentiation being the available support, documentation and tutorials. In practice, however, commercial quantum application developers typically don’t use these tools. Instead, they use the quantum languages and quantum computing software development platforms provided by commercial quantum computing vendors (see Chapter 3). They become thus locked into these vendors’ so-called "full-stack" approaches, which may be open-sourced in principle but are often proprietary in practice.

3. Quantum computing vendor software development platforms

3.1. Introduction

Quantum software development platforms from “incumbent” quantum computer vendors are mostly focused on their own quantum computers (see Figure 3.1 for a few examples), while the quantum software development platforms from “full-stack” quantum software vendors (often start-ups) are more open, more flexible and support multiple quantum computer platforms and associated software environments (see Figure 3.2 for a few examples).

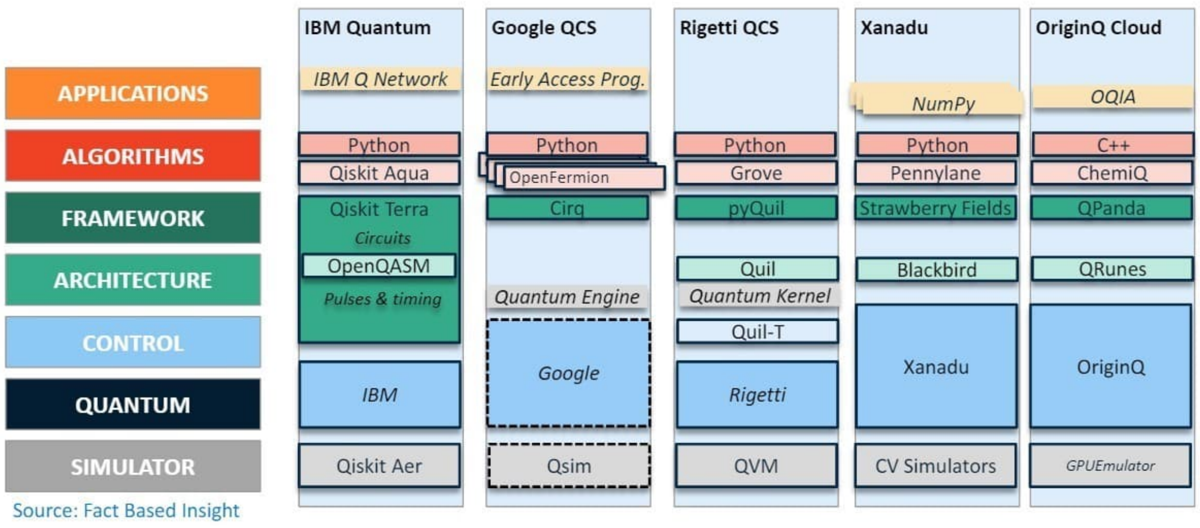


Figure 3.1: Quantum computer vendor software development platforms

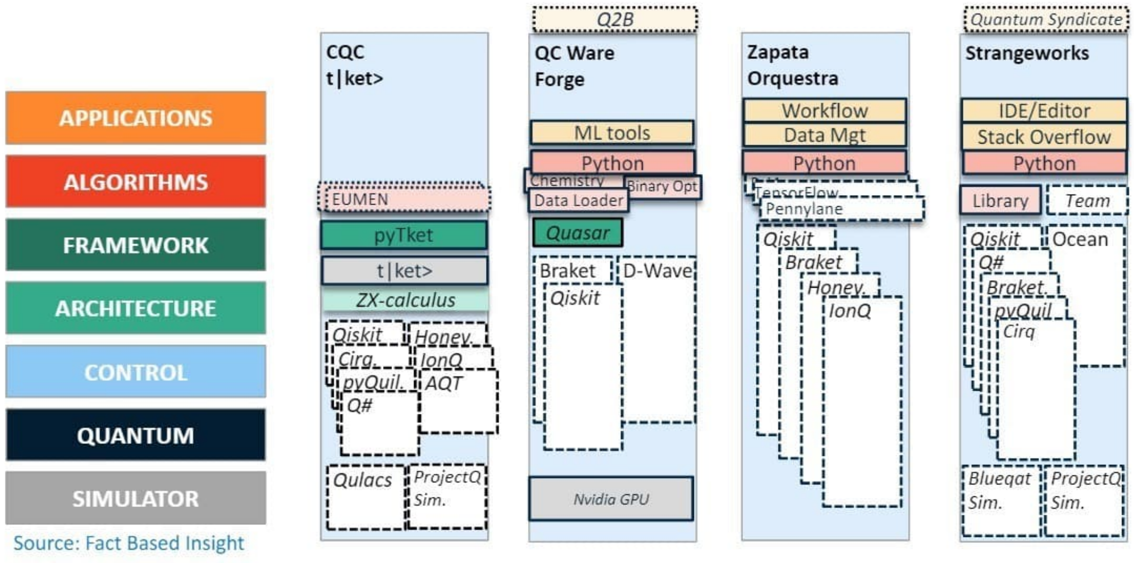


Figure 3.2: Full-stack quantum software vendor software development platforms

Microsoft is an odd man out because they have not yet succeeded in building their own (topological qubit-based) quantum computer, and because their Azure Quantum cloud service provides access to competitors' quantum computing platforms (see § 3.7).

To some extent, this is also the case for Google Quantum AI, as their Google Cloud Marketplace provides access to competitors' quantum computing platforms, while access to Google Quantum AI's own quantum computers is currently only granted to selected parties (see § 3.4).

The quantum software company Cambridge Quantum Computing (CQC) merged in 2021 with the quantum computer manufacturer Honeywell Quantum Solutions (HQS) to form Quantinuum. This explains why Quantinuum's quantum software development platform (see § 3.8), which was developed by CQC, differs from that of other "incumbent" quantum computer vendors.

The following sections provide a brief description of prevalent quantum computing vendor software development platforms (in alphabetical order of vendor names).

3.2. Amazon

Amazon Braket (figure 3.3) has been designed to be quantum hardware platform agnostic, removing the need to use different quantum programming tools for each type of quantum computer.

Amazon allows users to bring their own quantum development environment or to use Amazon Braket. Also, Amazon Braket natively supports PennyLane.

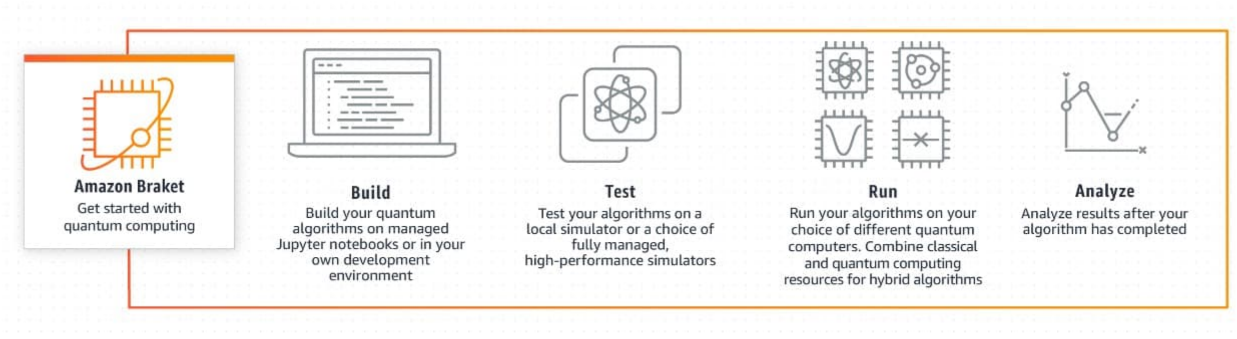

Amazon Braket
Get started with quantum computing
Build
Build your quantum algorithms on managed Jupyter notebooks or in your own development environment
Test
Test your algorithms on a local simulator or a choice of fully managed, high-performance simulators
Run
Run your algorithms on your choice of different quantum computers. Combine classical and quantum computing resources for hybrid algorithms
Analyze
Analyze results after your algorithm has completed

Figure 3.3: Quantum software development with Amazon Braket SDK (source: AWS)

AWS Marketplace provides AWS customers with QCaaS access to quantum computing technologies from multiple quantum hardware providers for quantum software development using its Amazon Braket SDK:

- |QuEra⟩: access to Aquila neutral atom qubit-based quantum computers;

- D-Wave Systems: access via D-Wave System's Leap QCaaS service (see § 3.3) to D-Wave 2000Q and Advantage superconducting flux qubit-based quantum annealers and to D-Wave System's hybrid solvers;

- IonQ: direct access or access via IonQ's Quantum Cloud (see § 3.6) to Harmony, Aria and Forte trapped-ion qubit-based quantum computers;

- OQC: access to superconducting coaxmon qubit-based quantum computer;

- Quantinuum: access to H1 and H2 trapped-ion qubit-based quantum computers;

- Rigetti Computing: access to Aspen-M Series superconducting flux qubit-based quantum computers;

- Xanadu: access to X-Series Measurement-Based Quantum Computing (MBQC) quantum computers.

Amazon Braket also provides access to a choice of quantum emulators, including the free local emulator in the Amazon Braket SDK and three fully managed on-demand emulators: State Vector 1 (SV1), Density Matrix 1 (DM1) and Tensor Network 1 (TN1).

Amazon Braket Hybrid Jobs provides additional flexibility to use embedded emulators, designed with high performance and ultra-low latency in mind. These emulators can be embedded within the same job container as the quantum application code. Amazon Braket Hybrid Jobs supports embedded emulators from PennyLane or the option to embed one's own circuit emulator as a container (using the "bring-your-own-container" feature).

3.3. D-Wave Systems

D-Wave Systems' Ocean (Figure 3.4) is a Python-based quantum Integrated Development Environment (IDE).

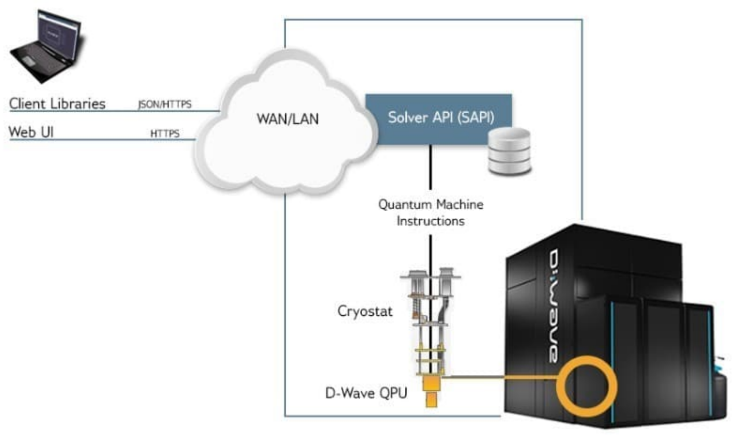


Figure 3.4: D-Wave's quantum software environment (source: D-Wave Systems)

D-Wave System's Leap real-time quantum cloud service (QCaaS) provides real-time access to its D-Wave 2000Q and Advantage quantum annealer platforms, constrained quadratic model solver, hybrid solvers, Ocean QSDK, live code, demos and learning resources.

D-Wave Systems' quantum annealers can also be accessed via the AWS Marketplace (see § 3.2).

3.4. Google Quantum AI

Google Quantum AI's Cirq (Figure 3.5) is an open-source Python software library. Cirq is designed to emulate universal gate quantum computers and also provides the facility to execute a program on a quantum computer.

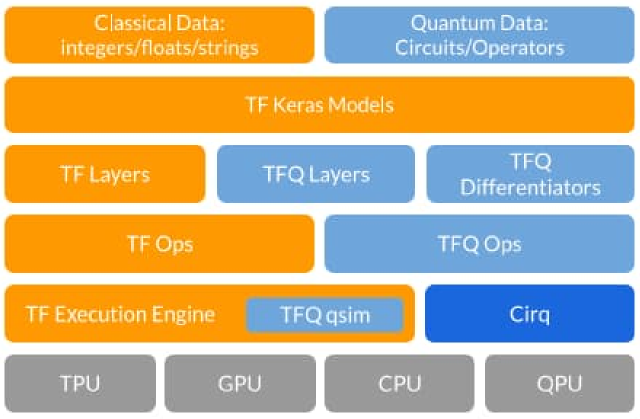


Figure 3.5: Google Cirq quantum software framework (source: M. Broughton et al.)

The Cirq ecosystem supports a vast set of external libraries that can be utilised for application specific development, such as:

- TensorFlow Quantum (TFQ): a hybrid quantum-classical Quantum Machine Learning (QML) library for rapid prototyping of hybrid quantum-classical Machine Learning (ML) models. The PennyLane Cirq plugin integrates the Cirq quantum computing platform with PennyLane's QML capabilities.
- OpenFermion: supports the creation of quantum algorithms for chemistry and material sciences.
- Forge: supports the creation of quantum algorithms for data science, finance, etc.
- ReCirq is a library of research experiments using Cirq. The ReCirq repository contains code for Google's flagship experiments, enabling one to reproduce and extend cutting edge quantum computing research.

Google Cloud Marketplace provides QCaaS access via Google Cloud to IonQ trapped-ion qubit-based quantum computers and to Cirq, qsim and PyQVM quantum emulators.

Quantum Computing Service (QCS) gives customers access to Google Quantum AI's FoxTail, Bristlecone[6] and Sycamore transmon qubit-based quantum computers. Programs that are written in Cirq can be sent to run on a quantum computer in Google's quantum computing lab in Santa Barbara, CA. No public access to the service is available at this time, access is currently only granted to those on an approved list.

3.5. IBM Q

The main elements of IBM Q's quantum software architecture (Figure 3.6) are:

- Model Developers and Algorithm Developers for building quantum applications: includes Machine Learning, Natural Science and Optimization;
- Quantum Serverless: includes intelligent orchestration, circuit knitting toolbox and circuit libraries (a collection of well-studied quantum circuits);
- Kernel Developers: Circuits (i.e. quantum circuit development using Qiskit) and Qiskit Runtime (cloud service that runs a Qiskit program remotely as a process, passing the input from the user, and handling the connectivity between the Qiskit program, the user and the QPU); including functionality for:
    - dynamic circuits: mid-circuit measurements that affect the control flow of quantum gate execution later in the quantum circuit (aka feed-forward operations);
    - threaded primitives;
    - error suppression and mitigation;
    - error correction;
- System Modularity: underlying IBM Q QPUs and quantum emulators.

IBM Q offers several quantum emulators. The ibm_qasm_simulator emulates up to 32 qubits as if it was a real quantum device (including measurement pseudo-randomness) and also supports noise models injection. The Qiskit Aer emulators provide access to a 32 qubits quantum state vector-based emulator, as well as 63- and 100-qubit emulators, and also a 5,000-qubit stabilizer mode (Clifford gate group only[7]) emulator, with some restrictions on the quantum gate sets.

[6] However, Google's Bristlecone quantum computer never saw the day of light!

[7] Quantum circuits using only quantum gates in the Clifford group can be emulated in polynomial time on a classical computer according to the Gottesman-Knill theorem.

Qiskit Aqua provides a library of quantum algorithms and components to build quantum applications and leverage NISQ quantum computers.

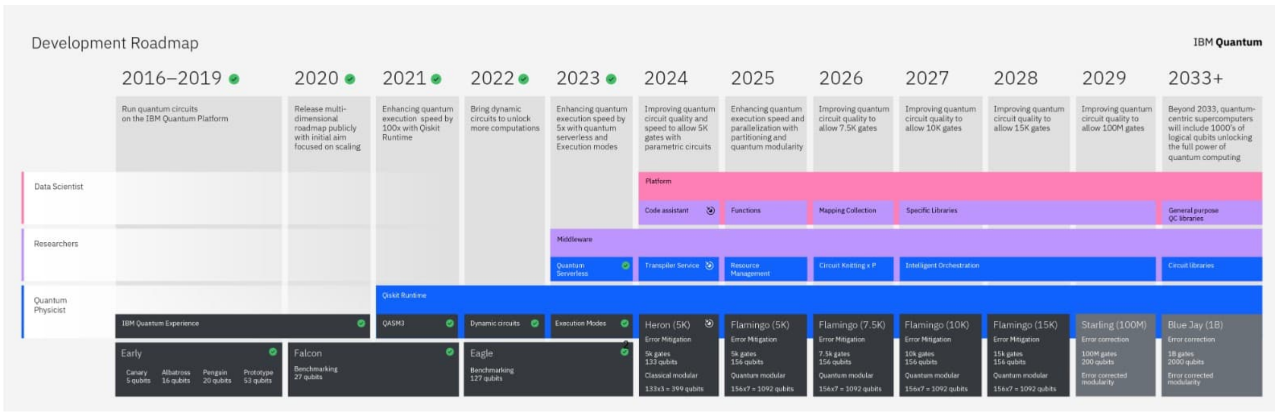


Figure 3.6: IBM Q Quantum software roadmap (source: IBM Q)

IBM Q provides the following options for QCaaS access to its quantum computers:

- Open Plan via IBM Quantum: free unlimited access to quantum simulators and free limited access (up to 10 minutes a month) to 7-qubit and 127-qubit Eagle IBM Q QPUs;
- Pay-As-You-Go Plan via IBM Cloud: paid access to quantum simulators, 27-qubit Falcon and 127-bit Eagle QPUs;
- Premium Plan via IBM Cloud: paid access to quantum simulators, 27-qubit Falcon, 127-bit Eagle and 433-bit Osprey QPUs based on reserved capacity.

IBM Q also provides a Dedicated Service powered by a dedicated co-hosted quantum computer that is managed by IBM Q.

3.6. IonQ

IonQ's cloud quantum computing platform Quantum Cloud allows customers to run quantum programs remotely (QCaaS) on IonQ's quantum hardware. With access to IonQ QPUs, noisy emulators, as well as an ideal state emulator, Quantum Cloud supports all stages of quantum algorithm development in one seamless API. Quantum Cloud is compatible with major QSDKs such as Cirq, PennyLane, ProjectQ, QDK, Qiskit and TKET.

Quantum Cloud can be accessed via the AWS Marketplace (see § 3.2), Google Cloud Marketplace (see § 3.4) and Azure Quantum Cloud (see § 3.7). IonQ also provides selected partner organisations with direct access to its cloud quantum computing platform.

Quantum programs are typically submitted via IonQ's API as language-agnostic JSON, but cQASM, OpenQASM and Quipper formats are also supported. One can simply use curl on the command line, but JavaScript, Python or a fully-featured SDK is usually preferred.

3.7. Microsoft

Microsoft's Quantum Development Kit (QDK) interfaces with the Azure Quantum service for building quantum programs that run on quantum hardware and quantum emulators that are available in Azure Quantum (Figure 3.7 and Figure 3.8). QDK is built into the Azure Quantum portal, where one can develop programs using the free hosted Jupyter Notebooks.

QDK includes the quantum programming language Q# and supports Cirq and Qiskit. It also contains some components that can be used standalone, independently from the Azure Quantum service:

- Q# language and quantum libraries (all open-source);
- quantum emulators that emulate current and future quantum machines, to run and debug quantum algorithms written in Q#;
- extensions for Visual Studio (VS) and Visual Studio Code (VSC) and integration with Jupyter Notebooks.

Azure Quantum is Azure's cloud quantum computing service (QCaaS). It supports a diverse set of quantum solutions and technologies. Azure Quantum ensures an open, flexible, and future-proofed path to quantum computing that allows running quantum programs (Cirq, Q# or Qiskit) on multiple quantum hardware platforms.

Azure Quantum supports quantum computers from IonQ, Pasqal, QCI, Quantinuum and Rigetti Computing. Azure Quantum also supports various quantum emulators: IonQ's GPU-accelerated quantum emulator, Quantinuum's H1 and H2 emulators, and Rigetti Computing's PyQVM emulator.

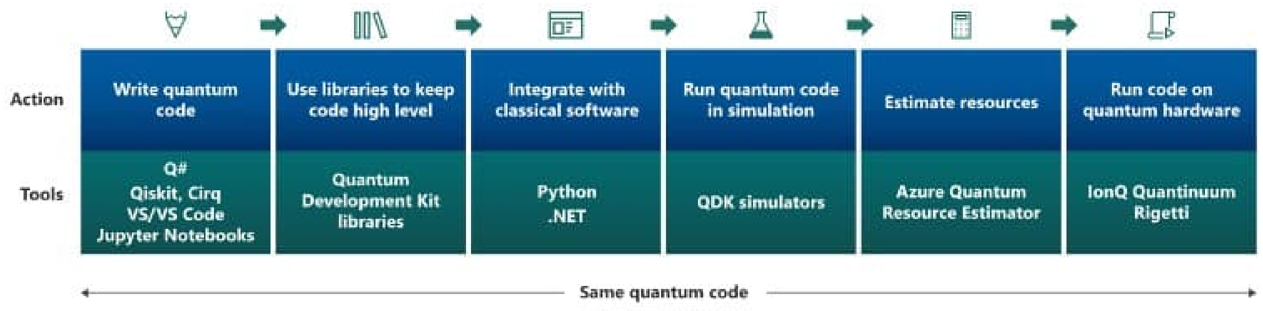

Action
Tools
Write quantum code
Use libraries to keep code high level
Integrate with classical software
Run quantum code in simulation
Estimate resources
Run code on quantum hardware
Q# Qiskit, Cirq VS/VS Code Jupyter Notebooks
Quantum Development Kit libraries
Python .NET
QDK simulators
Azure Quantum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

Figure 3.7: Quantum program stages and corresponding QDK tools (source: Microsoft)

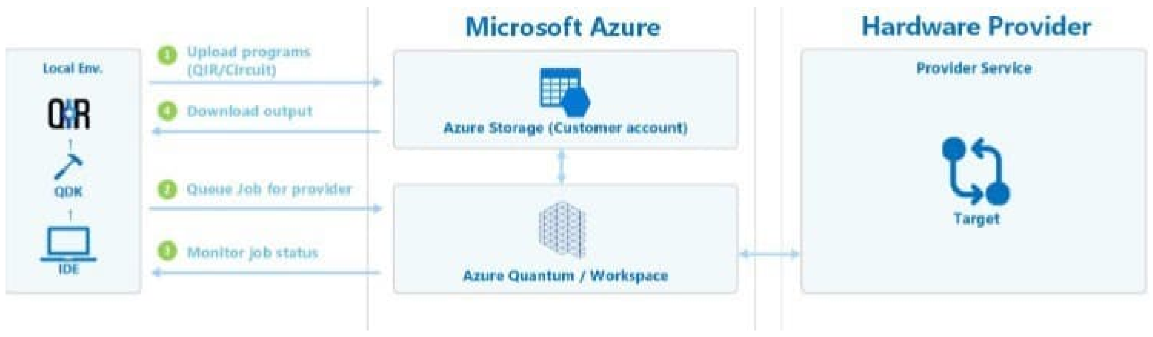


Figure 3.8: Azure Quantum workflow (source: Microsoft)

3.8. Quantinuum

TKET (Figure 3.9) is Quantinuum's software for developing, optimising and executing platform-agnostic quantum circuits.

Quantinuum offers access to their trapped-ion qubit-based quantum computers and emulators, accessible via their API and User Portal (QCaaS). Users are able to submit jobs that run remotely on Quantinuum’s quantum systems from a local Python development environment.

The following PyTKET backends are available: AQT QPUs and emulators, Cirq emulators, IBM Q QPUs and emulators, IonQ QPUs, IQM QPUs and emulators, ProjectQ emulator, Rigetti Computing QPUs, and Quantinuum H1-1 and H1-2 QPUs and emulators. Quantinuum quantum emulators include quantum state vector-based emulators, density matrix-based emulators and other specialised quantum emulators.

Quantinuum quantum computers and emulators can also be accessed via AWS Marketplace (see § 3.2) and Azure Quantum (see § 3.7).

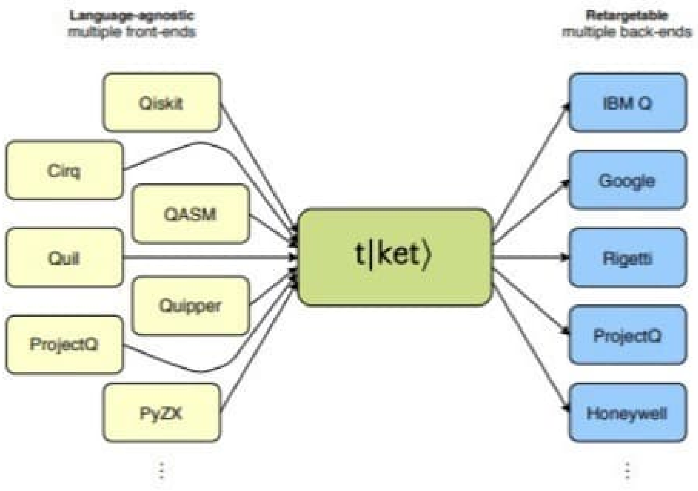


Figure 3.9: TKET quantum software architecture (source: Quantum Zeitgeist)

3.9. Rigetti Computing

Rigetti Computing’s Forest (Figure 3.10) is a set of software tools that allows to write quantum programs in Quil, then compile and run them via Quantum Cloud Services (QCS) or a quantum emulator.

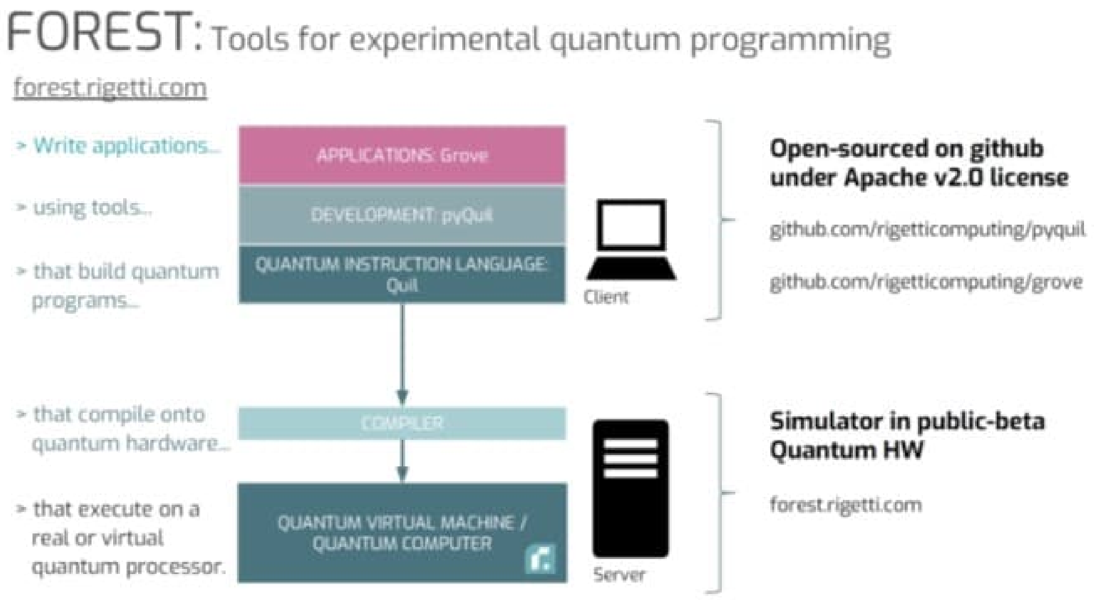

FOREST: Tools for experimental quantum programming
forest.rigetti.com
> Write applications...
> using tools...
> that build quantum programs...
> that compile onto quantum hardware...
> that execute on a real or virtual quantum processor.
APPLICATIONS: Grove
DEVELOPMENT: pyQuil
QUANTUM INSTRUCTION LANGUAGE: Quil
COMPILER
QUANTUM VIRTUAL MACHINE / QUANTUM COMPUTER
Client
Server
Open-sourced on github under Apache v2.0 license
github.com/rigetticomputing/pyquil
github.com/rigetticomputing/grove
Simulator in public-beta
Quantum HW
forest.rigetti.com

Figure 3.10: Forest quantum software stack (source: Rigetti Computing)

Forest comprises the following components:

- pyQuil: Python library for building and executing programs written in the Quil quantum programming language;
- quilc: optimising Quil compiler;
- PyQVM: quantum circuit emulator.

Grove is an open-source Python library containing quantum algorithms that use the quantum programming library pyQuil and the Rigetti Forest toolkit.

Rigetti Computing’s quantum computers can be accessed via its QCS (QCaaS) and can also be accessed via the AWS Marketplace (see § 3.2), Azure Quantum (see § 3.7), Strangeworks QC and the Zapata Computing Orquestra platform (see § 3.11).

Rigetti Computing is also promoting Quantum Programming Studio, a web-based GUI designed to allow developers to construct quantum algorithms and obtain results by simulating directly in the browser or by executing on real quantum computers.

3.10. Xanadu

Xanadu's Strawberry Fields is a full-stack Python library for constructing, simulating, and executing programs on photonic quantum computers according to the Continuous Variable (CV) model of quantum computing, in which the basic information-processing unit is the *qumode* (which uses a continuum of values to describe its quantum state) instead of the *qudit* (which uses a discrete set of values to describe its quantum state).

The Strawberry Fields software stack is separated into two main pieces: user-facing front-end components and lower-level backend component (Figure 3.11).

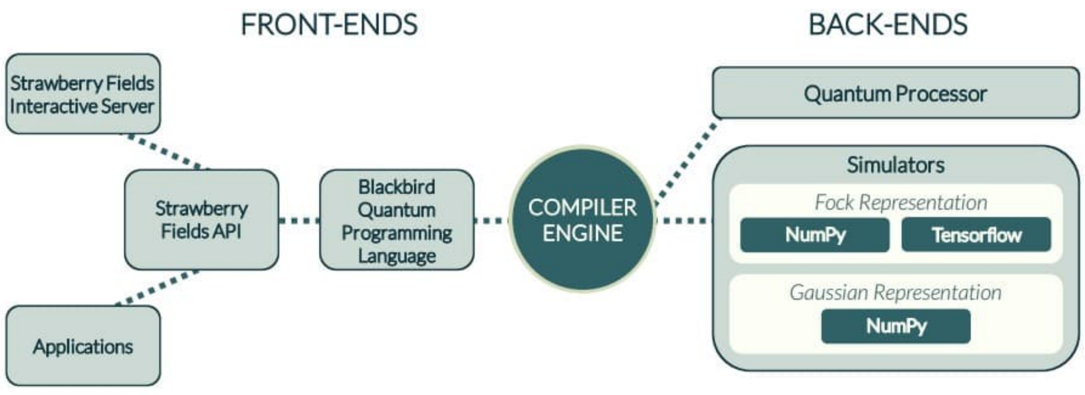


Figure 3.11: Strawberry Fields quantum software stack (source: Xanadu)

The front-end encompasses the Strawberry Fields API for building high-level quantum applications and the Blackbird quantum assembly language for designing quantum circuits. These quantum circuits are then linked to a back-end via a quantum compiler engine.

The back-end targets one of the included quantum computer emulators (Fock/NumPy, Fock/TensorFlow or Gaussian/NumPy) or a CV photonic quantum processor (such as Xanadu X-Series).

Xanadu's Jet and Lightning quantum emulators integrates seamlessly with PennyLane, providing techniques for Quantum Machine Learning (QML) optimisation.

High-level quantum computing applications can be built by leveraging the Strawberry Fields front-end API. Examples include the Strawberry Fields Interactive website, the Quantum Machine Learning Toolbox (for streamlining the training of variational quantum circuits), and SFOpenBoson (an interface for the OpenFermion library).

Strawberry Fields can be accessed through Xanadu Cloud (QCaaS) or can be installed on a local computer.

3.11. Zapata Computing Holding Inc.

Zapata Computing's Orquestra quantum development platform (Figure 3.12) is used to build and deploy quantum generative AI applications.

Through partnerships with leading quantum computer manufacturers, Zapata Computing has developed a quantum software development platform that enables building quantum solutions for solving complex computational problems in optimisation, Quantum Machine Learning (QML) and simulation across a range of industries.

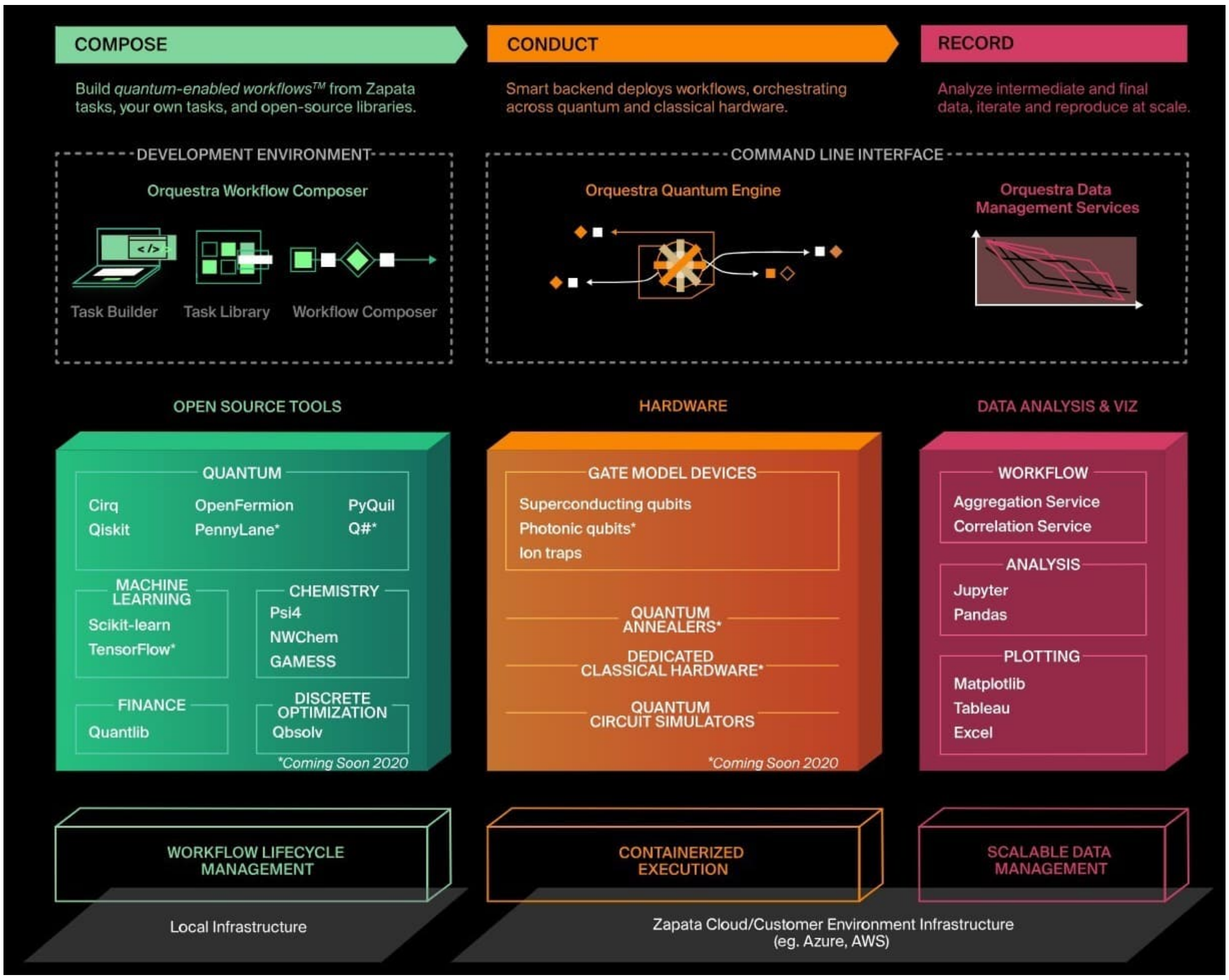


Figure 3.12: Orquestra quantum software development platform (source: Zapata AI)

Appendix A – Overview of quantum programming languages

This appendix provides a brief description (in alphabetical order) of a representative selection of quantum programming languages (low-level quantum assembly languages and high-level quantum programming languages) for quantum annealing, gate-based quantum computing and Measurement-Based Quantum Computing (MBQC).

A.1 Quantum assembly languages

Note
The quantum assembly languages described below are imperative languages.

Blackbird

Blackbird is a quantum assembly language created by Xanadu for developing quantum programs for photonic qubit-based quantum computers. Blackbird is built into Xanadu's Strawberry Fields QSDK but also exists as a separate open-source Python package.

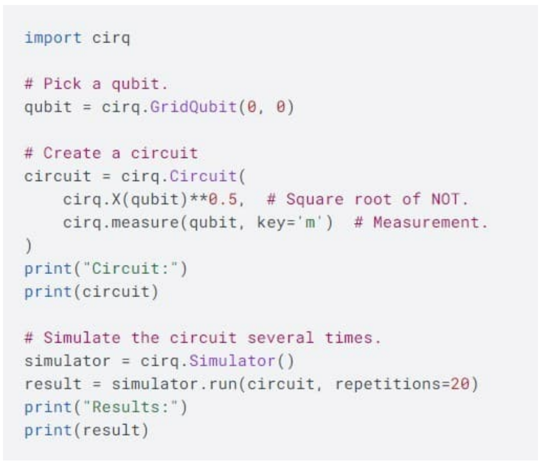
Blackbird is a Domain Specific Language (DSL) for the Continuous Variable (CV) quantum computation model. With a well-defined Extended Backus-Naur Form (EBNF) grammar (Box A.1), and both Python and C++ parsers available, Blackbird provides operations that match the basic CV quantum states, quantum gates and qubit measurements, and maps directly to low-level quantum hardware instructions. Blackbird's abstract syntax keeps a close connection between the code and the quantum operations that it implements, which is modelled after that of ProjectQ (but specialised to the CV setting).

Backus–Naur Form (BNF) is a meta-syntax notation for context-free grammars used to describe the syntax of computer programming languages, document formats, instruction sets and communication protocols. Many extensions and variants of the original BNF notation are in common use, some of which are exactly defined, including Extended Backus–Naur Form (EBNF) and Augmented Backus–Naur Form (ABNF).

Box A.1: Backus-Naur Form (BNF)

Cirq

Cirq is a low-level quantum programming language that is part of Google Quantum AI's Cirq QSDK. It is an open-source Python software library developed by Google Quantum AI for specifying, manipulating and optimising quantum circuits, and then running them on quantum computers and quantum emulators. Cirq provides useful abstractions for dealing with today's NISQ quantum computers, where use of specific hardware features is vital to achieving state-of-the-art results.

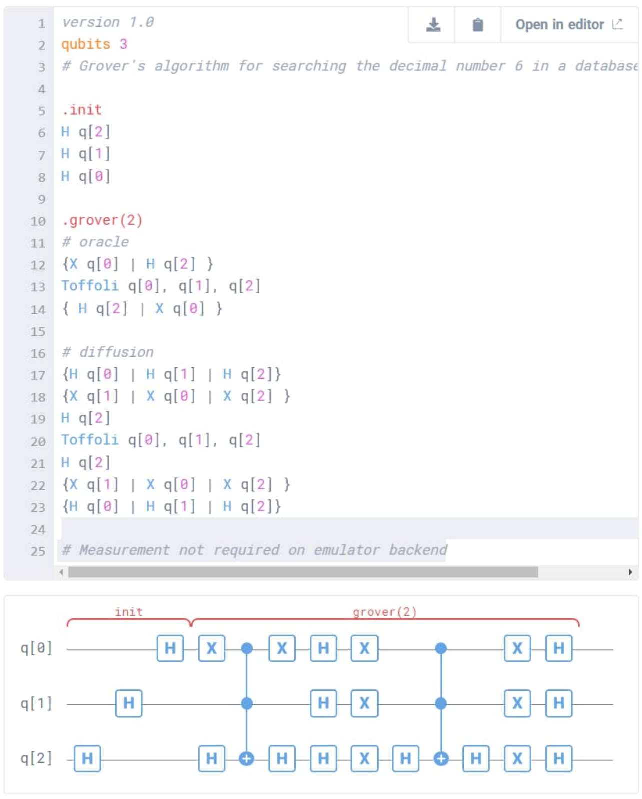


```
import cirq

# Pick a qubit.
qubit = cirq.GridQubit(0, 0)

# Create a circuit
circuit = cirq.Circuit(
    cirq.X(qubit)**0.5,  # Square root of NOT.
    cirq.measure(qubit, key='m')  # Measurement.
)
print("Circuit:")
print(circuit)

# Simulate the circuit several times.
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=20)
print("Results:")
print(result)
```

Figure A.1: Example Cirq source code (source: Google Quantum AI)

cQASM

common Quantum Assembly Language (cQASM) is a hardware-agnostic quantum assembly language which guarantees the interoperability between the quantum compilation and emulation tools. It was developed at TU Delft. A cQASM program declares the classical bits and qubits, describes the operations (quantum gates) on those qubits and the qubit measurements needed to obtain the classical result.



```
version 1.0
qubits 3
# Grover's algorithm for searching the decimal number 6 in a database

.init
H q[2]
H q[1]
H q[0]

.grover(2)
# oracle
{X q[0] | H q[2] }
Toffoli q[0], q[1], q[2]
{ H q[2] | X q[0] }

# diffusion
{H q[0] | H q[1] | H q[2]}
{X q[1] | X q[0] | X q[2] }
H q[2]
Toffoli q[0], q[1], q[2]
H q[2]
{X q[1] | X q[0] | X q[2] }
{H q[0] | H q[1] | H q[2]}

# Measurement not required on emulator backend
```

Figure A.2: Example cQASM source code: subset of Grover's quantum algorithm (source: Quantum Inspire)

eQASM

executable Quantum Assembly Language (eQASM, see Figure A.3) is an intermediate quantum machine language from TU Delft and its subsidiary QuTech. It sits in between high-level programming tools and the quantum processor. It is a compiled language, hence the "e" for “executable”. The compiler manages the dependencies with hardware implementation specifics.

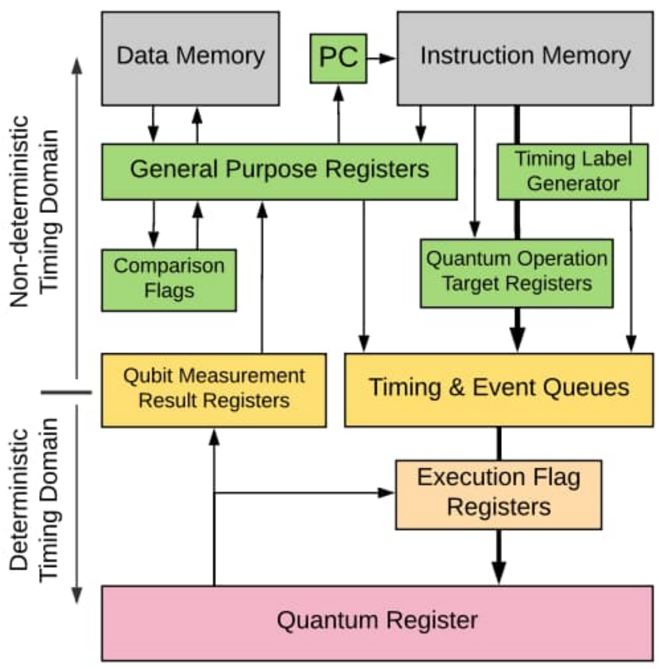
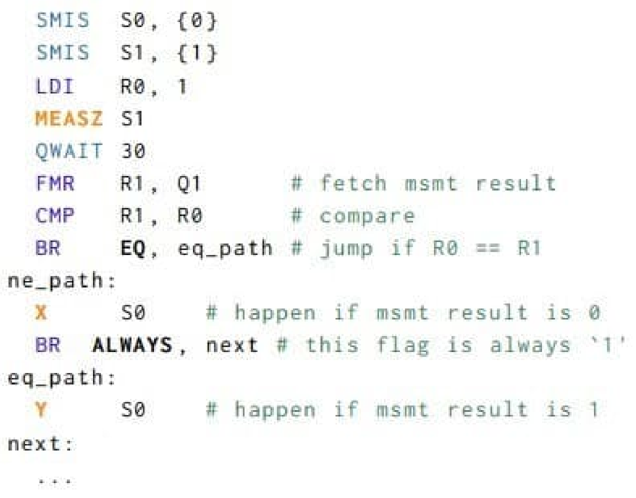

Non-deterministic Timing Domain
Deterministic Timing Domain
Data Memory
PC
Instruction Memory
General Purpose Registers
Timing Label Generator
Comparison Flags
Quantum Operation Target Registers
Qubit Measurement Result Registers
Timing & Event Queues
Execution Flag Registers
Quantum Register

Figure A.3: eQASM architecture (source: QuTech)

An eQASM program can consist of interleaved quantum instructions and auxiliary classical instructions. Since the host CPU can provide classical computation power, auxiliary classical instructions are simple instructions to support the execution of quantum instructions.



```
    SMIS   S0, {0}
    SMIS   S1, {1}
    LDI    R0, 1
    MEASZ  S1
    QWAIT  30
    FMR    R1, Q1        # fetch msmt result
    CMP    R1, R0        # compare
    BR     EQ, eq_path # jump if R0 == R1
ne_path:
    X      S0     # happen if msmt result is 0
    BR   ALWAYS, next # this flag is always '1'
eq_path:
    Y      S0     # happen if msmt result is 1
next:
    ...
```

Figure A.4: Example eQASM source code (source: QuTech)

OpenQASM

Open Quantum Assembly Language (OpenQASM, see Figure A.5) is a quantum programming language designed for describing quantum circuits (representing quantum algorithms) for execution on gate-based quantum computers. It is designed to be an Intermediate Representation (IR) that can be used by higher-level compilers to abstract from the quantum hardware details and allows for the description of a wide range of quantum operations (quantum gates), as well as classical feed-forward flow control based on qubit measurement outcomes.

IBM Q released an open-source reference source code implementation for OpenQASM as part of its Qiskit QSDK to use along with their IBM Quantum Composer. You can either use the IBM Quantum Composer to create quantum programs in OpenQASM or you can use IBM Quantum Composer to convert Qiskit source code to OpenQASM code.

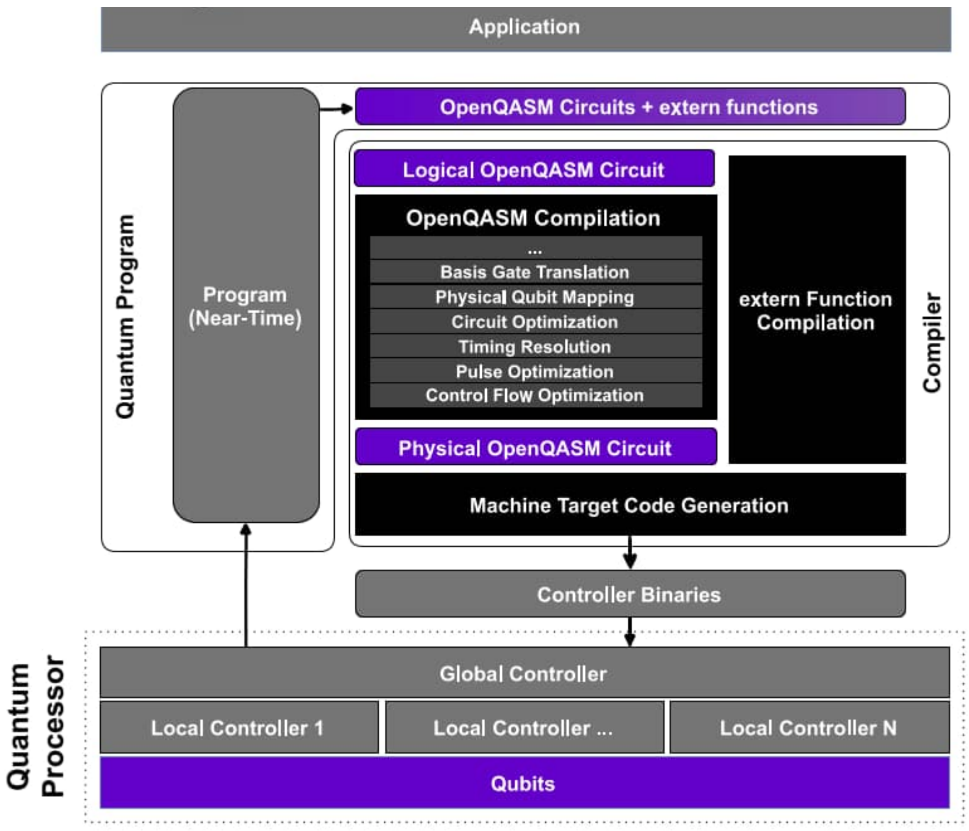
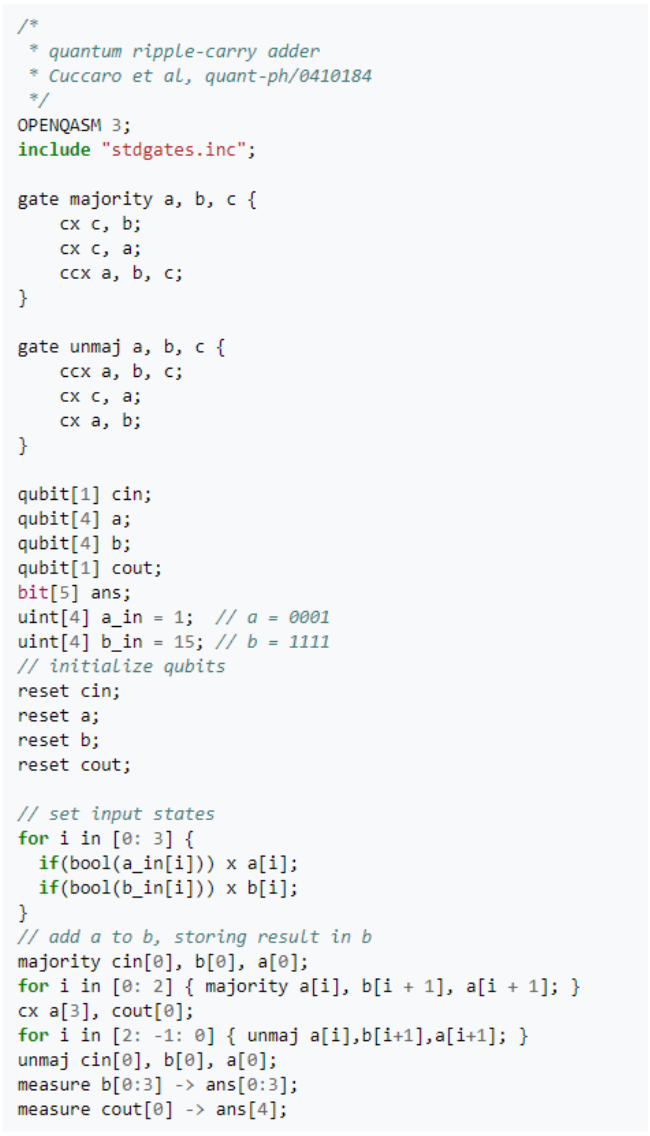


Figure A.5: OpenQASM architecture (source: IBM Q)

OpenQASM includes a mechanism for describing explicit timing of instructions, and allows for the attachment of low-level definitions to quantum gates for tasks such as calibration. Compilers for OpenQASM are expected to support a wide range of classical operations for compile-time constants, but the support for these operations on runtime values may vary between implementations. Furthermore, implementations of the language may not support the full range of data manipulation described in the specification.



```
/*
 * quantum ripple-carry adder
 * Cuccaro et al, quant-ph/0410184
 */
OPENQASM 3;
include "stdgates.inc";

gate majority a, b, c {
    cx c, b;
    cx c, a;
    ccx a, b, c;
}

gate unmaj a, b, c {
    ccx a, b, c;
    cx c, a;
    cx a, b;
}

qubit[1] cin;
qubit[4] a;
qubit[4] b;
qubit[1] cout;
bit[5] ans;
uint[4] a_in = 1;  // a = 0001
uint[4] b_in = 15; // b = 1111
// initialize qubits
reset cin;
reset a;
reset b;
reset cout;

// set input states
for i in [0: 3] {
  if(bool(a_in[i])) x a[i];
  if(bool(b_in[i])) x b[i];
}
// add a to b, storing result in b
majority cin[0], b[0], a[0];
for i in [0: 2] { majority a[i], b[i + 1], a[i + 1]; }
cx a[3], cout[0];
for i in [2: -1: 0] { unmaj a[i],b[i+1],a[i+1]; }
unmaj cin[0], b[0], a[0];
measure b[0:3] -> ans[0:3];
measure cout[0] -> ans[4];
```

Figure A.6: Example OpenQASM source code: adding two 4-bit numbers (source: Wikipedia)

Qibo

Qibo is a quantum programming language that was co-developed by Qilimanjaro. It features an open-source full-stack API for quantum emulation and quantum hardware control (Figure A.7). Its aim is to provide quantum middleware with the following characteristics:

- simplicity: agnostic design to quantum hardware platforms;
- flexibility: transparent mechanism to execute code both on classical and quantum hardware;
- community: a common place where find solutions to accelerate quantum development;
- documentation: describe all steps required to support new quantum processors or quantum emulators;

- applications: maintain a large ecosystem of quantum applications, quantum solution models and quantum algorithms.

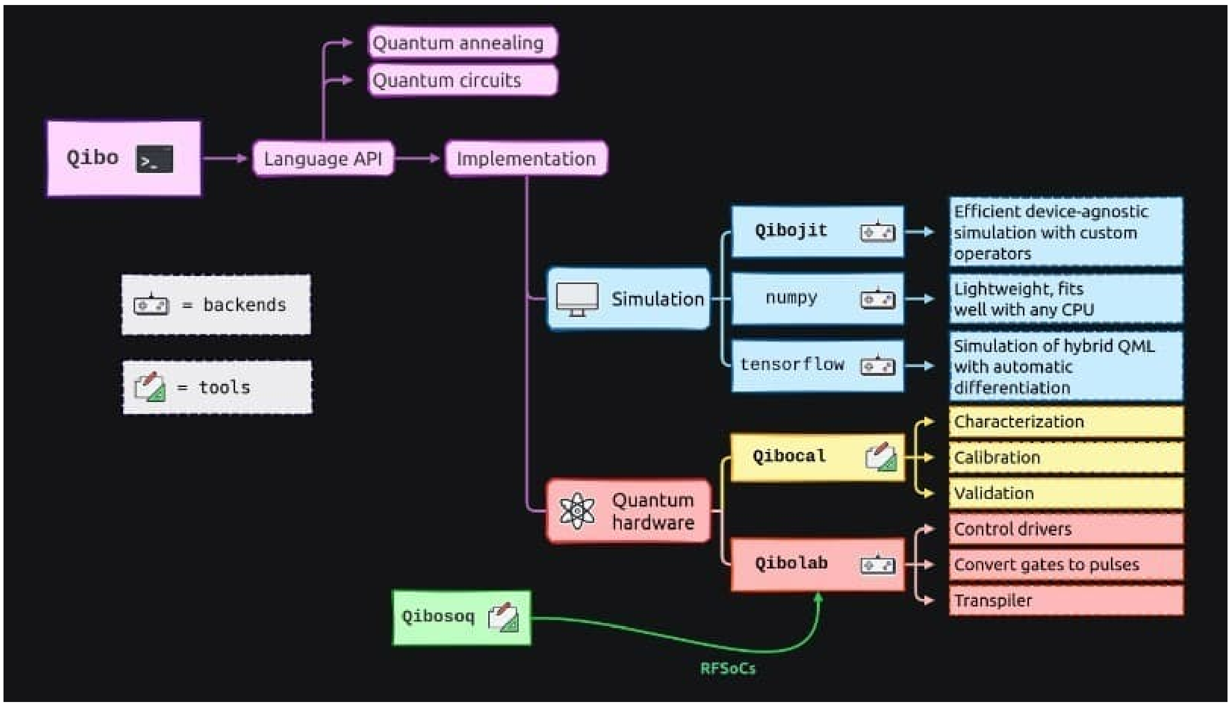


Figure A.7: Qibo components (source: Qibo)

QMASM

Quantum Macro Assembler (QMASM) is a low-level language specific to D-Wave Systems' quantum annealers[8].

QMASM fills a gap in the software ecosystem of D-Wave System's quantum annealers by shielding the programmer from having to know system-specific hardware details while still enabling programs to be expressed at a fairly low level of abstraction. It is therefore analogous to a conventional macro assembler and can be used in much the same way: as a target either for programmers who want a great deal of control over the hardware or for compilers that implement higher-level languages.

Some relevant QMASM language features are that it:

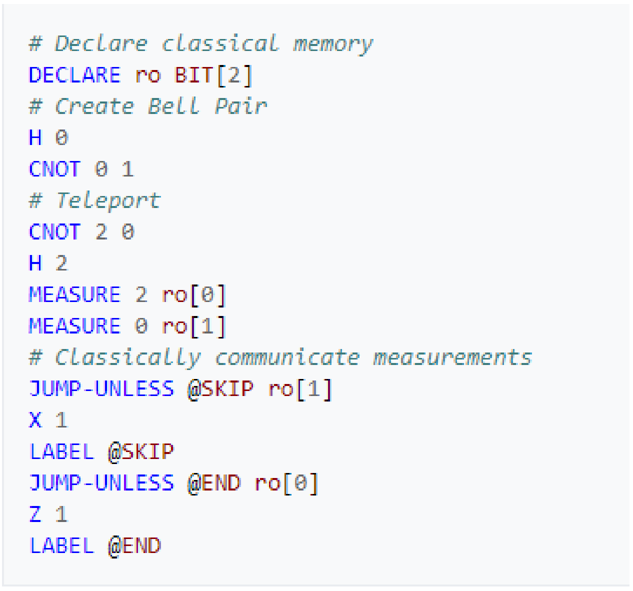
- allows programs to refer to variables symbolically;

[8] This tool used to be called "QASM" but was renamed to avoid confusion with MIT's QASM, which is used to describe quantum circuits (a different model of quantum computation from what D-Wave Systems uses), and the IBM Q QASM language (now OpenQASM) language, which is also used for describing quantum circuits.

- accepts arbitrary values for the function coefficients and automatically maps those onto what is accepted by the underlying hardware;
- provides shortcut syntax for biasing two variables to have the same value (or, respectively, the opposite value);
- supports macros to facilitate code reuse;
- allows sets of macros to appear in a separate file that can be included into a main routine.

Quil

Quil was developed by Rigetti Computing for programming its quantum processors. It is part of Rigetti's Forest QSDK. An open-source Python library called PyQuil was introduced to develop Quil programs with higher level constructs.



```
# Declare classical memory
DECLARE ro BIT[2]
# Create Bell Pair
H 0
CNOT 0 1
# Teleport
CNOT 2 0
H 2
MEASURE 2 ro[0]
MEASURE 0 ro[1]
# Classically communicate measurements
JUMP-UNLESS @SKIP ro[1]
X 1
LABEL @SKIP
JUMP-UNLESS @END ro[0]
Z 1
LABEL @END
```

Figure A.8: Example Quil source code: teleportation of a qubit (source: Wikipedia)

ZX-calculus

ZX-calculus is a graphical quantum programming language that uses topological composition rules. It was created by Bob Coecke and Ross Duncan. ZX-calculus visualises the modifications made to a set of qubits by means of ZX-diagrams (unidirectional multi-graphs).

ZX-calculus is particularly useful for the specification of Measurement-Based Quantum Computing (MBQC), for developing Quantum Error Correction (QEC) code and for optimisation of quantum source code by quantum compilers.

PyZX is an open-source Python tool that implements ZX-Calculus principles for the creation, visualisation and automated rewriting of large-scale quantum circuits.

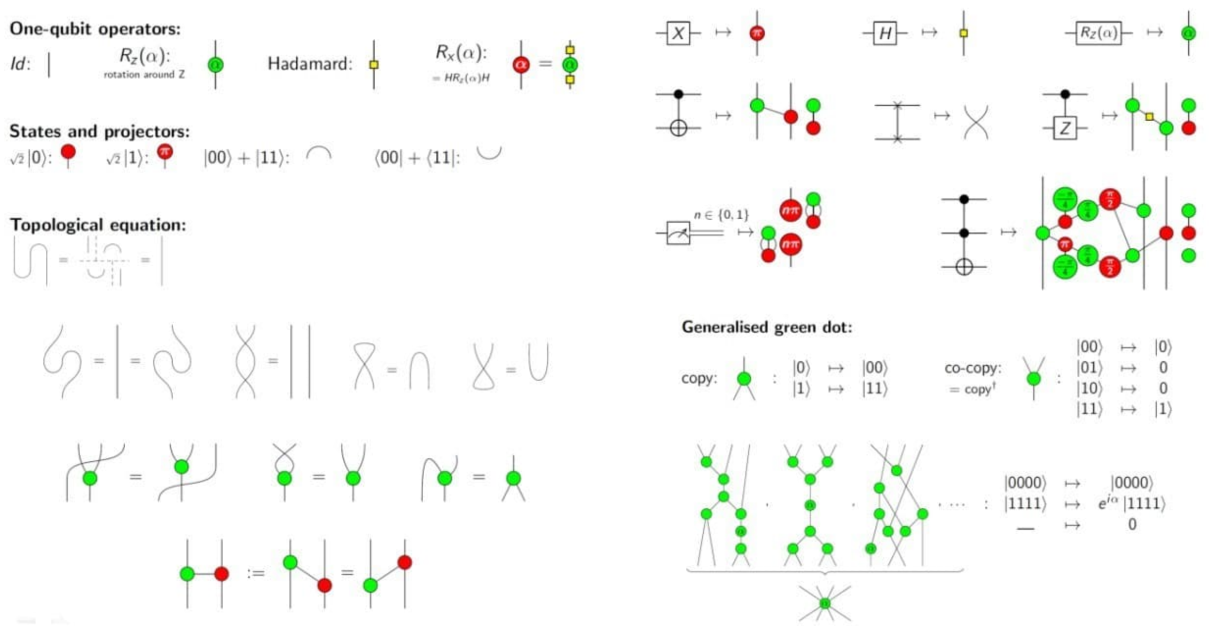

One-qubit operators:
Id:
$R_z(\alpha)$: rotation around Z
Hadamard:
$R_x(\alpha)$: $= HR_z(\alpha)H$
States and projectors:
$\sqrt{2}\,|0\rangle$:
$\sqrt{2}\,|1\rangle$:
$|00\rangle + |11\rangle$:
$\langle 00| + \langle 11|$:
Topological equation:
$n \in \{0,1\}$
Generalised green dot:
copy:
$|0\rangle \mapsto |00\rangle$
$|1\rangle \mapsto |11\rangle$
co-copy: = copy†
$|00\rangle \mapsto |0\rangle$
$|01\rangle \mapsto 0$
$|10\rangle \mapsto 0$
$|11\rangle \mapsto |1\rangle$
$|0000\rangle \mapsto |0000\rangle$
$|1111\rangle \mapsto e^{i\alpha}|1111\rangle$
$\mapsto 0$

Figure A.9: Examples of ZX calculus operations (source: Olivier Ezratty)

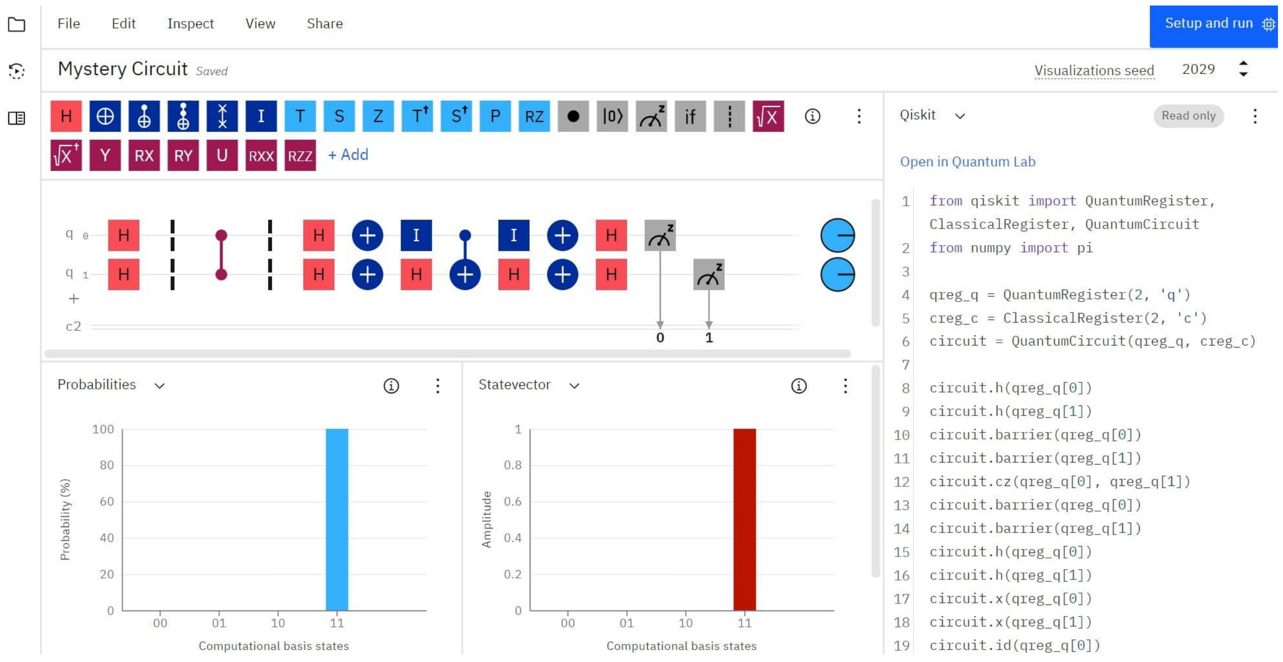
A.2 High-level quantum programming languages

High-level quantum programming languages are either imperative languages (described in A.2.1), declarative languages (described in A.2.2) or functional languages (described in A.2.3).

A.2.1 Imperative quantum programming languages

IBM Quantum Composer

The IBM Quantum Composer is a GUI that allows users to construct various quantum algorithms or run other quantum experiments. Users may see the results of their quantum algorithms by either running it on a real quantum processor or by using a quantum emulator. The IBM Quantum Composer can also be used in scripting mode, where the user can write programs in the OpenQASM-language instead.



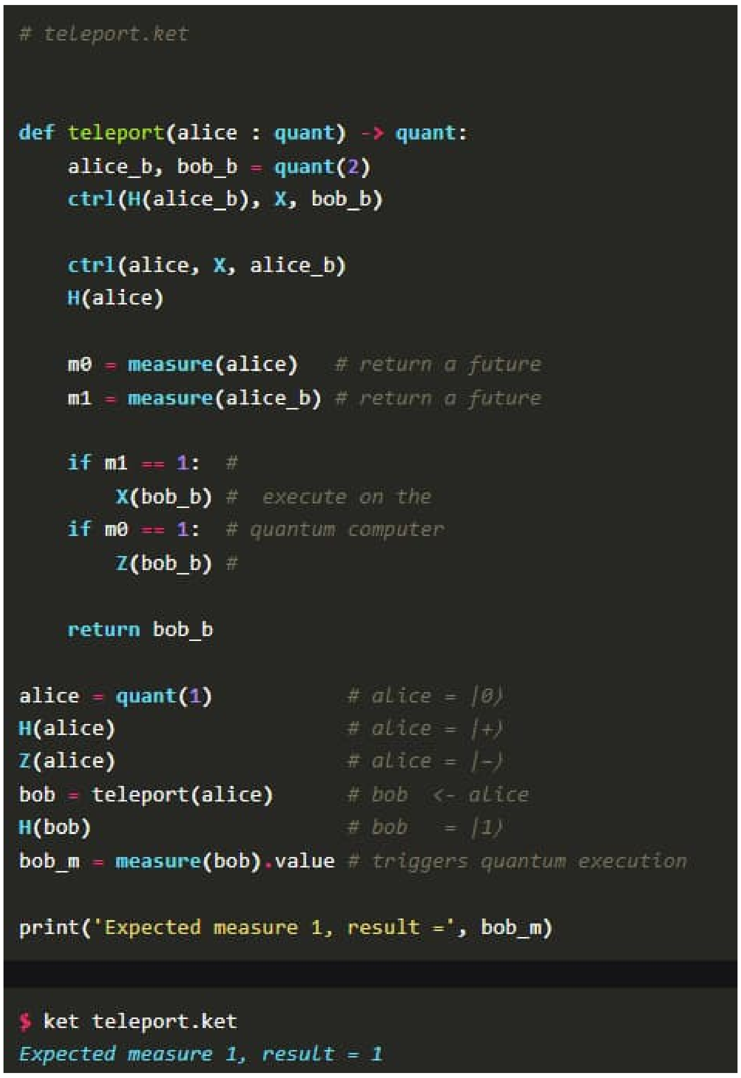
```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.h(qreg_q[0])
circuit.h(qreg_q[1])
circuit.barrier(qreg_q[0])
circuit.barrier(qreg_q[1])
circuit.cz(qreg_q[0], qreg_q[1])
circuit.barrier(qreg_q[0])
circuit.barrier(qreg_q[1])
circuit.h(qreg_q[0])
circuit.h(qreg_q[1])
circuit.x(qreg_q[0])
circuit.x(qreg_q[1])
circuit.id(qreg_q[0])
```

Figure A10: Example IBM Quantum Composer diagram (source: IBM Q)

The IBM Quantum Composer and the IBM Quantum Lab, which are part of IBM Q's Qiskit QSDK, form an online platform allowing public and premium access to cloud-based quantum computing services provided by IBM Q. This includes access to a set of IBM Q's quantum processors, a set of tutorials on quantum computation and access to an interactive textbook. There are currently more than 20 quantum processors on the service, some of which are freely available for the public. This service can be used to run quantum algorithms and experiments, and explore tutorials and quantum emulations around what might be possible with quantum computing.

Ket

Ket Quantum Programming is an open-source platform maintained by Quantuloop, providing dynamic interaction between classical and quantum data at the programming level for classical-quantum development.



```
# teleport.ket


def teleport(alice : quant) -> quant:
    alice_b, bob_b = quant(2)
    ctrl(H(alice_b), X, bob_b)

    ctrl(alice, X, alice_b)
    H(alice)

    m0 = measure(alice)   # return a future
    m1 = measure(alice_b) # return a future

    if m1 == 1:  #
        X(bob_b) #  execute on the
    if m0 == 1:  # quantum computer
        Z(bob_b) #

    return bob_b

alice = quant(1)            # alice = |0⟩
H(alice)                    # alice = |+⟩
Z(alice)                    # alice = |-⟩
bob = teleport(alice)       # bob  <- alice
H(bob)                      # bob   = |1⟩
bob_m = measure(bob).value # triggers quantum execution

print('Expected measure 1, result =', bob_m)
```

```
$ ket teleport.ket
Expected measure 1, result = 1
```

Figure A.11: Example Ket source code: quantum teleportation (source: Ket)

The Ket Quantum Programming platform consists of three main projects:

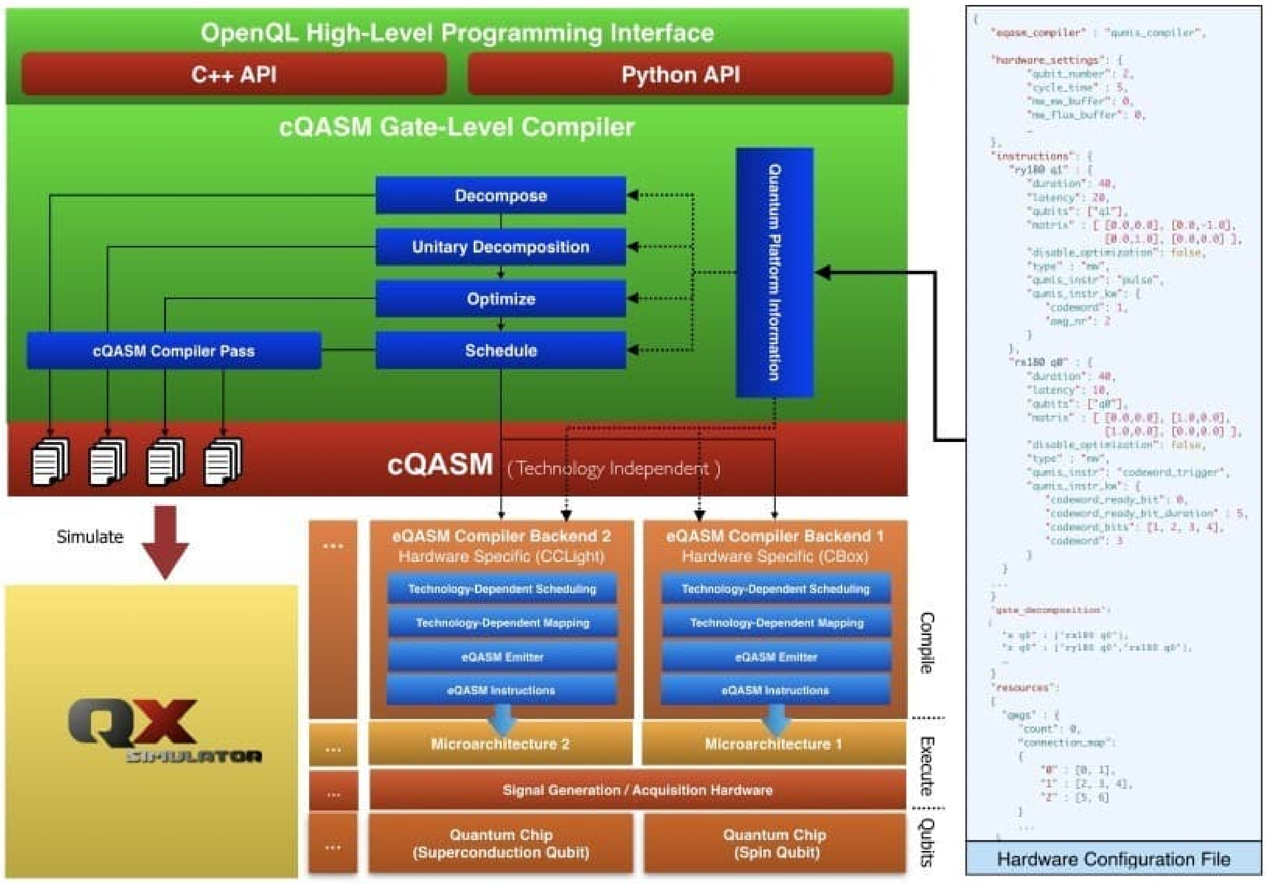
1. Ket is an embedded language designed to facilitate quantum programming, leveraging the familiar syntax and simplicity of Python, letting anyone quickly prototype and test a quantum application.

2. Libket is the runtime library for the Ket language, but one can use it for quantum acceleration on embedded systems using C, C++ or Rust.

3. Ket Bitwise Simulator (KBW) is a noise-free quantum emulator that allows anyone to test quantum applications on classical computers. KBW features two emulation methods: dense simulation using quantum state vector-based emulation and sparse emulation based on the Bitwise representation.

OpenQL

Open Quantum Library (OpenQL, see Figure A.12) is an open-source quantum programming framework created by TU Delft. It includes a high-level quantum programming language, its associated quantum compiler and a low-level assembly language, cQASM.

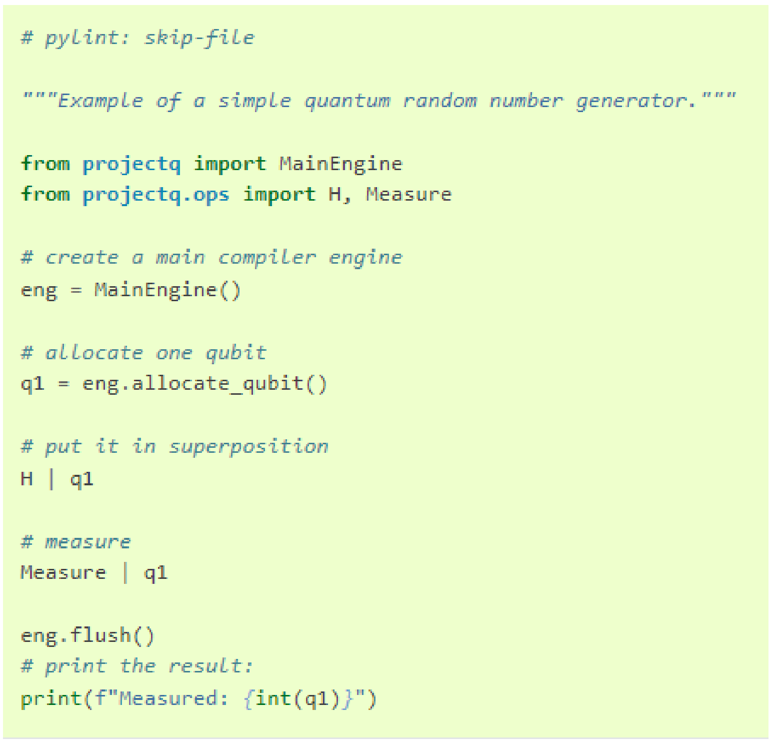

OpenQL High-Level Programming Interface
C++ API
Python API
cQASM Gate-Level Compiler
Decompose
Unitary Decomposition
Optimize
Schedule
cQASM Compiler Pass
Quantum Platform Information
cQASM (Technology Independent)
Simulate
QX SIMULATOR
eQASM Compiler Backend 2 Hardware Specific (CCLight)
eQASM Compiler Backend 1 Hardware Specific (CBox)
Technology-Dependent Scheduling
Technology-Dependent Mapping
eQASM Emitter
eQASM Instructions
Microarchitecture 2
Microarchitecture 1
Signal Generation / Acquisition Hardware
Quantum Chip (Superconduction Qubit)
Quantum Chip (Spin Qubit)
...
Compile
Execute
Qubits
Hardware Configuration File

```
{
  "eqasm_compiler" : "qumis_compiler",

  "hardware_settings": {
      "qubit_number": 2,
      "cycle_time" : 5,
      "mw_mw_buffer": 0,
      "mw_flux_buffer": 0,
      ...
  },
  "instructions": {
     "ry180 q1" : {
        "duration": 40,
        "latency": 20,
        "qubits": ["q1"],
        "matrix" : [ [0.0,0.0], [0.0,-1.0],
                     [0.0,1.0], [0.0,0.0] ],
        "disable_optimization": false,
        "type" : "mw",
        "qumis_instr": "pulse",
        "qumis_instr_kw": {
           "codeword": 1,
           "awg_nr": 2
        }
     },
     "rx180 q0" : {
        "duration": 40,
        "latency": 10,
        "qubits": ["q0"],
        "matrix" : [ [0.0,0.0], [1.0,0.0],
                     [1.0,0.0], [0.0,0.0] ],
        "disable_optimization": false,
        "type" : "mw",
        "qumis_instr": "codeword_trigger",
        "qumis_instr_kw": {
           "codeword_ready_bit": 0,
           "codeword_ready_bit_duration" : 5,
           "codeword_bits": [1, 2, 3, 4],
           "codeword": 3
        }
     }
  ...
  }
  "gate_decomposition":
  {
    "x q0" : ["rx180 q0"],
    "z q0" : ["ry180 q0","rx180 q0"],
    ...
  }
  "resources":
  {
    "qwgs" : {
      "count": 0,
      "connection_map":
      {
        "0" : [0, 1],
        "1" : [2, 3, 4],
        "2" : [5, 6]
      }
      ...
```

Figure A.12: OpenQL architecture (source: TU Delft)

ProjectQ

ProjectQ is a scripting quantum programming language from ETH Zurich that takes the form of an open-source Python framework. It includes a compiler that converts quantum code into C++ language for execution in a quantum emulator with a traditional processor. It supports IBM Q's quantum computers via their OpenQASM language, as well as emulation on a classical computer via a C++ implementation that supports up to 28 qubits. ProjectQ is compatible with the OpenFermion initiative.



```python
# pylint: skip-file

"""Example of a simple quantum random number generator."""

from projectq import MainEngine
from projectq.ops import H, Measure

# create a main compiler engine
eng = MainEngine()

# allocate one qubit
q1 = eng.allocate_qubit()

# put it in superposition
H | q1

# measure
Measure | q1

eng.flush()
# print the result:
print(f"Measured: {int(q1)}")
```

Figure A.13: Example Project Q source code: QRNG (source: ProjectQ)

Q#

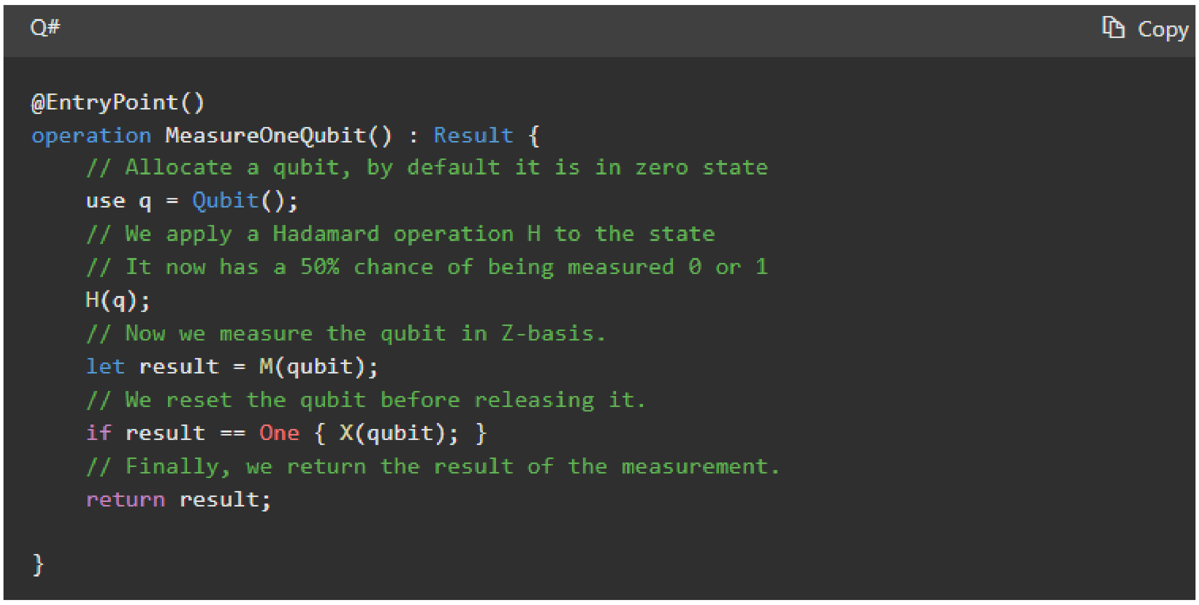
Q# is a high-level programming language developed by Microsoft for developing quantum algorithms. It is part of the Quantum Development Kit (QDK) and is designed to be quantum hardware agnostic, scale to the full range of quantum applications and to optimise execution.

As a programming language, Q# draws familiar elements from Python, C#, and F#, and supports a basic procedural model for writing programs with loops, if/then statements and common data types. It introduces new quantum-specific data structures and operations, such as repeat-until-success and adaptive phase estimation, which allow the integration of quantum and classical computations. For example, the flow control of the classical component can be based on the outcome of a quantum measurement.

In Q#, qubits are a resource that are requested from the runtime when needed and returned when no longer in use. This is similar to the way that classical languages deal with heap memory.

The Q# runtime is responsible for determining a mapping from a qubit variable in a quantum program to an actual logical or physical qubit that allows the quantum algorithm to execute, including any qubit state transfer and remapping required during execution. That mapping may be deferred until after the topology and other details of the target quantum processor are known.

The representation used in Q# has the interesting implication that all of the actual quantum computing is done by side effect. There is no way to directly interact with the quantum state of the quantum processor as it has no software representation at all. Instead, operations performed on qubit entities have the side effect of modifying its quantum state. Effectively, the quantum state of the quantum processor is an opaque global variable that is inaccessible, except through a small set of accessor primitives (and even these accessors have side effects on the quantum state, and so are really "mutators with results" rather than true accessors).



```
@EntryPoint()
operation MeasureOneQubit() : Result {
    // Allocate a qubit, by default it is in zero state
    use q = Qubit();
    // We apply a Hadamard operation H to the state
    // It now has a 50% chance of being measured 0 or 1
    H(q);
    // Now we measure the qubit in Z-basis.
    let result = M(qubit);
    // We reset the qubit before releasing it.
    if result == One { X(qubit); }
    // Finally, we return the result of the measurement.
    return result;

}
```

Figure A.14: Example Q# source code (source: Microsoft)

Q.js

Q.js is an open-source free graphical quantum emulator (a drag-and-drop quantum circuit editor, written in JavaScript and thus running in a browser. It includes a powerful JavaScript library and there is nothing to install and nothing to configure.

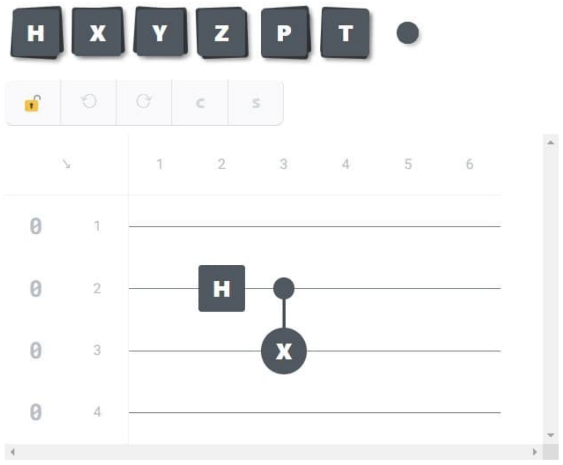


Figure A.15: Example Q.js source code: Bell state (source: Q.js)

QCL

Quantum Computation Language (QCL) is one of the first quantum programming languages, created at the Austrian Institute of Technology in Vienna.

The most important feature of QCL is its support for user-defined operators and functions. The syntax resembles that of the C programming language and its classical data types are similar to primitive data types in C. One can combine classical code and quantum code in the same program.

QCL introduces the quantum register (an array of qubits), called qureg, which is considered to be the foundation quantum data type in QCL.

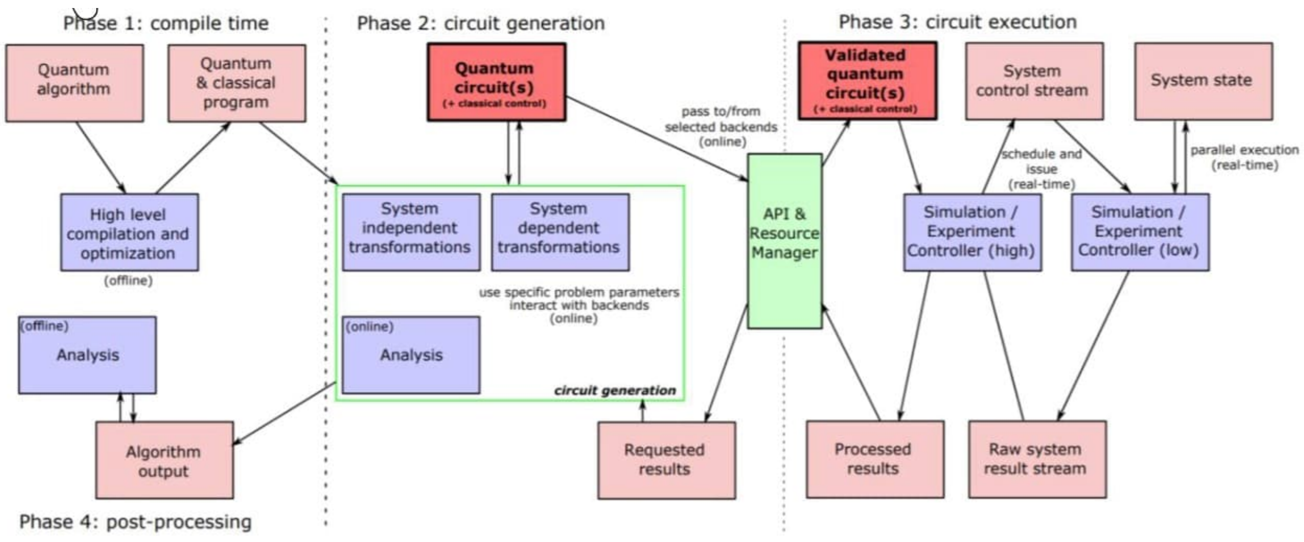
Similar to modern programming languages, QCL enables one to define new operations that can manipulate quantum data.

Qiskit

The Qiskit quantum programming language (Figure A.16), which is part of IBM Q's Qiskit QSDK, is a high-level scripting library associated with OpenQASM. It can be used with Python, JavaScript and Swift (a general-purpose language from Apple) and runs on Windows, Linux and MacOS platforms. It is published in open-source and is also supported by other quantum computers vendors such as trapped ion qubit-based AQT and IonQ, and cold atom qubit-based ColdQuanta.

Qiskit includes a graphical circuit-drawing function that generates a graphical visualisation of quantum circuits using the open-source document composition language LaTeX.

Qiskit comes with numerous templates and sample codes to exploit a wide range of known quantum algorithms.



Phase 1: compile time
Phase 2: circuit generation
Phase 3: circuit execution
Quantum algorithm
Quantum & classical program
Quantum circuit(s) (+ classical control)
pass to/from selected backends (online)
Validated quantum circuit(s) (+ classical control)
System control stream
System state
schedule and issue (real-time)
parallel execution (real-time)
High level compilation and optimization
(offline)
System independent transformations
System dependent transformations
use specific problem parameters interact with backends (online)
API & Resource Manager
Simulation / Experiment Controller (high)
Simulation / Experiment Controller (low)
(offline) Analysis
(online) Analysis
circuit generation
Algorithm output
Requested results
Processed results
Raw system result stream
Phase 4: post-processing

Figure A.16: Qiskit workflow (source: IBM Q)

Quantum source code compilation takes place either on IBM Q's classicàl cloud-based HPC emulator or on an IBM Q quantum computer that is available in the cloud, by means of free access on 7-qubit and 127-qubit systems or paid access on 27-qubit to 433-qubit systems.

Qunity

Qunity is a quantum programming language created by the universities of Maryland and Chicago and by AWS. Its goal is to unify quantum and classical programming concepts in a single language. Its syntax uses familiar programming constructs that can have both quantum and classical effects, such as summing linear operators, using exception handling syntax with projective measurements and using aliasing to induce entanglement. It can also automatically construct reversible subroutines from irreversible quantum algorithms through the uncomputation of “garbage” outputs. It can for example create full quantum oracle functions for quantum algorithms like Grover, Deutsch-Jozsa and Simon.

Qunity is still being developed; Qunity source code will be compiled to generate OpenQASM quantum assembly code.

$$
\begin{aligned}
\mathsf{and} &\coloneqq \lambda x \xmapsto{\mathtt{Bit}\otimes\mathtt{Bit}} \\
&\qquad \mathtt{ctrl}\ x \quad {}_{\mathtt{Bit}\otimes\mathtt{Bit}}\begin{Bmatrix} (0,0) \mapsto (x,0) \\ (0,1) \mapsto (x,0) \\ (1,0) \mapsto (x,0) \\ (1,1) \mapsto (x,1) \end{Bmatrix}_{(\mathtt{Bit}\otimes\mathtt{Bit})\otimes\mathtt{Bit}} \triangleright \mathsf{snd}_{(\mathtt{Bit}\otimes\mathtt{Bit})\otimes\mathtt{Bit}} \\
\mathsf{couple}(k) &\coloneqq \lambda(x_0,x_1) \xmapsto{\mathtt{Bit}\otimes\mathtt{Bit}} \\
&\qquad \mathtt{ctrl}\ \mathsf{and}(x_0,x_1) \quad {}_{\mathtt{Bit}}\begin{Bmatrix} 0 \mapsto (x_1,x_0) \\ 1 \mapsto (x_1,x_0) \triangleright \mathsf{gphase}_{\mathtt{Bit}\otimes\mathtt{Bit}}(2\pi\ /\ 2^{\mathsf{k}}) \end{Bmatrix}_{\mathtt{Bit}\otimes\mathtt{Bit}} \\
\mathtt{rotations}(0) &\coloneqq \lambda() \xmapsto{\mathtt{Bit}^{\otimes 0}} () \\
\mathtt{rotations}(1) &\coloneqq \lambda(x,()) \xmapsto{\mathtt{Bit}^{\otimes 1}} (\mathsf{had}\,x,()) \\
\mathtt{rotations}(n+2) &\coloneqq \lambda(x_0,x) \xmapsto{\mathtt{Bit}^{\otimes(n+2)}} \\
&\qquad \begin{pmatrix} \mathtt{let}\ (x_0,(y_0',y)) =_{\mathtt{Bit}^{\otimes(n+2)}} (x_0, x \triangleright \mathtt{rotations}(n+1))\ \mathtt{in} \\ \mathtt{let}\ ((y_0,y_1),y) =_{(\mathtt{Bit}\otimes\mathtt{Bit})\otimes\mathtt{Bit}^{\otimes n}} ((x_0,y_0') \triangleright \mathtt{couple}(n+2), y)\ \mathtt{in} \\ (y_0,(y_1,y)) \end{pmatrix} \\
\mathtt{qft}(0) &\coloneqq \lambda() \xmapsto{\mathtt{Bit}^{\otimes 0}} () \\
\mathtt{qft}(n+1) &\coloneqq \lambda x \xmapsto{\mathtt{Bit}^{\otimes(n+1)}} \mathtt{let}\ (x_0,x') =_{\mathtt{Bit}^{\otimes n}} x \triangleright \mathtt{rotations}(n+1)\ \mathtt{in}\ (x_0, x' \triangleright \mathtt{qft}(n))
\end{aligned}
$$

Figure A.17: Example Qunity source code: QFT (source: University of Maryland)

Scaffold

Scaffold (Figure A.18) is a C-like language that compiles to OpenQASM quantum assembler code. It is open source built on top of the LLVM Compiler Infrastructure (a collection of modular and reusable compiler and toolchain technologies[9]) to perform optimisations on Scaffold source code before generating a specified quantum processor instruction set. It is used to program traditional code which is then automatically transformed into quantum gates via its Classical Code to Quantum Gates (C2QG) function.

Scaffold was developed at Princeton University and its development was funded by IARPA.

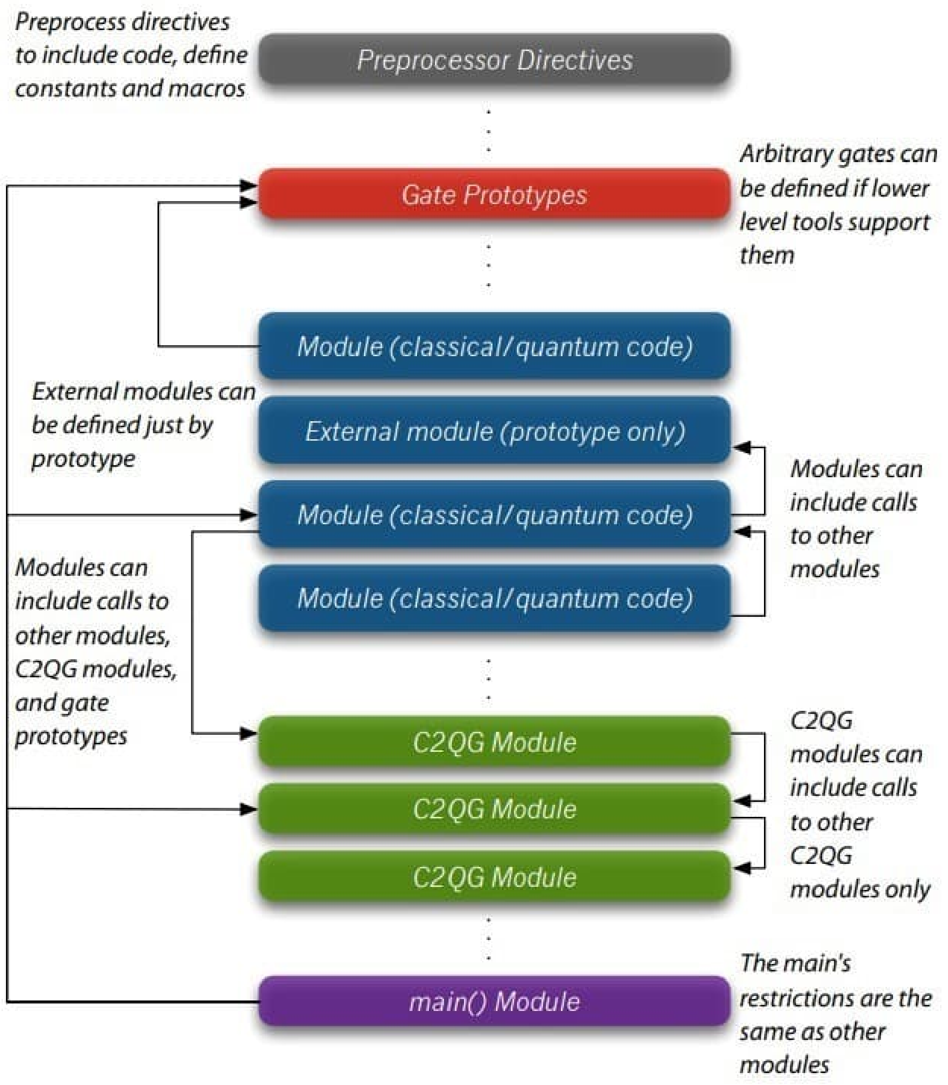
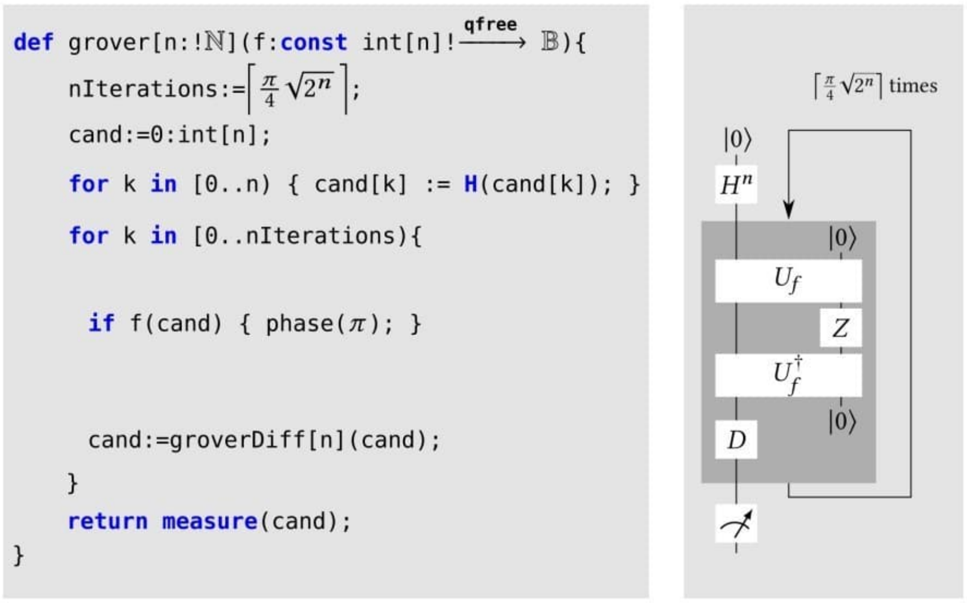

Preprocess directives to include code, define constants and macros
Preprocessor Directives
Gate Prototypes
Arbitrary gates can be defined if lower level tools support them
Module (classical/quantum code)
External modules can be defined just by prototype
External module (prototype only)
Modules can include calls to other modules
Module (classical/quantum code)
Module (classical/quantum code)
Modules can include calls to other modules, C2QG modules, and gate prototypes
C2QG Module
C2QG Module
C2QG Module
C2QG modules can include calls to other C2QG modules only
main() Module
The main's restrictions are the same as other modules

Figure A.18: Structure of a Scaffold program (source: Princeton University)

[9] Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" is not an acronym; it is just the name of the project.

Silq

Silq is a high-level quantum programming language with a strong static type system. It was developed at ETH Zürich.



```
def grover[n:!ℕ](f:const int[n]! -qfree-> 𝔹){
    nIterations:=⌈π/4 √2^n⌉;
    cand:=0:int[n];
    for k in [0..n) { cand[k] := H(cand[k]); }
    for k in [0..nIterations){

      if f(cand) { phase(π); }


      cand:=groverDiff[n](cand);
    }
    return measure(cand);
}
```

Figure A.19: Example Silq source code: Grover's quantum algorithm (source: ETH Zurich)

A.2.2 Declarative quantum programming languages

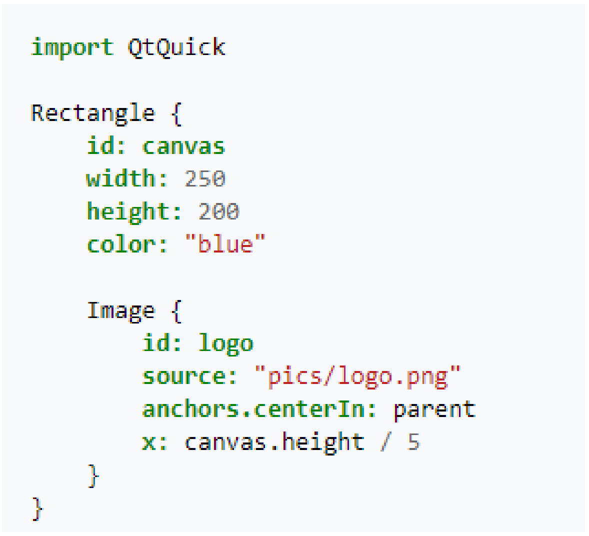
QML

Qt Modeling Language (QML) is a user interface markup language. It is a declarative language for designing user interface–centric applications. Inline JavaScript code handles imperative aspects. QML is associated with Qt Quick, the UI creation kit originally developed by Nokia within the Qt framework[10]. QML also introduces both classical and quantum control operators.

A QML document describes a hierarchical object tree. QML modules shipped with Qt include primitive graphical building blocks, modelling components, behavioural components and more complex controls. These elements can be combined to build components ranging in complexity from simple buttons and sliders, to complete internet-enabled programs.

QML elements can be augmented by standard JavaScript both inline and via included .js files. Elements can also be seamlessly integrated and extended by C++ components using the Qt framework.

Because QML and JavaScript are very similar, almost all code editors supporting JavaScript will work. QML and JavaScript code can be compiled into native C++ binaries with the Qt Quick Compiler. Alternatively there is a QML cache file format which stores a compiled version of QML dynamically for faster start-up the next time it is run.



```
import QtQuick

Rectangle {
    id: canvas
    width: 250
    height: 200
    color: "blue"

    Image {
        id: logo
        source: "pics/logo.png"
        anchors.centerIn: parent
        x: canvas.height / 5
    }
}
```

Figure A.20: Example QML source code (source: Wikipedia)

[10] Qt Quick is used for mobile applications where touch input, fluid animations and user experience are crucial.

A.2.3 Functional quantum programming languages

LIQUi|⟩

Language-Integrated Quantum Operations (LIQUi|⟩) is a quantum emulation extension on the F# functional programming language. It is currently being developed by the Quantum Architectures and Computation (QuArC) Group, part of the StationQ efforts at Microsoft Research. LIQUi|⟩ seeks to allow quantum developers to experiment with quantum algorithm design before physical quantum computers are available for use.

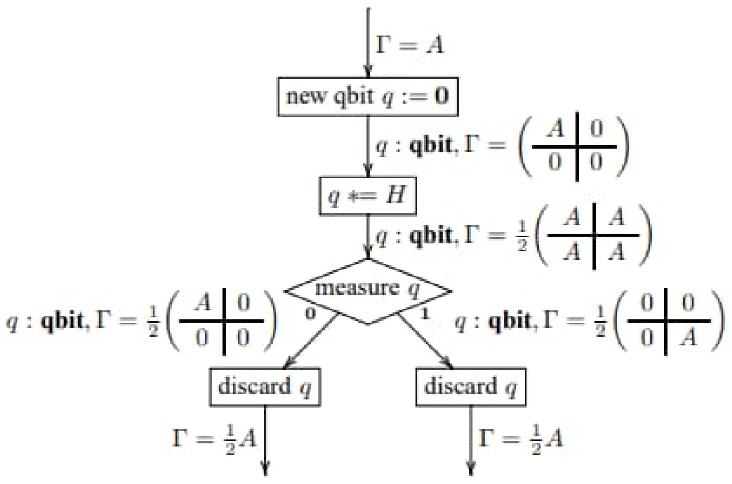
LIQUi|⟩ can be used to translate a quantum algorithm written in the form of a high-level program into low-level machine instructions for a quantum processor. The toolkit includes a compiler, optimisers, translators, various emulators and a host of examples.

LIQUi|⟩ allows the emulation of Hamiltonians, quantum circuits, quantum stabilizer circuits and quantum noise models, and supports client, service and cloud modes of operation. It allows the user to express circuits in F#, and supports the extraction of quantum circuit data structures that can be passed to other components for circuit optimisation, Quantum Error Correction (QEC), quantum gate replacement, export or rendering. The LIQUi|⟩ software is architected to be fully modular to permit easy extension as desired.

LIQUi|⟩ includes state-of-the-art circuit emulation of up to 30 qubits, limited only by memory and computing threads.

QFC and QPL

Quantum Flow Charts (QFC) and Quantum Programming Language (QPL) are two closely related quantum programming languages defined by Peter Selinger. They differ only in their syntax: QFC uses a flowchart syntax, whereas QPL uses a textual syntax. Both languages can operate on both quantum and classical data.



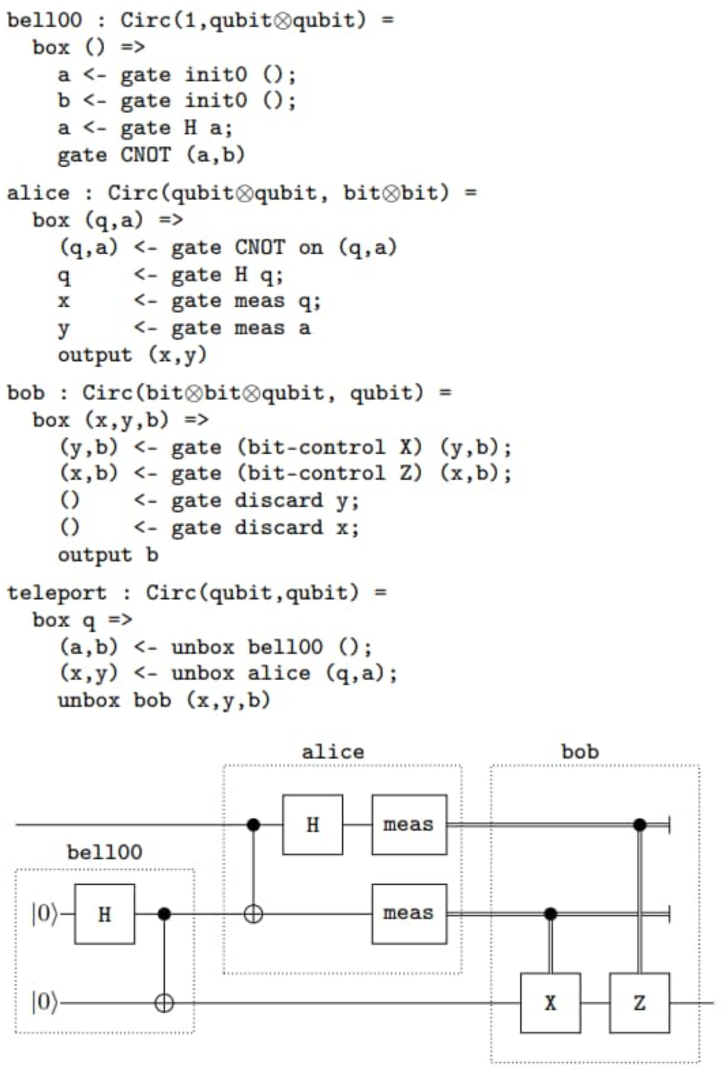
$\Gamma = A$
new qbit $q := 0$
$q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right)$
$q \mathrel{*}= H$
$q : \textbf{qbit}, \Gamma = \frac{1}{2}\left(\begin{array}{c|c} A & A \\ \hline A & A \end{array}\right)$
measure q
0
1
$q : \textbf{qbit}, \Gamma = \frac{1}{2}\left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right)$
$q : \textbf{qbit}, \Gamma = \frac{1}{2}\left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & A \end{array}\right)$
discard q
discard q
$\Gamma = \frac{1}{2}A$
$\Gamma = \frac{1}{2}A$

Figure A.21: Example QFC flowchart: probabilistic fair coin toss (source: Peter Selinger)

QWIRE

QWIRE is a quantum programming language from the University of Pennsylvania. It is a small quantum circuit language embedded in a classical host language, which provides three core features:

1. a platform for high level quantum computing, with the expressiveness of embedded languages like LIQUi|⟩;
2. a linear type system that guarantees that generated circuits are well-formed and respect the laws of quantum mechanics;
3. a concrete denotational semantics, specified in terms of density matrices, for proving properties and equivalences of quantum circuits.



```
bell00 : Circ(1,qubit⊗qubit) =
  box () =>
    a <- gate init0 ();
    b <- gate init0 ();
    a <- gate H a;
    gate CNOT (a,b)
alice : Circ(qubit⊗qubit, bit⊗bit) =
  box (q,a) =>
    (q,a) <- gate CNOT on (q,a)
    q     <- gate H q;
    x     <- gate meas q;
    y     <- gate meas a
    output (x,y)
bob : Circ(bit⊗bit⊗qubit, qubit) =
  box (x,y,b) =>
    (y,b) <- gate (bit-control X) (y,b);
    (x,b) <- gate (bit-control Z) (x,b);
    ()    <- gate discard y;
    ()    <- gate discard x;
    output b
teleport : Circ(qubit,qubit) =
  box q =>
    (a,b) <- unbox bell00 ();
    (x,y) <- unbox alice (q,a);
    unbox bob (x,y,b)
```

Figure A.22: Example QWIRE source code: quantum teleportation
(source: University of Pennsylvania)

TWIST

TWIST is a language created at MIT’s CSAIL lab that enforces how qubits are entangled or not, handles the notion of purity (for a set of qubits not influenced by others) and enables the creation of safer quantum programs. It introduces µQ, a functional language featuring classical control and quantum data (in the style of the Quantum Lambda Calculus).

Since it is not a full quantum programming language (it does not provide the means for building larger quantum circuits), one has first to build the quantum circuit with TWIST, do all the appropriate analyses, and then build the quantum circuit with a full-blown quantum language in order to actually be able to emulate it and/or run it on a real quantum processor. While that’s just a little extra work for small quantum circuits, it quickly becomes a challenge when designing larger quantum circuits. Also, compared to quantum languages such as OpenQASM, TWIST source code is noticeably lengthier.

Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs 0:7

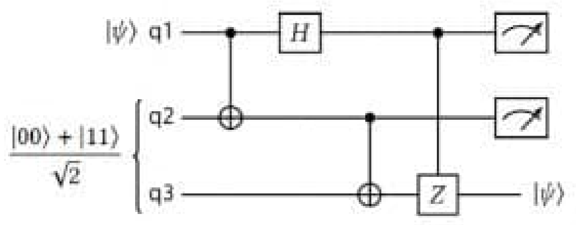


Fig. 1. Teleportation protocol with deferred measurement expressed as a quantum circuit. The inputs are q1, the qubit to be teleported, and a Bell pair of two qubits. The circuit applies conditional-NOT (CNOT) and conditional-Z (CZ) gates to q3. It measures two qubits and outputs one with the teleported state of q1.

```
fun teleport (q1 : qubit) : qubit =
  let (q2 : qubit, q3 : qubit) = bell_pair () in
  let (q1 : qubit, q2 : qubit) = CNOT (q1, q2) in
  let q1 : qubit = H (q1) in
  let (q2 : qubit, q3 : qubit) = CNOT (q2, q3) in
  let (q1 : qubit, q3 : qubit) = CZ (q1, q3) in
  let _ : bool * bool = measure (q1, q2) in q3
```

Fig. 2. Implementation of the teleportation circuit of Figure 1 as a quantum program.

Figure A.23: Example TWIST source code: quantum teleportation (source: Quantum Zeitgeist)

Appendix B – Overview of quantum emulators

This appendix provides a brief description (in alphabetical order) of a representative selection of quantum emulators.

Amazon Bracket emulators

Amazon Bracket emulators include the DM-based DM-1, the SV-based SV-1 and the TN-based TN-1 emulators.

Cirq emulators

Google Quantum AI 's Cirq QSDK comes with built-in open-source Python emulators for testing small quantum circuits. The two main types of emulations that Cirq supports are pure state and mixed state. The pure state emulations are supported by cirq.Simulator and the mixed state emulators are supported by cirq.DensityMatrixSimulator.

cuStateVec and cuTensorNet

Nvidia develops and maintains the cuStateVec (quantum state vector-based) and cuTensorNet (tensor network-based) quantum emulators as part of its cuQuantum QSDK that runs on top of Nvidia GPGPUs.

FermionIQ

FermionIQ delivers quantum circuit emulators as a SaaS platform to design and test quantum algorithms at scale.

IQS

Intel Quantum Simulator (IQS), aka qHiPSTER, is an open-source high-performance generic qubit quantum circuit emulator. It supports up to 42 pure-state qubits in quantum state vector emulation mode.

Jet

Jet is a cross-platform C++ and Python-based library created by Xanadu for emulating quantum circuits using tensor network contractions. It has built-in support for concurrent tensor contractions on CPUs and GPUs. In addition, Jet integrates seamlessly with PennyLane, providing for quantum optimisation for Quantum Machine Learning (QML) applications.

Lightning

Lightning is a high-performance quantum circuit emulator created by Xanadu, designed to provide high performance for Quantum Machine Learning (QML) applications. Written in C++ and accessible via Python, there are two emulators in the Lightning family: lightning.qubit and lightning.gpu. Lightning ensures quick emulation of large workflows, whether using GPUs locally or running on cloud-based supercomputers. In addition, Lightning integrates seamlessly with PennyLane, providing cutting-edge techniques for optimisation of QML workflows.

Pulser

Pulser is a framework developed by Pasqal for composing, emulating and executing pulse sequences for neutral-atom qubit-based quantum processors. The pulser_simulation extension provides tools for classical emulation (using Python QuTiP libraries) to aid in the development and testing of new pulse sequences.

PyQVM

Rigetti Computing's Python Quantum Virtual Machine (PyQVM) is a flexible and efficient emulation library for Quil, which is contained in its Forest QSDK. PyQVM evaluates Quil programs (parsed and compiled by quilc) on a virtual machine that can model various characteristics of a true quantum computer (though without needing access to it).

QCE

Quantum Computer Emulator (QCE) is a software tool developed at the University of Groningen that emulates various types of quantum circuits. QCE provides an environment to debug and execute quantum algorithms under realistic experimental conditions. The QCE software consists of a GUI and the quantum circuit emulator itself.

Qiskit Aer

Qiskit Aer (open-source) from IBM Q supports quantum circuit emulations in state quantum vector mode, density matrix mode and also in the less common matrix product state mode (adapted to weakly entangled states) and stabilizer mode (supporting only Clifford group gates).

QLM

Quantum Learning Machine (QLM) is a proprietary, Intel platform-based, extensive quantum circuit emulation suite developed and maintained by Eviden (formerly Atos[11]). It has been widely adopted by quantum developers worldwide.

myQLM is a freely available subset of the full QLM suite. It supports emulation for the three quantum computing paradigms: quantum simulation, quantum annealing and gate-based quantum computing.

qsim

Google Quantum AI's open-source qsim emulator can simulate up to 30 qubits on a laptop and up to 40 qubits in Google Cloud.

QuEST

The open-source Quantum Exact Simulation Toolkit (QuEST) was developed at the University of

[11] Eviden is an Atos Group company that brings together its digital, cloud, big data and security business lines.

Oxford. QuEST is a high performance emulator of quantum circuits, state-vectors and density matrices. QuEST uses multithreading, GPU acceleration and distribution to run fast on laptops, desktops and networked supercomputers. It is stand-alone, requires no installation, and is trivial to compile and get running.

QuIDDPro

QuIDDPro (open-source) was developed at the University of Michigan. It is an easy-to-use computational interface for generic quantum circuit emulation. It supports quantum state vectors, density matrices, and related operations using the Quantum Information Decision Diagram (QuIDD) data structure. QuIDDPro does not always suffer from the exponential blow-up in size of the matrices required to simulate large quantum circuits. As a result, QuIDDPro is significantly faster and uses significantly less memory compared to other generic emulation methods for quantum circuits with (many) more than ten qubits.

Quirk

Quirk (open-source) was developed by Craig Gidney (now at Google). It runs in a browser and even on a smartphone.

QuTiP

QuTiP is open-source software for emulating the dynamics of open quantum systems. The QuTiP library depends on the NumPy, Scipy and Cython numerical packages. In addition, graphical output is provided by Matplotlib. QuTiP aims to provide user-friendly and efficient numerical emulations of a wide variety of Hamiltonians (including those with arbitrary time-dependence), as commonly found in a wide range of physics applications such as quantum optics, trapped ions, superconducting circuits and quantum nanomechanical resonators. QuTiP is freely available for use and/or modification on Linux, Mac OSX and Windows platforms.

QVM

Google Quantum AI's open-source Quantum Virtual Machine (QVM) emulates a Sycamore quantum circuit with high accuracy.

QX Emulator

QX Emulator is an open-source quantum state vector-based quantum circuit emulator developed at QuTech. It allows quantum algorithm designers to emulate the execution of their quantum circuits on a quantum computer (up to 34 qubits). The emulator defines a low-level quantum assembly language named Quantum Code which allows users to describe their quantum circuits in a simple textual source code file; the source code file is then used as the input of the emulator which executes its content.

SandBox AQ emulator

The SandBox AQ emulator is a DMRG-based quantum circuit emulator developed together with Google Quantum AI.

SimulaQron

SimulaQron (open-source) was developed by QuTech. It runs on their Quantum Inspire QSDK with two quantum processors using 2 qubits (Spin-2) and 5 qubits (Starmon-5) and on two hardware emulators supporting 26, 31 and 34 qubits.

Appendix C – Overview of Quantum Software Development Kits

This appendix provides a brief description (in alphabetical order) of a representative selection of Quantum Software Development Kits (QSDKs), including those from the quantum computing vendors of Chapter 3.

Note

The name of a QSDK is sometimes the same as the name of its principal quantum programming language or the name of the quantum computing service provided by the vendor.

Amazon Braket

The Amazon Braket Python QSDK (Figure C.1) is an open-source library that provides a framework that can be used to interact with quantum computing hardware devices and quantum circuit emulators through Amazon Braket. It provides access to multiple different types of quantum computers.

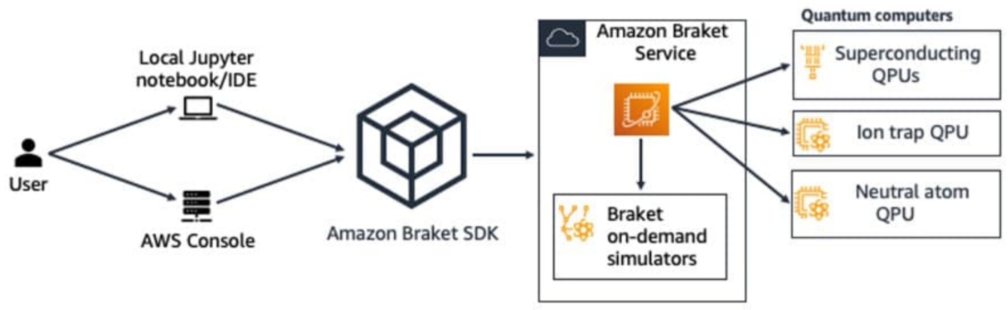


Figure C.1: Amazon Braket (source: AWS)

There are a couple of different ways to use Amazon Braket:

1. Use a managed notebook instance via the AWS console, which runs a Jupyter notebook hosted on AWS servers that comes with the Braket SDK and other useful libraries pre-installed.

2. Use a Jupyter notebook or IDE in a local development environment by installing the Braket SDK and setting up the AWS Command Line Interface (CLI) on a workstation.

Alternatively, it is possible to use one of the Amazon Braket plugins, such as for example PennyLane or Qiskit plugins, to access Amazon Braket devices.

Cirq

Google Quantum AI's Cirq QSDK (figure C.2) is an open-source project developed by Google Quantum AI, which uses the Python programming language to create and manipulate NISQ quantum circuits. Programs written in Cirq can be run on many different quantum computers, including those from AQT, Google Quantum AI, IonQ, Pasqal and Rigetti Computing.

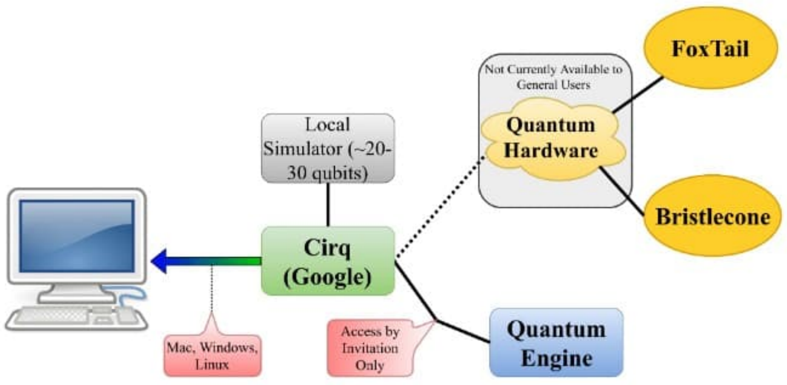

FoxTail
Not Currently Available to General Users
Quantum Hardware
Bristlecone
Local Simulator (~20-30 qubits)
Cirq (Google)
Mac, Windows, Linux
Access by Invitation Only
Quantum Engine

Figure C.2: Cirq QSDK components (source: Quantum Computing Report by GQI)

Google Quantum AI's FoxTail and Bristlecone quantum computers implement a CZ gate as their two-qubit unitary (as opposed to a CNOT gate as is common with other quantum computer architectures).

Cirq also supports Google's AI Quantum Sycamore quantum computer and provides two built-in quantum circuit emulators.

Several companies have used Cirq in collaboration with Google for various projects. For example, QC Ware used Cirq for implementing QAOA quantum algorithms and Cirq integrates with OpenFermion, an hardware-agnostic library for simulating fermionic systems on quantum computers.

cuQuantum

Nvidia developed the cuQuantum QSDK running on top of their GPGPUs (Figure C.3) thatt implements quantum circuit emulation.

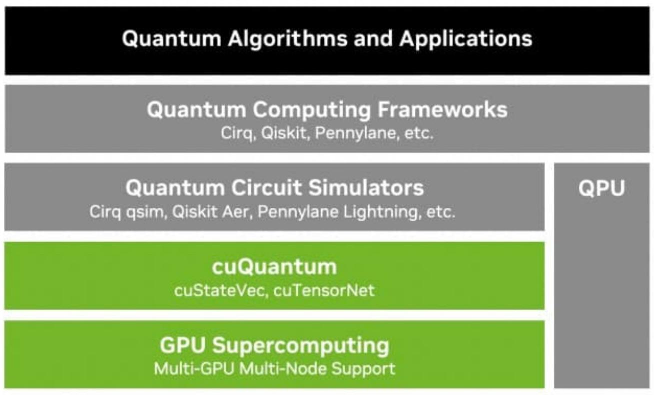

Quantum Algorithms and Applications
Quantum Computing Frameworks
Cirq, Qiskit, Pennylane, etc.
Quantum Circuit Simulators
Cirq qsim, Qiskit Aer, Pennylane Lightning, etc.
QPU
cuQuantum
cuStateVec, cuTensorNet
GPU Supercomputing
Multi-GPU Multi-Node Support

Figure C.3: cuQuantum QSDK architecture (source: Nvidia)

The cuQuantum QSDK contains both a quantum state vector-based emulator (cuStateVec) with tens of qubits and a less resources-hungry tensor network-based emulator (cuTensorNet) that supports up to thousands of qubits. Nvidia also integrated cuStateVec into qsim, Google Quantum AI's state vector-based simulator that can be used through Cirq, and into IBM Q's Qiskit Aer.

Nvidia also developed a quantum compiler, nvq++, which targets the Quantum Intermediate Representation (QIR), a low-level quantum language specification covering hybrid classical /quantum computing needs. It is supported by the Linux Foundation led QIR Alliance with contributions from Microsoft, ORNL, Quantum Circuits Inc., Quantinuum and Rigetti Computing.

Forest

Rigetti Computing's Forest QSDK (Figure C.4) is an open-source hybrid classical/quantum architecture that is optimised for NISQ quantum computers. It includes the pyQuil quantum programming language, the Quil compiler (quilc) and the PyQVM quantum circuit emulator.

Quantum programs written in Quil can be executed on any implementation of a Quantum Abstract Machine (QAM), such as a Rigetti Computing QPU or a Rigetti quantum circuit emulator. The quilc compiler compiles Quil source code for a given QAM according to its supported instruction set architecture.

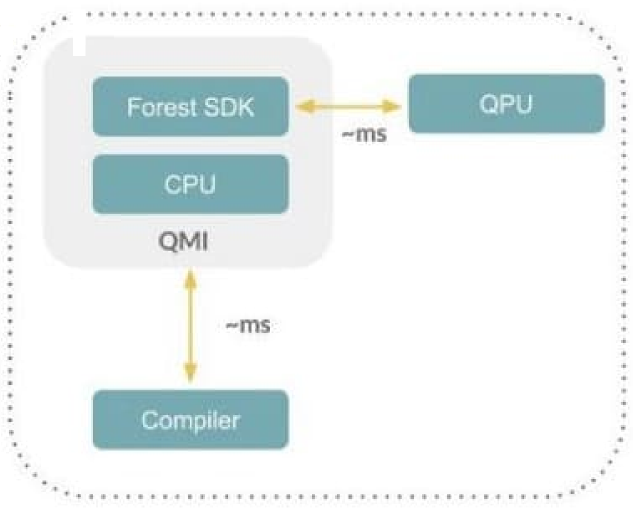


Figure C.4: Forest hybrid classical/quantum architecture (source: Rigetti Computing)

Ocean

D-Wave Systems' Ocean QSDK (Figure C.5) is an open-source suite of quantum development tools written mostly in the Python programming language. Ocean enables users to formulate problems in the Ising and Quadratic Unconstrained Binary Optimization (QUBO) models. Results can be obtained by submitting a quantum job to an online D-Wave quantum annealer or an hybrid solver in Leap, D-Wave System's real-time quantum application environment.

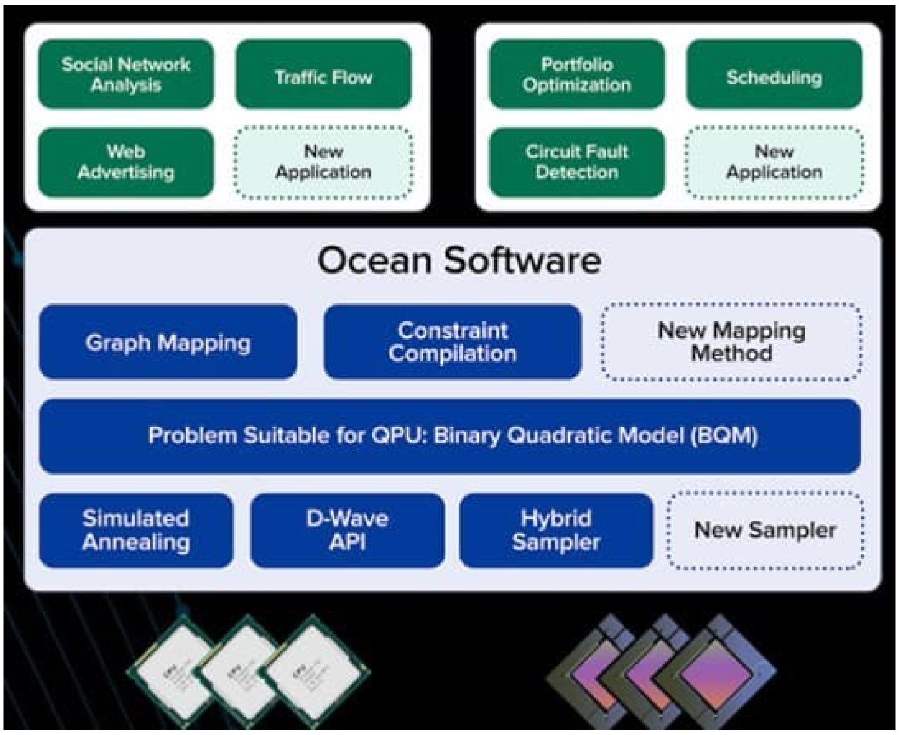


Figure C.5: Ocean quantum software development framework (source: D-Wave Systems)

Orquestra

Orquestra (figure C.6) is Zapata Computing's hardware-agnostic software development platform for quantum, quantum-inspired and classical generative model solutions. It is based on Zapata Computing's proprietary Generator-Enhanced Optimization (GEO) strategy, which is flexible to adopt any generative model from quantum to quantum-inspired or classical, such as for example Generative Adversarial Networks (GANs) or Quantum Circuit Born Machines (QCBMs).

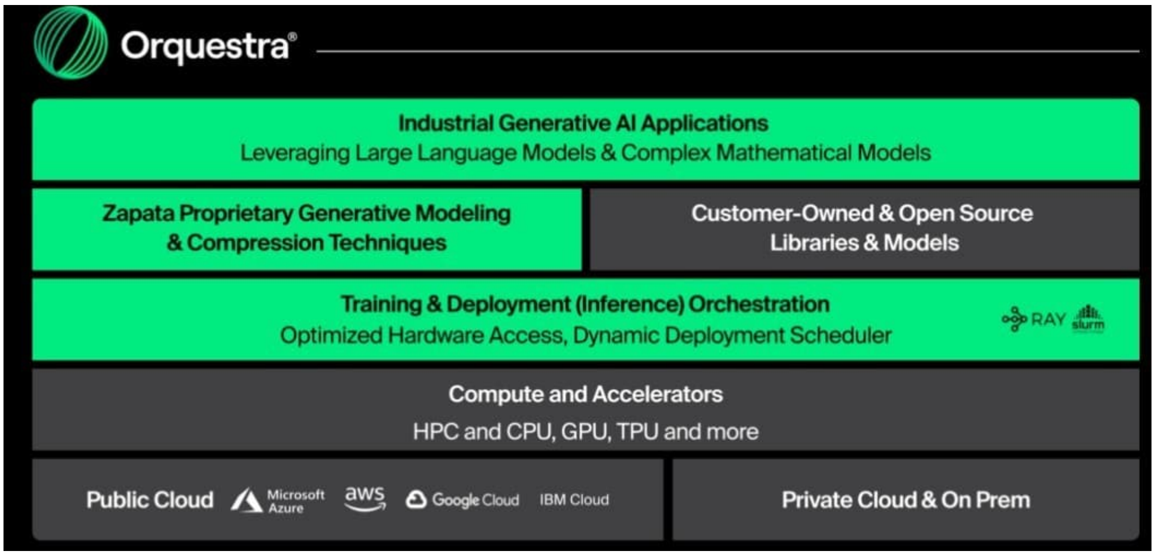

Orquestra®
Industrial Generative AI Applications
Leveraging Large Language Models & Complex Mathematical Models
Zapata Proprietary Generative Modeling & Compression Techniques
Customer-Owned & Open Source Libraries & Models
Training & Deployment (Inference) Orchestration
Optimized Hardware Access, Dynamic Deployment Scheduler
RAY
slurm
Compute and Accelerators
HPC and CPU, GPU, TPU and more
Public Cloud
Microsoft Azure
aws
Google Cloud
IBM Cloud
Private Cloud & On Prem

Figure C.6: Orquestra quantum software development platform (source: Zapata AI)

Orquestra provides various means for composing quantum computing workflows:

- use of built-in capabilities for quantum workflow design, execution and analysis, taking advantage of open-source algorithms and proprietary Zapata Computing algorithms;
- mix and match modules written with popular quantum libraries and languages including Cirq, PennyLane, PyQuil and Qiskit, leveraging a library of back-end solutions;
- compare the relative strengths of various back-end systems when applied to particular problems by benchmarking how quantum workflows perform across them.

From managing complex data records to automated parallelisation via container orchestration, Orquestra enables iterative, flexible experiments at scale. Users can submit quantum workflows to the Orquestra Quantum Engine (OQE) servers with command line and orchestrate workflow tasks across a full range of back-ends (quantum annealers, gate-based quantum processors, quantum circuit emulators and classical HPC resources). Both intermediate data and final results can be exported for analysis into either a Jupyter Notebook or a Microsoft Excel spreadsheet, or on the Tableau Platform.

Users can leverage the OQE servers and run work on either Zapata Computing's cloud, their public cloud of choice (AWS Market Place, Google Cloud Market Place, IBM Cloud or Microsoft Azure Quantum) or on on-premise machines.

PennyLane

PennyLane is an open-source cross-platform Python library developed by Xanadu for differentiable programming (Box C.1) of quantum computers.

Differentiable programming is a programming paradigm in which a computer program can be differentiated throughout via automatic differentiation. This allows for gradient-based optimisation of parameters in the program, often via gradient descent, as well as other learning approaches that are based on higher order derivative information.

Most differentiable programming frameworks, such as TensorFlow, work by constructing a graph containing the control flow and data structures in the program.

Box C.1: Differentiable programming

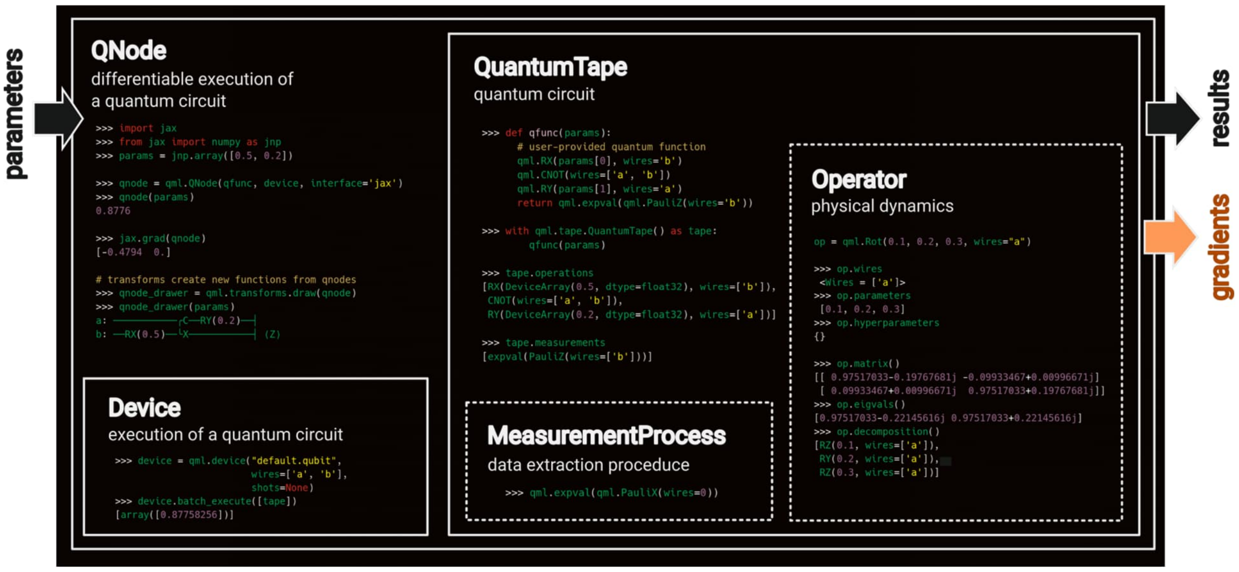
The central object in PennyLane (Figure C.7) is a QNode, which represents a node performing a quantum computation. Several QNodes may be part of a larger hybrid quantum-classical computation.

QNodes run quantum circuits on quantum devices, which may be simulators built into PennyLane or external quantum devices provided by plugins. The power of a QNode lies in the fact that it can be run in a "forwards" fashion to execute the quantum circuit, or in a "backwards" fashion in which it provides gradients.

The quantum circuit is specified by defining a quantum function, which is a Python function that contains quantum operations and measurements. Internally, the quantum function is used to construct one or more quantum tapes. A quantum tape is a context manager that records a queue of instructions required to run a quantum circuit.

PennyLane supports a comprehensive set of features, quantum circuit emulators, QPUs and community-led resources that enable users of all levels to easily build, optimise and deploy quantum-classical applications.

PennyLane enables seamless integration with Quantum Machine Learning (QML) tools. It provides users the ability to create models using NumPy, PyTorch or TensorFlow, and connect them with quantum computer back-ends available from Alpine Quantum Technologies (AQT), Google Quantum AI, IBM Q, Quantinuum and Rigetti Computing.



```
>>> import jax
>>> from jax import numpy as jnp
>>> params = jnp.array([0.5, 0.2])

>>> qnode = qml.QNode(qfunc, device, interface='jax')
>>> qnode(params)
0.8776

>>> jax.grad(qnode)
[-0.4794  0.]

# transforms create new functions from qnodes
>>> qnode_drawer = qml.transforms.draw(qnode)
>>> qnode_drawer(params)
a: ──────────╭C──RY(0.2)──┤
b: ──RX(0.5)──╰X──────────┤ ⟨Z⟩
```

```
>>> device = qml.device("default.qubit",
                        wires=['a', 'b'],
                        shots=None)
>>> device.batch_execute([tape])
[array([0.87758256])]
```

```
>>> def qfunc(params):
        # user-provided quantum function
        qml.RX(params[0], wires='b')
        qml.CNOT(wires=['a', 'b'])
        qml.RY(params[1], wires='a')
        return qml.expval(qml.PauliZ(wires='b'))

>>> with qml.tape.QuantumTape() as tape:
        qfunc(params)

>>> tape.operations
[RX(DeviceArray(0.5, dtype=float32), wires=['b']),
 CNOT(wires=['a', 'b']),
 RY(DeviceArray(0.2, dtype=float32), wires=['a'])]

>>> tape.measurements
[expval(PauliZ(wires=['b']))]
```

```
>>> qml.expval(qml.PauliX(wires=0))
```

```
op = qml.Rot(0.1, 0.2, 0.3, wires="a")

>>> op.wires
 <Wires = ['a']>
>>> op.parameters
 [0.1, 0.2, 0.3]
>>> op.hyperparameters
{}

>>> op.matrix()
[[ 0.97517033-0.19767681j -0.09933467+0.00996671j]
 [ 0.09933467+0.00996671j  0.97517033+0.19767681j]]
>>> op.eigvals()
[0.97517033-0.22145616j 0.97517033+0.22145616j]
>>> op.decomposition()
[RZ(0.1, wires=['a']),
 RY(0.2, wires=['a']),
 RZ(0.3, wires=['a'])]
```

Figure C.7: Pennylane architectural overview (source: PennyLane AI)

ProjectQ

ProjectQ (Figure C.8) is an extensible open-source project developed at the Institute for Theoretical Physics at ETH Zurich, which uses the Python programming language to create and manipulate quantum circuits. Results are obtained either using a quantum circuit emulator or by sending jobs to IBM Q quantum devices.

The ProjectQ compiler is modular and allows new compilers to be built by combining existing and new components. This design allows to customise intermediate quantum gate sets to improve optimisation for specific quantum algorithmic primitives. It also allows to adapt the compilation process to different quantum hardware architectures by replacing some of the compiler engines (including hardware-specific mappers), which maximises the re-use of individual compiler components.

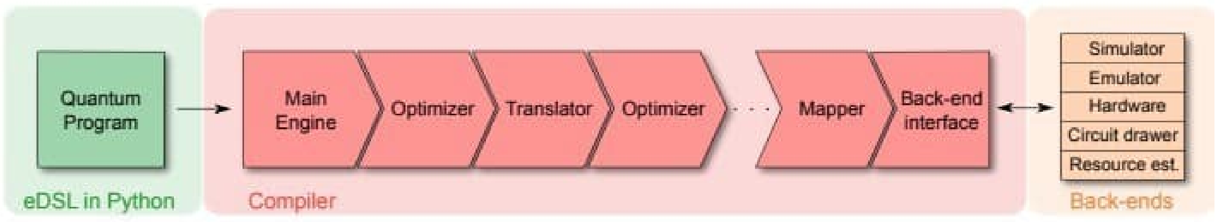


Figure C.8: ProjectQ full-stack software framework (source: ETH Zurich)

Qadence

Pasqal's Qadence is a user-friendly Python programming package designed to implement analogue, digital-analogue or digital quantum algorithms, tailored for Quantum Machine Learning (QML) workloads.

Qadence supports Digital-Analogue Quantum Computing (DAQC), a hybrid approach that aims to combine the precision of digital quantum computing (aka gate-based quantum computing) with the continuous control and interactions of analogue quantum computing. Pasqal's next-generation neutral-atom qubit-based quantum computers will be capable of natively executing DAQC algorithms.

Qadence stands out particularly in DAQC QML applications, supporting the native symbolic parameters, integration with PyTorch[12] automatic differentiation engine and advanced parameter shift rules for higher-order differentiation on real quantum devices. It provides developers with a simplified interface to accomplish the following:

- easily construct analogue and digital-analogue quantum algorithms;
- seamlessly transition from simulations to real quantum devices, such as Pasqal's neutral atom qubit-based quantum computers;
- easily express complex interaction among qubits and readily incorporate them into efficient executions on quantum simulation backends;
- translate certain types of analogue or digital-analogue operations into numerically efficient simulations similar to digital quantum circuits;
- general and higher-order parameter shift rules for efficient differentiation of digital-analogue quantum programs.

Qadence allows quantum algorithms to be structured in "blocks". Each block can represent a single quantum gate or a composition of gates. Large bocks are compositions of smaller blocks that can

[12] PyTorch is an open-source machine learning library used for developing and training neural network based deep learning models.

also be compounded while creating the quantum circuit (figure C.9). This approach is inspired by the Yao open-source framework for constructing quantum programs.

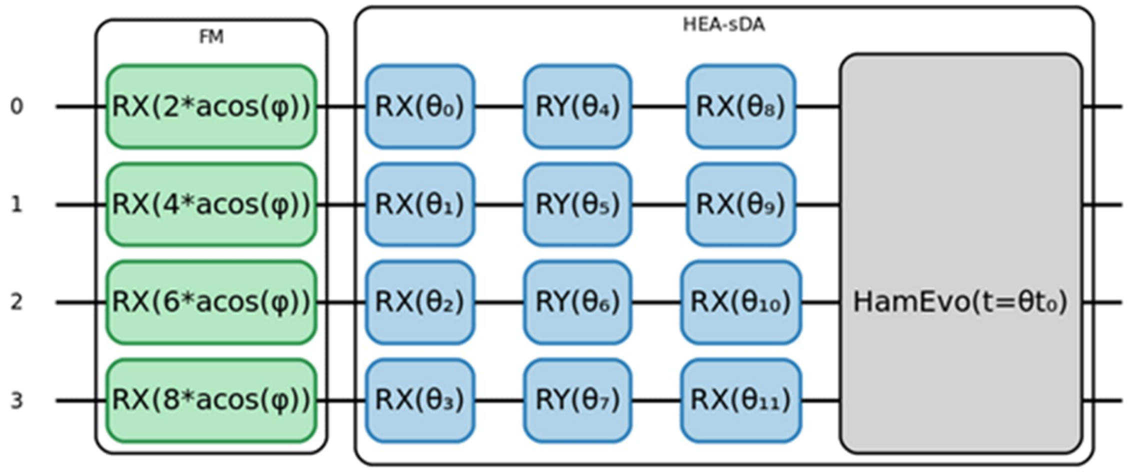


Figure C.9: Qadence blocks structure (source: Pasqal)

Pasqal plans to augment the Qadence library by incorporating noise channels, tailored error mitigation techniques for interacting qubit systems, and additional digital-analogue emulation modes.

Qiskit

IBM Q's Qiskit is an open-source QSDK for working with quantum computers at the level of quantum algorithms, quantum circuits, and qubit control and readout pulses.

Qiskit follows the quantum gate-based (aka quantum circuit-based) model for universal quantum computation and can in principle be used for any type of gate-based quantum computer (it currently supports superconducting qubits and trapped ions).

Qiskit provides the ability to develop quantum software both at the machine code level of OpenQASM and at abstract levels suitable for end-users without extensive quantum computing expertise.

The Qiskit QSDK contains the following components (Figure C.10):

- High-level applications that target specific domains and plug into the tools used by experts:
  - Qiskit Optimization;
  - Qiskit Machine Learning;
  - Qiskit Finance;

- Qiskit Nature.

- Qiskit Aqua: library of cross-platform NISQ quantum algorithms.

- Qiskit Patterns: pre-existing hybrid classical-quantum application templates for execution on either on-premises HPC infrastructures or on IBM's Elastic Cloud "quantum serverless" CPU/GPU/QPU infrastructure.

- Low-level applications:
  - Qiskit Metal: framework for engineering and designing superconducting quantum devices;
  - Qiskit Ignis: framework for understanding and mitigating noise in quantum circuits and devices;
  - Qiskit Dynamics: provides access to different numerical methods for solving differential equations;
  - Qiskit Experiments: contains tools with a focus on running experiments, analyses and calibration of pulse schedules (e.g. for implementing error mitigation techniques).

- Qiskit Terra: provides the core Qiskit QSDK capabilities.

- Qiskit Aer: provides a set of quantum circuit emulators.

- Hardware providers: support for IBM Q quantum computers and other quantum computers (AQT, IonQ, etc.).

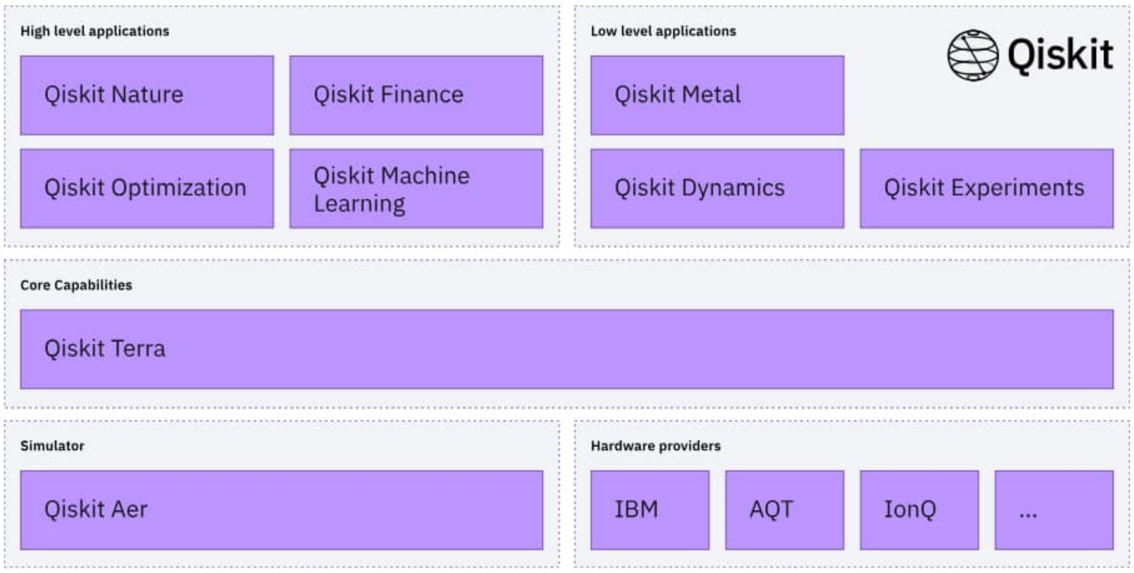


Figure C.10: Qiskit QSDK components (source: qiskit.org)

Qiskit provides the following execution modes:

- single-job mode;
- iterative mode;
- batch mode, including multi-job workflows;
- session mode, including multiple-session windows.

Note

The IBM Q Qiskit QDSK component landscape is constantly changing and hard to follow.

Quantify

Quantify is a Python-based, high-level data acquisition platform developed by Qblox. It is focused on providing all the necessary tools for quantum computing experiments. It is built on top of QCoDeS. The simple software framework enables setting-up typical characterisation experiments and advanced experimental procedures.

Quantum Development Kit

The Quantum Development Kit (QDK, see Figure C.11) is an open-source set of tools developed by Microsoft. Quantum programs can be written and run within Visual Studio using the quantum programming language Q#.

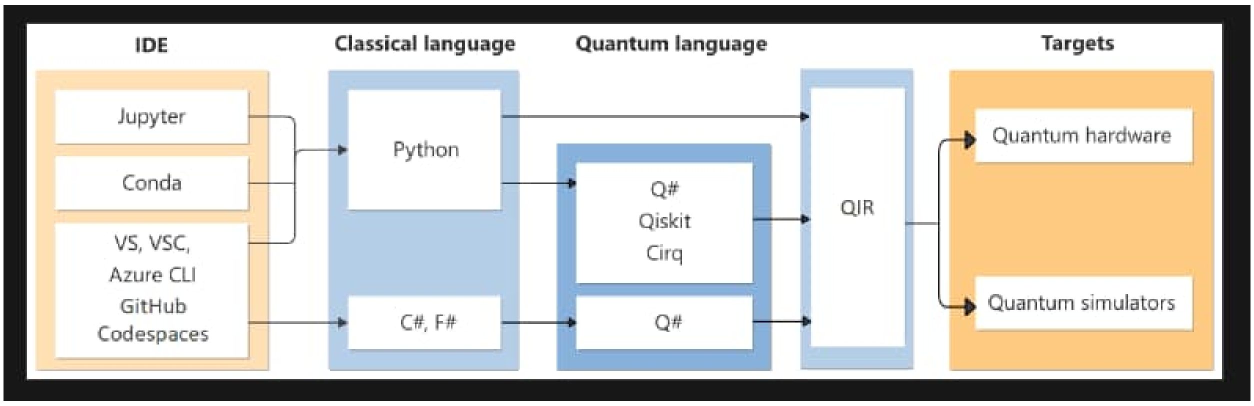


Figure C.11: QDK development environment (source: Microsoft)

Programs developed with the QDK can be run in Microsoft's Azure Quantum on quantum computers from IonQ, Pasqal, Quantinuum and Rigetti Computing, and also on various quantum circuit emulators.

QDK consists of the following components:

- Q# programming language and libraries;
- APIs for using Python and .NET languages (C#, F# and VB.NET) with Q#;

- IQ# kernel for running Q# on Jupyter Notebooks;
- extensions for Visual Studio Code (VSC) and Visual Studio (VS);
- Python packages to submit Cirq, Q# and Qiskit quantum applications to the Azure Quantum service;
- Azure CLI extension to manage the Azure Quantum service and submit Q# applications.

Quantum Inspire

Quantum Inspire (QI, see Figure C.12) is a quantum computing platform designed and built by QuTech. The goal of Quantum Inspire is to provide users access to various quantum hardware technologies to perform quantum computations.

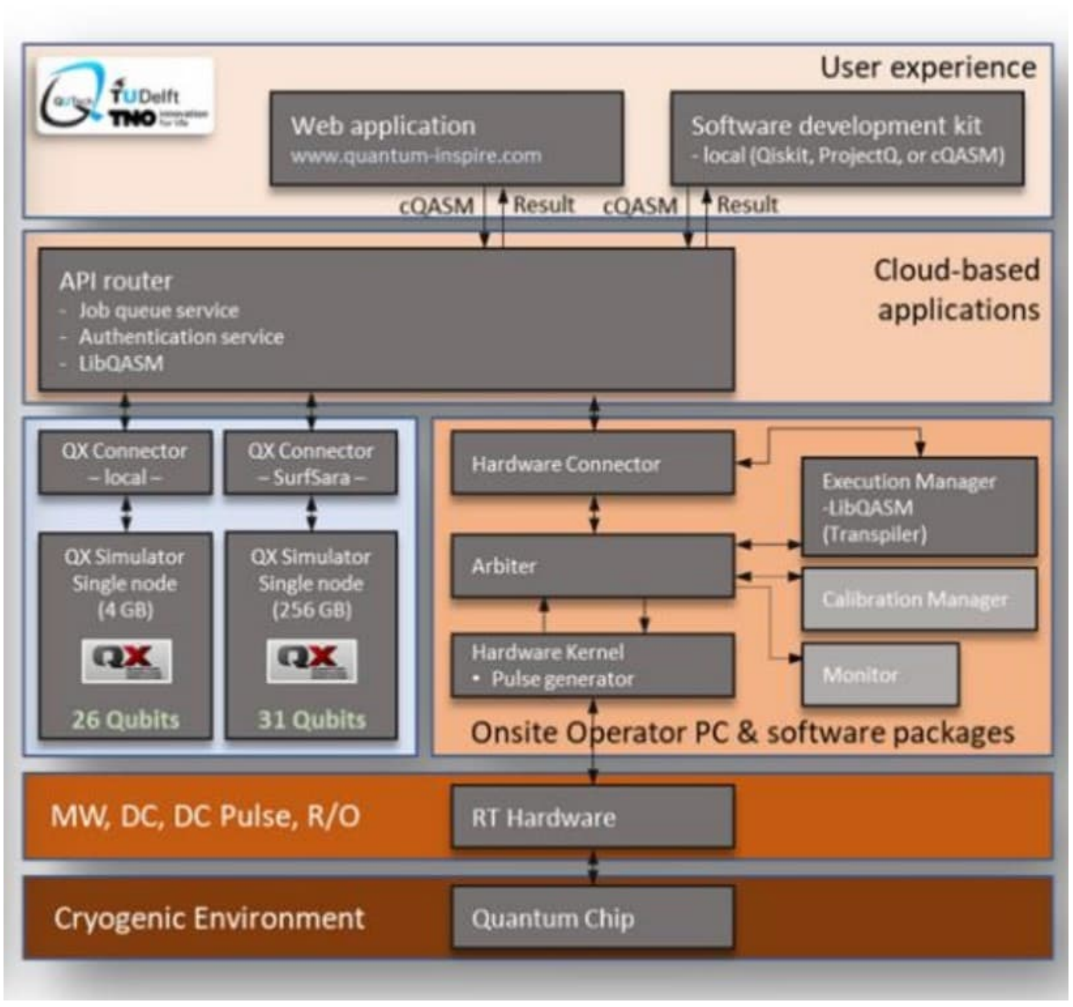


Figure C.12: QI architecture (source: QuTech)

QI provides a variety of ways to program quantum algorithms, execute these algorithms and examine the results. It includes a graphical interface to program in QASM and to visualise operations in quantum circuit diagrams.

QI enables access to two quantum processor back-ends, 2-bit semiconductor electron spin qubit-based (Spin-2) and 5-qubit transmon qubit-based (Starmon-5), and to three versions of the QX Emulator back-end (up to 26 qubits on a commodity cloud server, up to 31 qubits on the Cartesius SURFsara single-node computer and up to 34 qubits on the Lisa SURFsara cluster computer[13]).

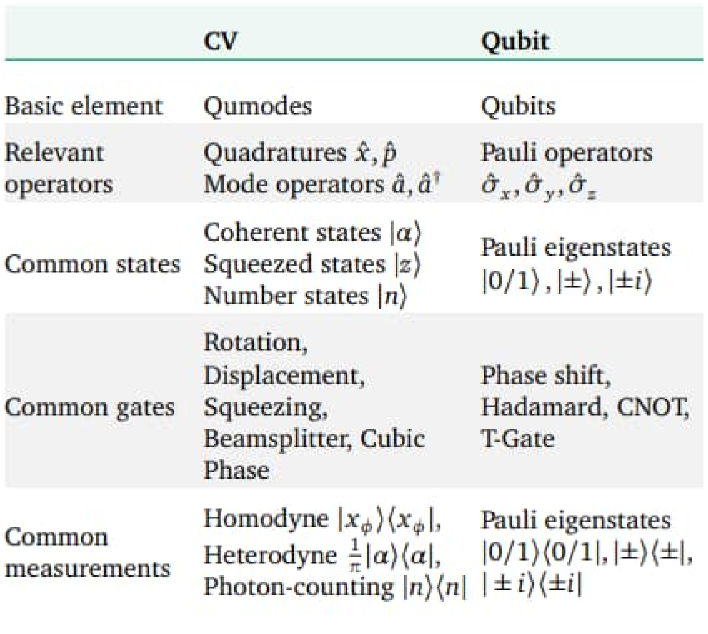
The Quantum Inspire QSDK consists of:

- a Python interface to the QI platform, which makes it possible to programmatically generate quantum circuits and process the results;
- a sub-package for writing algorithms in the ProjectQ quantum programming language;
- a sub-package for writing algorithms in the Qiskit quantum programming language.

Strawberry Fields

Xanadu's Strawberry Fields QSDK is an open-source Python library for designing, simulating, and optimising Continuous Variable (CV) quantum circuits for photonic quantum processors. It is the library for executing programs on Xanadu's X-Series quantum processors, which are based on qumodes instead of qubits (Table C.1). In addition, three emulators are provided.

Furthermore, Strawberry Fields provides built-in quantum application components, i.e. high-level functionality for solving problems such as graph and network optimisation, ML and chemistry.



	CV	Qubit
Basic element	Qumodes	Qubits
Relevant operators	Quadratures $\hat{x}, \hat{p}$ Mode operators $\hat{a}, \hat{a}^\dagger$	Pauli operators $\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z$
Common states	Coherent states $\vert\alpha\rangle$ Squeezed states $\vert z\rangle$ Number states $\vert n\rangle$	Pauli eigenstates $\vert 0/1\rangle, \vert\pm\rangle, \vert\pm i\rangle$
Common gates	Rotation, Displacement, Squeezing, Beamsplitter, Cubic Phase	Phase shift, Hadamard, CNOT, T-Gate
Common measurements	Homodyne $\vert x_\phi\rangle\langle x_\phi\vert$, Heterodyne $\frac{1}{\pi}\vert\alpha\rangle\langle\alpha\vert$, Photon-counting $\vert n\rangle\langle n\vert$	Pauli eigenstates $\vert 0/1\rangle\langle 0/1\vert, \vert\pm\rangle\langle\pm\vert$, $\vert\pm i\rangle\langle\pm i\vert$

Table C.1: Comparison of qumode (CV) and qubit models (source: Xanadu)

[13] Resource requirements (predominantly the RAM size) typically grow exponentially with the number of emulated qubits!

Superstaq

Inflеqtion's Superstaq QSDK (Figure C.13) is a Python-based open-source quantum software platform that optimises the execution of quantum programs by cross-layer optimisation and extreme device physics-aware tailoring. Superstaq plugins are available for the Qiskit and Cirq SDK's.

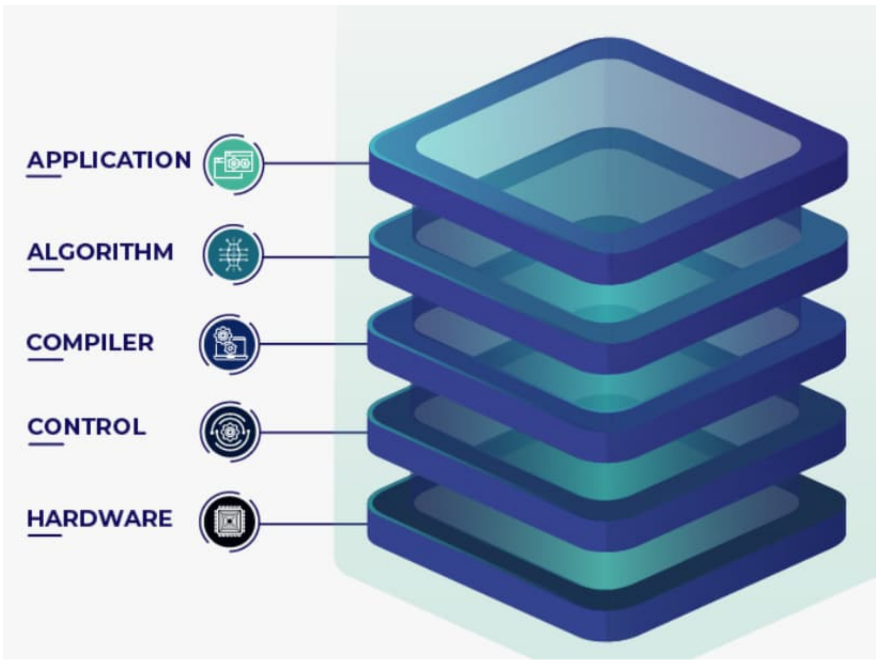


Figure C.13: Superstaq layers (source: infleqtion)

Optimisations are currently available for Inflection's (Hilbert) cold-atom quantum computer, AQT's, IBM's and Rigetti's (Aspen M-3) superconducting quantum computers and SNL's QSCOUT testbed trapped-ion quantum computer.

TKET

Quantinuum's TKET (aka t|ket⟩) QSDK (Figure C.14) is an open-source toolkit for the creation and execution of quantum programs for gate-based quantum computers and quantum circuit emulators[14]. Its quantum circuit optimisation routines allow users to extract as much power as possible from any of today's NISQ devices.

TKET is accessible through the PyTKET Python package, with extension modules providing compatibility with several quantum computers, quantum circuit emulators and popular quantum software libraries.

[14] TKET was created by the quantum software company Cambridge Quantum Computing (CQC). In 2021, CQC merged with the quantum computer manufacturer Honeywell Quantum Solutions (HQS) to form Quantinuum.

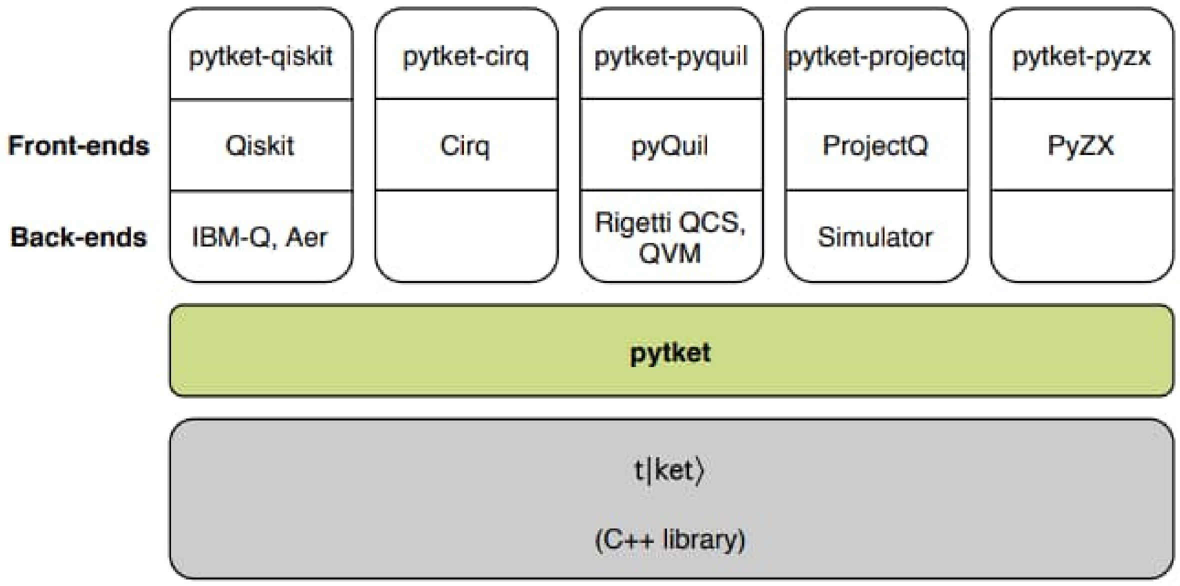


Figure C.14: TKET architecture (source: CQC)

PyTKET provides many shortcuts and higher-level components for building quantum circuits, including custom gate definitions, circuit composition, quantum gates with symbolic parameters and conditional quantum gates. PyTKET's flexible interface allows to include circuits defined in a number of quantum languages, including raw source code languages such as OpenQASM and Quipper, or embedded Python frameworks such as Google Quantum AI's Cirq, Microsoft's Q# and IBM Q's Qiskit.

XACC

XACC (Figure C.15) is an open-source framework for hybrid quantum-classical computing architectures developed at the US DoE's ORNL Science and Energy laboratory. It provides extensible language front-end and hardware back-end compilation components glued together via a polymorphic quantum intermediate representation. XACC supports quantum-classical programming and enables the execution of quantum kernels on D-Wave Systems QAs, IBM Q, IonQ and Rigetti Computing QPUs, as well as on a number of quantum circuit emulators.

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

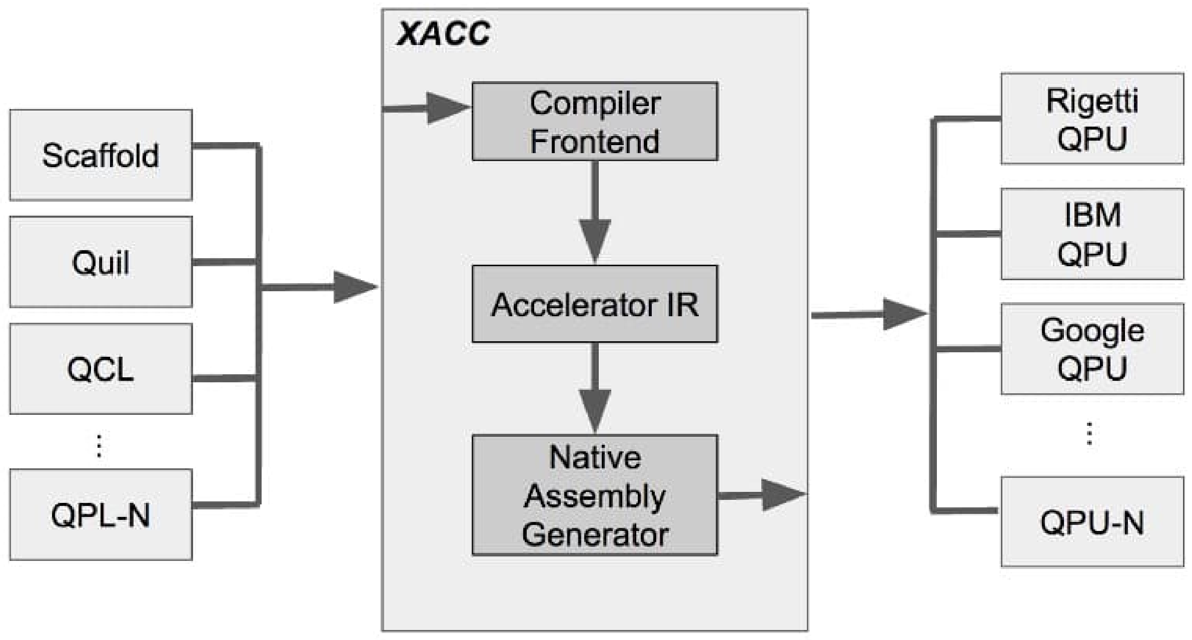


Figure C.15: XACC architecture (source: XACC)

Appendix D - References

[Ezratty 2023] Understanding Quantum Technologies - Sixth edition 2023

[Fact Based Insight 2021] Various presentations by David Shaw

[IEEE 2023] Quantum Computing Toolkit – From Nuts and Bolts to Sack of Tools

[NOREA 2024] Quantum Annealing Explained

[NOREA 2024] Quantum Computing Explained

Appendix E - Acronyms and abbreviations

ABNF	Augmented Backus-Naur Form
acos	arccosinus
ADC	Analogue-to-Digital Converter
AI	Artificial Intelligence
aka	also known as
ans	answer
AOD	Acousto-Optical Coupler
API	Application Programming Interface
app	application
AQT	Alpine Quantum Technologies
Aqua	Algorithms for quantum applications
ASIC	Application-Specific Integrated Circuit
AWG	Arbitrary Wave Generator
AWS	Amazon Web Services
BEC	Bose–Einstein Condensate
bit	binary digit
BNF	Backus-Naur Form
BQM	Binary Quadratic Model
BR	BRanch
C2QG	Classical Code to Quantum Gates
C#	C *sharp*
CA	C*a*lifornia
cand	candidate
CCD	Charge-Coupled Device
Circ	Circuit
Cirq	*Cirquit*
CLI	Command Line Interface
CMP	CoMPare
CNOT	Controlled NOT gate *(aka CX gate)*
CPU	Central Processing Unit
cQASM	common Quantum A*s*se*m*bly language

CQC	Cambridge Quantum Computing
cryostat	*from* cryo *meaning cold and* stat *meaning stable*
CSAIL	Computer Science and Artificial Intelligence Laboratory
ctrl	control
CUDA	Compute Unified Device Architecture
curl	*C*lient *URL*
CV	Continuous Variable
CX	Controlled X gate *(aka CNOT gate)*
Cython	C-extensions for *P*ython
CZ	Controlled Z gate
D	Deutsch gate
DAC	Digital-to-Analogue Converter
DAQC	Digital-Analogue Quantum Computing
DARPA	Defense Advanced Research Projects Agency
DC	Direct Current
def	define
DM	Density Matrix
DM1	Density Matrix 1
DMRG	Density Matrix Renormalization Group
DoE	Department of Energy
DSL	Domain Specific Language
e.g.	exempli gratia
EBNF	Extended Backus-Naur Form
EDP	Electronic Data Processing
eDSL	embedded Domain Specific Language
EQ	EQual
eQASM	executable Quantum A*s*se*m*bly language
est.	estimation
et al.	et alia
etc.	et cetera
ETH	Eidgenössische Technische Hochschule
F#	F *sharp*
FM	Feature Map

FMR	Fetch Measurement Result
FPGA	Field-Programmable Gate Array
FT	Fault-Tolerant
FTQC	Fault-Tolerant Quantum Computer
GAN	Generative Adversarial Network
GB	GigaByte
GEO	Generator-Enhanced Optimization
GPGPU	General-Purpose Graphics Processing Unit
gphase	gate phase
GPU	Graphics Processing Unit
GQI	Global Quantum Intelligence
groverDiff	Grover Diffusion operator
GUI	Graphical User Interface
H	Hadamard gate
had	Hadamard gate
HamEvo	Hamiltonian Evolution
HEA	Hardware Efficient Ansatz
Honey.	Honeywell
HPC	High-Performance Computing
HQS	Honeywell Quantum Solutions
HW	Hardware
i.e.	id est
IARPA	Intelligence Advanced Research Projects Activity
IBM	International Business Machines
id	identifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
Inc.	Incorporated
incl.	inclusive
init	initialisation
int	integer
IQ#	Interactive Q#
IQS	Intel Quantum Simulator

IR	Intermediate Representation
js	JavaScript
JSON	JavaScript Object Notation
KBW	Ket Bitwise Simulator
lab	laboratory
LAN	Local Area Network
Libs	Libraries
LIQUi|⟩	Language-Integrated Quantum Operations
macOS	Mac Operating System
MBQC	Measurement-Based Quantum Computing
meas	measurement
mgt.	management
Mgt	Management
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MPS	Matrix Product State
ms	millisecond
msmt	measurement
MW	Microwave
	Middleware
NISQ	Noisy Intermediate-Scale Quantum
nIterations	number of Iterations
NOREA	Nederlandse Orde van Register EDP-auditors
NP	Nondeterministic-Polynomial
NumPy	Numerical Python library
On Prem	On Premises
OpenQASM	Open Quantum Assembly language
OpenQL	Open Quantum Library
Ops	Operations
Opt	Option

OQC	Oxford Quantum Circuits
OQE	Orquestra Quantum Engine
OQIA	Origin Quantum *C*omputing Industry Alliance
OriginQ	Origin Quantum
ORNL	Oak Ridge National Laboratory
OS	Operating System
OSX	Operating System *10*
P	Phase gate
PC	Personal Computer
	Program Counter
pGCL	probabilistic Guarded Command Language
PHP	PHP: Hypertext Preprocessor
Prog.	Program
pyQuil	*P*ython *L*ibrary for Quil
PyQuil	Python *L*ibrary for Quil
PyQVM	Python Quantum Virtual Machine
PyTKET	Python TKET package
PyZX	Python ZX-calculus library
q	qubit
Q	Quantum
Q2B	Quantum-to-Business
Q#	Quantum *sharp*
Q.js	Quantum *J*ava*S*cript
QA	Quantum Annealer
QAM	Quantum Abstract Machine
QAOA	Quantum Approximate Optimization Algorithm
QASM	Quantum A*s*se*m*bly language
Qbsolv	Q*UBO* solver
QCaaS	Quantum Computing-as-a-Servie
QCBM	Quantum Circuit Born Machine
QCE	Quantum Computer Emulator
QCI	Quantum Circuits Inc.
QCL	Quantum Computation Language
QCoDeS	Quantum Copenhagen Delft Sydney

QCS	Quantum Cloud Services
	Quantum Computing Service
QDK	Quantum Development Kit
QEC	Quantum Error Correction
QEM	Quantum Error Mitigation
QFC	Quantum Flow Charts
qft	*Q*uantum *F*ourier *T*ransform
QFT	Quantum Fourier Transform
qGCL	quantum Guarded Command Language
qHiPSTER	Quantum High-Performance Software Testing Envi*r*onment
QI	Quantum Inspire
Qibocal	Qibo calibration
Qibojit	Qibo just-in-time
Qibolab	Qibo laboratory
Qibosoq	Qibo server on *QICK*
QICK	Quantum Instrumentation Control Kit
QIR	Quantum Intermediate Representation
Qiskit	Quantum *I*nformation *S*oftware *K*it
QLM	Quantum Learning Machine
QMASM	Quantum Macro A*s*se*m*bler
QMI	Quantum Machine Image
QML	Qt Modeling Language
	Quantum Machine Learning
QNode	Quantum Node
Qop	Quantum operator
QPE	Quantum Phase Estimator
QPL	Quantum Programming Language
QPS	Quantum Programming Studio
QPU	Quantum Processor Unit
QRAM	Quantum Random-Access Memory
QRNG	Quantum Random Number Generator
QSDK	Quantum Software Development Kit
qsim	quantum simulator
Qsim	Quantum simulator
QuArC	Quantum Architectures and Computation
qubit	quantum bit

QUBO	Quadratic Unconstrained Binary Optimization
qudit	quantum digit
QuIDD	Quantum Information Decision Diagram
QuEST	Quantum Exact Simulation Toolkit
Quil	Quantum instruction language
quilc	*Q*uil compiler
QuRE	Quantum Resource Estimator
qureg	quantum register
Qureg	Quantum register
QuTiP	Quantum Toolbox in Python
QVM	Quantum Virtual Machine
QWA	Quantum World Association
R	Rotational gate
R/O	Read*o*ut
RAM	Random-Access Memory
ReCirq	Research using Cirq
RFSoC	Radio Frequency System-on-Chip
RPC	Remote Procedure Call
rpcq	*RPC* for *Q*uantum
RT	Real-Time
RX	RX gate
RY	RY gate
SaaS	Software-as-a-Service
SAPI	Solver API
SBM	Simulated Bifurcation Machine
SciPy	Scientific and technical computing Python library
sDA	*S*tepwise Digital-Analogue
SDK	Software Development Kit
Sim.	Simulator
SLM	Spatial Light Modulator
SLOS	Strong Linear Optical Simulation
snd	send
SURF	Samenwerkende Universitaire Reken*f*aciliteiten
SURFsara	SURF - *S*tichting *A*cademisch *R*ekencentrum *A*msterdam

SV	State Vector
SV1	State Vector 1
t	time
T	T gate
TF	TensorFlow
TFQ	TensorFlow Quantum
TN	Tensor Network
TN1	Tensor Network 1
TNO	*N*ederlandse *O*rganisatie voor Toegepast Natuurwetenschappelijk Onderzoek
TPU	Tensor Processing Unit
TrueQ	True Quantum
TU	Technische Universiteit
U	U gate
UCL	University College London
UCSB	University of California at Santa Barbara
UI	User Interface
URL	Uniform Resource Locator
USC	University of Southern California
v2.0	Version 2.0
VB	Virtual Basic
VQE	Variational Quantum Eigensolver
VS	Virtual Studio
VSC	Virtual Studio Code
WAN	Wide Area Network
X	*P*auli X gate *(aka NOT gate)*
XACC	eXtreme-scale ACCelerator
Y	*P*auli Y gate
Z	*P*auli Z gate