A Survey of Quantum Programming Languages

Henry McNeil, CYBR8436, University of Nebraska, Omaha

August 11, 2020

1 Introduction

As quantum computers become increasingly sophisticated and their number of qubits approaches a level where they become useful for practical computations, knowledge of quantum programming is likely to become an increasingly sought after skill in the software engineering discipline. Quantum computing has applications to a broad number of areas, including cryptography, machine learning, statistics, chemistry, pure mathematics, and numerous others, and many quantum algorithms in these fields lead to superpolynomial increases in performance over classical algorithms[10]. Quantum algorithms typically involve a circuit-based approach in which gate operations are directly applied to incoming data, while in classical computing the direct circuit-level manipulation of data is often far removed from any commands being written.

One important aspect of all engineering disciplines is selecting the right tools for the job at hand, and in software engineering this involves selecting the right programming language, libraries, and development tools such as editors, package managers, and version control systems. Software engineers working on quantum applications will face the same challenges. While quantum programming languages are relatively new in comparison to classical programming languages, with the first quantum languages emerging in the late 1990s[7], there are still a large number in existence, and new languages are announced on a regular basis[1, 2]. Papers have been written to analyze the history and popularity of quantum programming languages[7, 11], but little work appears to have been done to address language selection from a software engineering standpoint. As such, in this paper we attempt to answer the following two research questions:

- **RQ1**: Which quantum programming language or framework should a professional software engineer select for enterprise development of quantum applications?
- **RQ2**: Which quantum programming language or framework should a student or hobbyist select for learning about quantum programming?

The evaluation and comparison of programming languages can be a heated topic bordering on the religious, so it should be noted that this paper is not meant to provide a source of absolute truth on the subject. Several aspects of analysis in this paper are purely qualitative and are based on the professional and academic experience of the author as applied to limited use of the languages in question.

2 Related Work

A great deal of work has been done on the topic of programming language analysis and comparisons. While this work focuses primarily on classical computing, work has been done to compare quantum languages as well.

Nanz et al[12] conduct an analysis of several programming languages spanning the procedural, object-oriented, functional, and scripting paradigms using programming challenge solutions found on the website Rosetta Code[1] for their analysis. This allows the authors to compare the solutions of the same problem in multiple languages, which allows for more robust analysis. The authors compare their selected languages in several ways, including lines of code (LOC), compile-time characteristics such as executable size, and runtime performance characteristics such as resource usage, execution time, and failure rate.

Alic et al[4] compare the object-oriented and functional paradigms. In their paper, they select the languages C# and Java from the object-oriented paradigm, and F# and Haskell from the functional paradigm. They compare the languages by analyzing lines of code and execution time on three different algorithms. The authors found that while the functional languages resulted in significantly shorter programs, having fewer lines of code doesn't mean better readability or performance. However, as the number of bugs in a program is correlated to the number of lines of code, more concise and expressive code may lead to more correct programs.

Garhwal et al[7] present an analysis of quantum programming languages, dis-

[1] `http://rosettacode.org/wiki/Rosetta_Code`

cussing 17 languages across multiple programming paradigms. Their work highlights features of each language and serves as an excellent survey of the current (as of 2019) quantum computing landscape, though it does not focus on comparing the languages in any way. In addition, it does not include an analysis of any libraries such as Cirq or Qiskit, though the authors do highlight Qiskit and QCL as candidates for the teaching of quantum computing.

LaRose[11] conducts a similar analysis to that found in this paper by comparing four "gate level quantum software platforms" including Forest, Q#, Qiskit, and ProjectQ. The author examines each of these platforms and provides analysis and discussion around their installation, documentation, syntax, quantum language, hardware integration, and simulators, as well as presenting a code sample for a random bit generator in each language. The paper explicitly compares the library support, hardware support, compilers, performance, and other features of the languages, and does so in significantly more depth than we accomplish within this work.

3 Methodology

In conducting this comparison of quantum programming languages, we adopt a simple methodology consisting of four stages:

- Select appropriate languages and libraries
- Implement and run a non-trivial algorithm
- Perform quantitative analysis of language/library features
- Conduct qualitative analysis based on the previous step

In selecting the languages and libraries used in this paper, we initially considered selecting languages of each of the different programming paradigms, namely imperative, functional, object oriented, and hybrid. Unfortunately the only object oriented quantum language we were able to locate was FJQuantum[6], which has minimal support and documentation, so we selected a different approach. Instead we looked at well known languages, as this would be a more realistic approach for a software engineer starting a new project, and selected from among those with sufficient learning resources and code examples available to meet our goals within the constraints of the academic term. This approach resulted in the selection of Microsoft's Q# language, Google's Cirq library for Python, IBM's Qiskit library for Python, and IBM's OpenQASM. The language Silq created by the SRI Lab at ETH Zurich was selected as well due to a desire to include the bleeding edge of quantum program-

ming languages. The language LiQUI|⟩[15] was also considered for inclusion, but was omitted due to time constraints and its similarities to Q#. While Cirq and Qiskit are libraries for the Python programming language, we will often refer to them as languages as well for simplicity.

In the author's experience, the best way to learn a programming language is by implementing a non-trivial program in the language. To this end, we selected Grover's algorithm as our target, as it is interesting, has practical applications in cryptography[8] and database searching, is sufficiently complex to demonstrate a selection of language features, and is simple enough that our analysis could focus on the languages themselves rather than a difficult algorithm. Additionally, numerous tutorials and sample implementations of the algorithm are available[5, 14, 9], which makes it simple to verify that a given implementation behaves as expected. For ease of implementation and general comparability we selected a two qubit implementation of the algorithm.

To quantitatively compare the languages, we chose to evaluate each language in five categories. First was the number and types of gates supported by the language, as well as the ability to implement custom gates with arbitrary functionality. Next was the types of control structures offered in the language, looking for the presence or absence of conditional statements and loops. Third was the ability to include code in functions, and the overall treatment of functions by the language. Fourth was the presence or absence of objects and object-oriented concepts within the language. Fifth, we analyze the number of lines of code used in each implementation. While the author's experience is that lines of code is generally not a helpful metric in practical situations due to other concerns such as maintainability of code by other users[2], it is commonly used in most other comparisons[4, 12]. Last was the ability of the language or library to interface with real quantum hardware.

For a qualitative comparison of the languages, we first examine their ease of installation and configuration to the point of running a simple "hello world" style of program. Next we consider the difficulty of learning the required syntax and semantics required for our algorithm implementation, in terms of documentation and other learning resources. Tools such as integrated development environments, syntax highlighting/checking/completion, and debugging are an integral part of any software engineer's arsenal, so we present a comparison in that aspect as well. Finally, we consider the languages in terms of community support and the difficulty of finding

[2]For example, the statement if (x) {y} else {z} could be written on one line or 4+, but most software engineers would likely say 4 lines is better.

answers to questions or issues encountered while working in the language.

4 Analysis

Table 1: Quantitative Comparison of Quantum Programming Languages

	Q#	**Silq**	**Cirq**	**Qiskit**	**OpenQASM**
Included Gates	21	9	24	41	29
Custom Gates	Y	Y	Y	Y	Y
Control Structures	Both	Both	Both	Both	Conditional
Functions	Y	Y	Y	Y	N
Objects	N	N	Y	Y	N
Hardware Support	Y	N	Y	Y	Y
Grover's LOC	59	20	41	37	25

4.1 Quantitative Comparison

Gate totals for each language and the other details of our quantitative comparisons can be found in Table 1. Our analysis of the included gates in each language was based on reading the documentation and manually counting. However, the presence of custom gates in all reviewed languages renders the number of gates included in their standard libraries somewhat moot, as a user should be able to implement any missing gates. While Silq has the lowest number of default gates, its ability to use qubits as the argument to a conditional meant that the explicit declaration of several traditional gates such as CNOT, CCNOT, and CSWAP is unnecessary in the language.

In examining the control structures of each language, we looked for the presence of conditionals and loops. All of the languages contained conditionals, and all but OpenQASM contained some form of loop statement. Function calls were supported by all languages except OpenQASM. The concept of objects exists in Python, and as such Qiskit and Cirq can be used in a somewhat object-oriented manner. Q# can integrate with object-oriented .NET languages, but on its own does not include objects – custom types are supported, however. Silq and OpenQASM do not include any object-oriented functionality.

Cirq, Qiskit, and OpenQASM are all capable of easily interfacing with cloud-connected quantum hardware, through the Google and IBM interfaces, respectively. Q# does not (at the time of this writing) include a publicly accessible way of interfacing with quantum hardware, though Microsoft is actively developing Azure Quantum to provide this functionality. Silq currently does not have any ability to interface with quantum hardware, nor were we able to find any indication that this functionality is being actively developed.

4.2 Qualitative Comparison

The target machine on which all of the languages were tested is a 2019 MacBook Pro with a 2.3 GHz 8-core Intel i9 processor and 32 GB of DDR4 memory, running OS X Catalina. All languages were simple to install and run locally, requiring a plugin for VSCode in the case of Q# and Silq, and a simple `pip install` command for Cirq and Qiskit. OpenQASM was included automatically with our installation of Qiskit, though the ability to run it via the IBM Quantum Experience meant that this was not directly used when working on our local machine. For writing code in Cirq and Qiskit, we used Jupyter Notebooks as well as Atom text editor, though these libraries should function just as well in PyCharm or another Python IDE.

Of the five selections examined, it was by far the easiest to learn the basics of Qiskit due to its simple circuit model and the excellent documentation and tutorials available through `https://qiskit.org`. Cirq was quite easy as well, though this may be due to its highly similar program structure to what Qiskit uses. OpenQASM does not have much documentation to describe its use, but due to the highly simplified nature of that language and the ability to view its code while visually assembling a circuit meant it was not difficult to learn enough to implement Grover's algorithm. Q# has excellent documentation available online, but due to its rich feature set and complexity it proved more challenging to complete our algorithm implementation in this language. Finally, Silq proved to be the most complex of the languages due to its complicated syntax and relative lack of documentation.

Q# had the best IDE support of the languages reviewed, and its VSCode plugin included syntax highlighting, autocomplete, syntax error checking, and limited debugging functionality – breakpoints could be used, but only limited aspects of program state could be examined. The plugin for Silq offered syntax highlighting and checking, but did not include autocomplete or IDE-based debugging. The Silq plugin does offer a statement by statement execution view of one's program however,

which we found to be extremely helpful and one of the language's best features. We did not experiment much with Qiskit or Cirq in an IDE, preferring to use Jupyter Notebooks instead, but we found most standard Python IDE features to be well supported for both of these libraries. OpenQASM had no IDE support, but the IBM Quantum Experience served as a good way of interacting with the language to the extent necessary.

The community support around Q# is very strong, and resources were available to provide answers to most challenges encountered while learning the basics of the language. Qiskit and Cirq both had a large number of community resources available as well. Silq and OpenQASM had virtually none, though many OpenQASM questions were included within Qiskit communities due to their close relationship.

Other qualitative language features we found beneficial were Qiskit and Cirq's circuit visualizations, Silq's statement-level execution information, and the Gitter and Slack channels for Q#.

5 Conclusion

We found that all of the languages were similarly easy to set up and start using, with the exception of OpenQASM being difficult to use locally but trivial through the IBM Quantum Experience. While the number of gates available varied considerably between languages, the basic operations were present in all cases and the ability to implement arbitrary custom gates makes this difference one of convenience rather than functionality. OpenQASM had the most limited functionality overall due to its lack of functions or control statements, making it impractical to write by hand for most complicated applications. Our recommendation would be to avoid programming directly in OpenQASM whenever possible, using a higher level language and converting code into it instead. Silq's lack of maturity in terms of community support and documentation, as well as its current lack of interaction with quantum hardware means we could not recommend it for an enterprise project at this time, though its powerful and unique language features would be worth exploring further in an academic context. As it is likely to continue active development in the future, upcoming changes may eliminate these concerns.

The remaining three candidates, Q#, Cirq, and Qiskit, exhibit similar functionality and a high level of maturity, and all three could be used as the answer to either **RQ1** or **RQ2**. However, in the interest of providing a more definitive answer, it is

not difficult to narrow down our selection to a single choice for each question. In addition to its language features, Q#'s integration with .NET and its accompanying development tools, community support, and robust libraries make it an ideal candidate for an enterprise quantum project, making it our choice for **RQ1**. Qiskit and Cirq have a similar feel in many ways, but due to Qiskit's excellent tutorials, the online textbook, and its easy integration with the IBM Quantum Experience and IBM's quantum computers, it was easily our choice for **RQ2** and is an excellent option for students or hobbyists to learn about and experiment with quantum programming.

5.1 Areas for Future Study

Due to the constraints of the academic term and the introductory nature of the course for which this work was completed, only a handful of languages were selected for this comparison, and none were exercised to their fullest capabilities. Many of these languages offer unique features which were not explored within the scope of this project, such as Silq's emphasis on uncomputation. An examination of Wikipedia's page on quantum programming[3] and other quantum computing resources[13] quickly demonstrates that this study is far from comprehensive, and more than 20 other quantum programming languages and libraries exist at the time of writing. While it is our belief that the languages selected are top tier in terms of maturity and features, it would be valuable to conduct a foundational of analysis on all existing languages while this number is still small enough to be reviewed by a few dedicated individuals. It would also be highly valuable to implement a project of greater complexity that pushes the limits of each language, as Grover's algorithm is certainly at the simpler end of quantum algorithms. Additional research could be conducted on the capabilities of each language in terms of operability with different quantum hardware, as technological breakthroughs could lead to a clear frontrunner in quantum computer implementations.

6 Appendix: Code Samples

In this appendix we present a code sample from each language for comparison purposes. For brevity only the Grover Diffusion Operator is reproduced in each example.

6.1 Q#

```
  ApplyToEach(H, input);
  ApplyToEach(X, input);
  H(input[1]);
  CNOT(input[0], input[1]);
  H(input[1]);
  ApplyToEach(X, input);
  ApplyToEach(H, input);
```

6.2 Silq

This code block was copied from Silq's very elegant test code at https://github.com/eth-sri/silq/blob/master/test/grover2.slq and was written for a general n-qubit case.

```
def groverDiffusion[n:!N](cand:uint[n])mfree: uint[n]{
  for k in [0..n) { cand[k] := H(cand[k]); }
  if cand!=0{ phase(pi); }
  for k in [0..n) { cand[k] := H(cand[k]); }
}
```

6.3 Cirq

This code block was taken from the Grover's algorithm example in chapter 8 of [9]

```
  c.append(cirq.H.on_each(*input_qubits))
  c.append(cirq.X.on_each(*input_qubits))
  c.append(cirq.H.on(input_qubits[1]))
  c.append(cirq.CNOT(input_qubits[0], input_qubits[1]))
  c.append(cirq.H.on(input_qubits[1]))
  c.append(cirq.X.on_each(*input_qubits))
  c.append(cirq.H.on_each(*input_qubits))
```

6.4 Qiskit

```
def diffusion(circuit, qr):
  for q in qr:
    circuit.h(q)
    circuit.x(q)
  circuit.h(qr[1])
  circuit.cx(qr[0], qr[1])
  circuit.h(qr[1])
  for q in qr:
    circuit.x(q)
    circuit.h(q)
```

6.5 OpenQASM

```
h q[0];
h q[1];
x q[0];
x q[1];
h q[0];
cx q[1],q[0];
h q[0];
x q[1];
x q[0];
h q[1];
h q[0];
```

References

[1] S. Jordan, "Quantum Algorithm Zoo." [Online]. Available: http://quantumalgorithmzoo.org/

[2] S. Garhwal, M. Ghorani, and A. Ahmad, "Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages," *Archives of Computational Methods in Engineering*, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s11831-019-09372-6

[3] "Realize the Quantum Possibilities of Tomorrow." [Online]. Available: https://www.quantum-machines.co/

[4] "What is Silq?" [Online]. Available: https://silq.ethz.ch/

[5] R. LaRose, "Overview and Comparison of Gate Level Quantum Software Platforms," *Quantum*, vol. 3, p. 130, Mar. 2019. [Online]. Available: https://quantum-journal.org/papers/q-2019-03-25-130/

[6] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 778–788, iSSN: 1558-1225.

[7] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2016, pp. 667–672.

[8] S. S. Feitosa, J. K. Vizzotto, E. K. Piveta, and A. R. Du Bois, "FJQuantum – A Quantum Object Oriented Language," *Electronic Notes in Theoretical Computer Science*, vol. 324, pp. 67–77, Sep. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066116300512

[9] D. Wecker and K. M. Svore, "LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing," *arXiv:1402.4467 [quant-ph]*, Feb. 2014, arXiv: 1402.4467. [Online]. Available: http://arxiv.org/abs/1402.4467

[10] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's Algorithm to AES: Quantum Resource Estimates," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, T. Takagi, Ed. Cham: Springer International Publishing, 2016, pp. 29–43.

[11] cgranade, "Run Grover's search algorithm in Q# - Quantum Development Kit - Microsoft Quantum." [Online]. Available: https://docs.microsoft.com/en-us/quantum/tutorials/search

[12] D. Voorhoede, "Code example: Grover's algorithm." [Online]. Available: https://www.quantum-inspire.com/kbase/grover-algorithm/

[13] J. Hidary, *Quantum Computing: An Applied Approach.* Springer Nature Switzerland, 2019.

[14] "Quantum programming," Jun. 2020, page Version ID: 964758447. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Quantum_programming&oldid=964758447

[15] D. Vogt-Lee, "desireevl/awesome-quantum-computing," Aug. 2020, original-date: 2018-01-23T10:07:44Z. [Online]. Available: https://github.com/desireevl/awesome-quantum-computing