Baltic J. Modern Computing, Vol. 10 (2022), No. 1, pp. 87–104
https://doi.org/10.22364/bjmc.2022.10.1.06

Comparing Quantum Software Development Kits for Introductory Level Education

Marija ŠĆEKIĆ,[1] Abuzer YAKARYILMAZ[2,3]

[1] Faculty for Information Technologies, Mediterranean University, Podgorica, Montenegro
[2] Center for Quantum Computer Science, University of Latvia, Rīga, Latvia
[3] QWorld Association, Tallinn, Estonia, https://qworld.net

marijascekic1982@gmail.com, abuzer.yakaryilmaz@lu.lv

Abstract. We initiate a study to overview and compare quantum software development kits (QSDKs) in terms of their usability for introductory level quantum education. In this work, we focus on Qiskit, ProjectQ, Cirq, and Forest. For comparison, we define six tasks based on QWorld's introductory tutorial called Bronze. We implement each task on these QSDKs. Besides, we check how easy it is to install them. According to our results, not every QSDK comes as a stand-alone Python package. This may create certain installation and execution problems. Visualization of quantum circuits may be poor or fail in some case. For the rest of tasks, all QSDKs are sufficient to work with.

Keywords: Quantum Software, Quantum Programming, Software Development Kits, Quantum Education, Qiskit, Cirq, ProjectQ, Forest

1 Introduction

There has been a global competition among the technological giants to build intermediate scale and reliable real quantum computers with an increasing interest of obtaining useful results by using them. In parallel, quantum software development kits (QSDKs) have been created for programming these devices (Hassija et al. (2020)). These QSDKs also come with quantum computer simulators, so the users can design and simulate their quantum programs on their personal computers. The possibility of such hands-on experiences has stimulated new pedagogical approaches to teach quantum computing and algorithms through quantum programming tasks. QWorld's introductory tutorial Bronze (Yakaryılmaz et al., (a, b)) and elementary level tutorial Silver (Salehi et al. (2019)), Qiskit Textbook (Abbas et al. (2020)), and Quantum Katas (WEB, a) are some of the example tutorials developed based on these approaches. Here we should also

emphasize the key role of using jupyter notebooks that increases the interaction significantly (e.g., Wootton et al. (2020)). Among all, Bronze was designed particularly for the introductory level education. Up to date, it has been used to organize around 80 workshops, online and on site, globally and locally (around 25 countries). We refer the reader to (Salehi et al. (2021)) for a study on the positive effects of the workshops using Bronze.

Bronze uses gate-based quantum programs, widely used in standard textbooks on quantum computing (e.g., Nielsen and Chuang (2000), Kaye et al. (2006)). One advantage of gate-based models is their compatibility with the existing quantum computers (using superconducting technologies) developed by the well-known manufacturers such as IBM, Google, or Rigetti. Thus, the written code and executed code on real machines are closely related, and so the programmers can directly work on the optimization of their code or using certain error correcting/mitigation techniques (Endo et al. (2018, 2020)).

Qiskit, Cirq, and Forest are the QSDKs created and supported by IBM, Google, and Rigetti (WEB, b, c, d), respectively; and ProjectQ is an academic project (Steiger et al. (2018)) with industrial support such as Huawei (WEB, e). In this work, we initiated a study for investigating their pedagogical usability and then compare them for introductory level education. Their scope is beyond the pedagogical purposes, and they can be compared from different perspectives (e.g., LaRose (2019), Vietz et al. (2021)). But, we find it valuable to evaluate how ready they are for the new comers, especially ones learning quantum concepts for the first time. Our work is an initial step towards this direction.

The versions of libraries we use in this work are Qiskit 0.29.0, Cirq 0.12.0, ProjectQ 0.7.0, and Forest 3.0.0 using pyQuil as the Python library.

We use a task-based approach for our study and give their list in Section 2. In Section 3, we give the details of the implementation of each task with an overview of QSDKs. Our findings are presented in Section 4. We close the paper (Section 5) by summarizing our findings as a table and discussing possible future directions.[1]

2 The list of basic functionalities

We use QWorld's Bronze as the base for defining our six mains tasks. We give the details of the tasks in Section 3, and here is a concise list of functionalities we check in each QSDK:

- Randomly generating very simple quantum circuits (using classical conditionals and loops)
- Executing quantum circuits and reading the measurement results
- Visualizing the quantum circuits
- Simplifying the design by using classical subroutines
- Reading the quantum state and unitary operator of the circuit
- Applying controlled quantum operators

[1] All code related to this work is accessible from the public repository: `https://gitlab.com/marijascekic/qsdk-comparison/-/tree/BJMC`

- Implementing intermediate measurements
- Applying classically controlled quantum operators
- Implementing quantum teleportation as a sophisticated protocol for the introductory level

Our main criterion is whether each of these functionalities is implemented or not by every QSDK. Our work consists of case studies where each case is defined by a task. It is clear that further tasks may also be added. As an initial study, we draw our borders with QWorld's tutorial Bronze.

3 Task details with an overview of QSDKs

Here we list each task under a subsection. The first task (next subsection) is the main one and so we also provide an overview of each library. For the rest of the tasks, we highlight only the most significant points. We refer the reader to check the repository of this work for the details.

3.1 Random quantum circuit outputting 8-bit binary numbers

This is a very basic example of using classical loops and conditionals when designing a quantum circuit.

The task is to design a quantum circuit with 8 qubits. For each qubit, we apply a NOT gate with probability 1/2. Then, we measure all qubits. The sub-tasks are

- to design the circuit,
- to draw the circuit, and,
- to read the result as a binary number.

3.1.1 Qiskit Quantum programs are designed as quantum circuits in Qiskit, which are composed by quantum and classical registers having qubits and bits, respectively. The quantum operators are applied to the qubits, and when measured, the results are written to the classical bits.

```
q  = QuantumRegister(8)
c  = ClassicalRegister(8)
qc = QuantumCircuit(q,c)
for i in range(8): #  i in {0,...,7}
    if randrange(2) == 0: # with probability 1/2
        qc.x(q[i]) # x is the NOT gate applied to q[i]
qc.measure(q,c) # measure q[i] => c[i]
```

The quantum circuit has a method called "draw" to visualize the quantum program.

```
qc.draw(output='mpl') # using Mathplotlib
```

A representative example is given in Figure 1. Qiskit uses separate objects and methods to execute circuits and process the outputs. Here we use local simulator called "qasm_simulator".

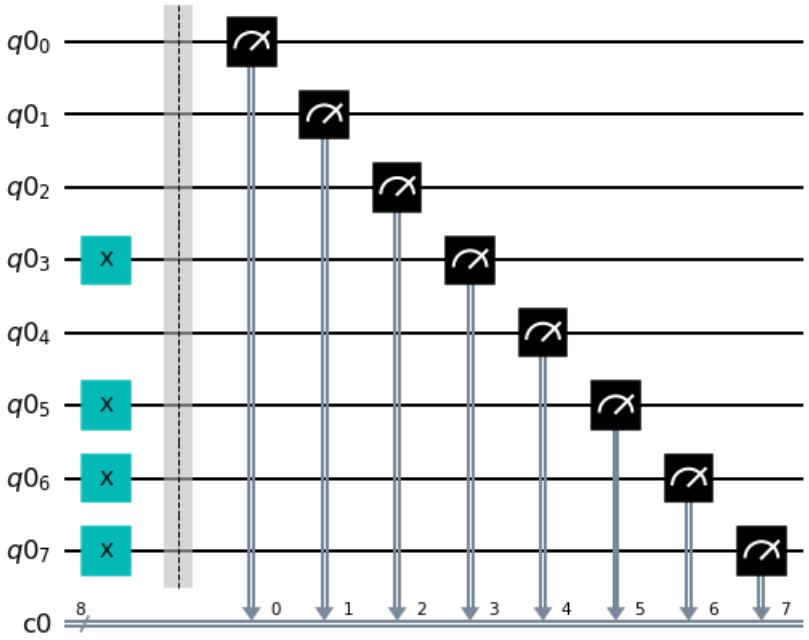

Fig. 1. An example quantum circuit in Qiskit

```
job = execute(qc,Aer.get_backend('qasm_simulator'),shots=1000)
counts = job.result().get_counts(qc)
print(counts)
```

The output of each execution is a binary number, and, a circuit can be executed several times. Thus, we obtain a dictionary of the binary outcomes with their frequencies.

```
{'11101000': 1000}
```

We can use default Python functions to convert the binary numbers in decimals. The user may use a loop to go through the dictionary.

3.1.2 Cirq Quantum programs are designed as quantum circuits in Cirq having only qubits (no explicit classical registers). On real quantum computers, the connectivity of qubits form a 2D topology, and Cirq allows us to define qubits as a line or as a grid.

In Cirq, circuit and quantum registers are separate objects. The circuit is constructed by appending the operators and qubits. Our example circuit has a line of qubits.

```
q = [cirq.LineQubit(i) for i in range(8)] # define qubits
qc = cirq.Circuit() # define circuit
for i in range(8):
    if randrange(2)==0:
        qc.append(cirq.X(q[i])) # define operator(s)
qc.append(cirq.measure(*q, key='result')) # define measurement
```

We also define a key value to access the measurement outcome of the specified qubits. A circuit or quantum register can be printed as text.

```
print(q) # print the quantum register
print(qc) # print the circuit
```

An example output circuit is given in Figure 2.

Cirq has a simulator object to execute the circuit a specified number of times.

```
simulator = cirq.Simulator()
result = simulator.run(qc, repetitions=10)
print(result)
print(result.histogram(key='result'))
```

The result gives the outcomes of each qubit separately (a single binary string for each qubit):

```
result=1111111111, 1111111111, 0000000000,  1111111111,
1111111111, 1111111111, 1111111111, 0000000000
```

To read the outcomes of all qubits as a single number, we use the key value, and so the result becomes a collection of decimal number with their frequencies:

```
Counter({222: 10})
```

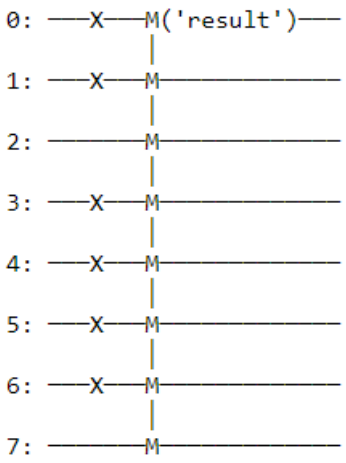

Fig. 2. An example quantum circuit in Cirq

3.1.3 Forest The library pyQuil has a "Program" object to define quantum circuits. The qubits are defined as a list. The program object also allows to define classical bits, to which the measurement outcomes can be written. Each operator is added to the program.

```
qc = Program()
q = range(8)  # qubits
for i in range (8):
    if randrange(2) == 0:
        qc+=Program(X(q[i])) # add operations
# define a classical register
ro = qc.declare('ro', memory_type='BIT', memory_size=8)
# measure qubits => bits
for i in range (8):
    qc+=MEASURE(i,ro[i])
```

A circuit can be drawn by using LaTeX.

```
from pyquil.latex import display
display(qc)
```

However, this uses the external programs "pdflatex" and "convert" and the installation of LaTeX including class and style files for "standalone", "geometry", "tikz", and "quantikz". An example circuit drawing is given in Figure 3. The quantum program may also be printed as a list of its elements. An example of a list obtained by the command "print (qc)" is given in Figure 4.

To execute the program on a local virtual machine, an external program "qvm" is used. Before that, the program is compiled by another external program called "quilc". These programs can be executed on separate consoles as

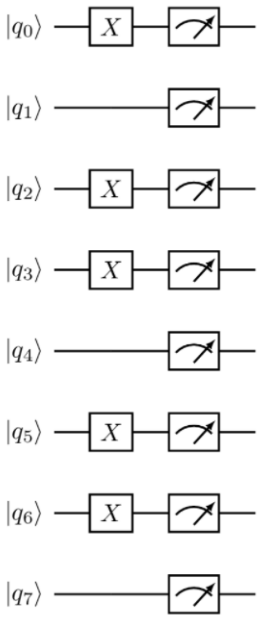

Fig. 3. A quantum circuit in pyQuil

```
X 0
X 2
X 3
X 5
X 6
DECLARE ro BIT[8]
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
MEASURE 3 ro[3]
MEASURE 4 ro[4]
MEASURE 5 ro[5]
MEASURE 6 ro[6]
MEASURE 7 ro[7]
```

Fig. 4. A program in pyQuil can be printed

```
> qvm -S
> quilc -S
```

or as “subprocess” in Python as

```
import subprocess
subprocess.Popen(["qvm", "--quiet", "-S"])
subprocess.Popen(["quilc", "--quiet", "-R"])
```

First, a “quantum computer” is selected, which is the local virtual machine running on a terminal in our case. Then, the circuit is compiled and executed on that machine.

```
from pyquil import get_qc
machine = get_qc('9q-square-qvm')
executable = machine.compile(qc.wrap_in_numshots_loop(10))
results = machine.run(executable).readout_data.get("ro")
print(results)
```

The result is a numpy array. The outcome of each execution is an array of binary values, and all outcomes form the result array. Therefore, the result should be processed.

```
[[1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]
 [1 0 1 1 0 1 1 0]]
```

3.1.4 ProjectQ ProjectQ is different from the others besides being a python library, *it features a lean syntax which is close to the mathematical notation used in quantum physics* (WEB, f):

```
Rx(theta) | qubit
```

The main objects in ProjectQ are engines. An engine instance is also the program, on which we allocate a certain number of qubits. Then, the operators are directly applied to the qubits. The circuit drawer using Matplotlib is also added to the program as an engine.

```
drawing_engine = CircuitDrawerMatplotlib()
engines=[drawing_engine]+get_engine_list()
qc = MainEngine(engine_list=engines)
q = qc.allocate_qureg(8) # define qubits
for i in range(8):
    if randrange(2)==0:
```

```
        X | q[i] # apply X operator
All(Measure) | q # Measure the qubits
print(drawing_engine.draw())
```

An example circuit drawing is given in Figure 5.

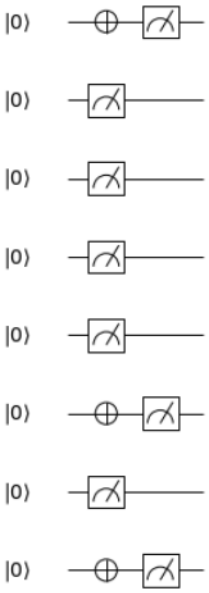

Fig. 5. An example quantum circuit in ProjectQ

Executing of a program on the simulator is different for ProjectQ. Instead of sending the program to the simulator with the number of shots, the program can be executed one by one. So, for multiple executions, we can use a loop.

```
for i in range (3):
    All(Measure) | q
    qc.flush()
    print("iteration",(i+1))
    for qubit in q:
        print(int(qubit))
```

An example outcome is as follows:

```
iteration 1
1
0
0
0
```

```
0
0
1
0
iteration 2
1
0
0
0
0
0
1
0
iteration 3
1
0
0
0
0
0
1
0
```

The outcome of each qubit is accessed separately. Thus, post-processing of the outcomes may be needed.

3.2 Module for superdense coding

Superdense coding is one of the basic quantum protocols using quantum entanglement (see Figure 6). Here we use a very basic form of classical modular programming.

The task is to write a module (function) which returns a quantum circuit that implements the superdense coding protocol where the classical message is provided by the user. The message can be any 2-bit string: (00,01,10,11). The sub-tasks are

- to design the circuit,
- to draw the circuit, and,
- to read the classical message.

The implementations were successful in each QSDK without any issues to be pointed out.

3.3 Intermediate measurement of a single qubit

This is a very simple example using intermediate measurements. We find this task important as we can implement part (see Figure 7) of a fundamental quantum interference experiment.

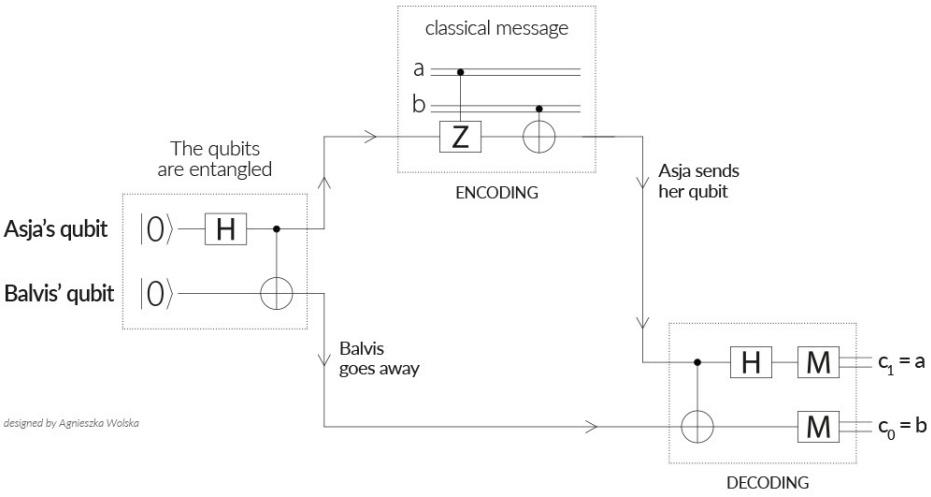

Fig. 6. The protocol for superdense coding

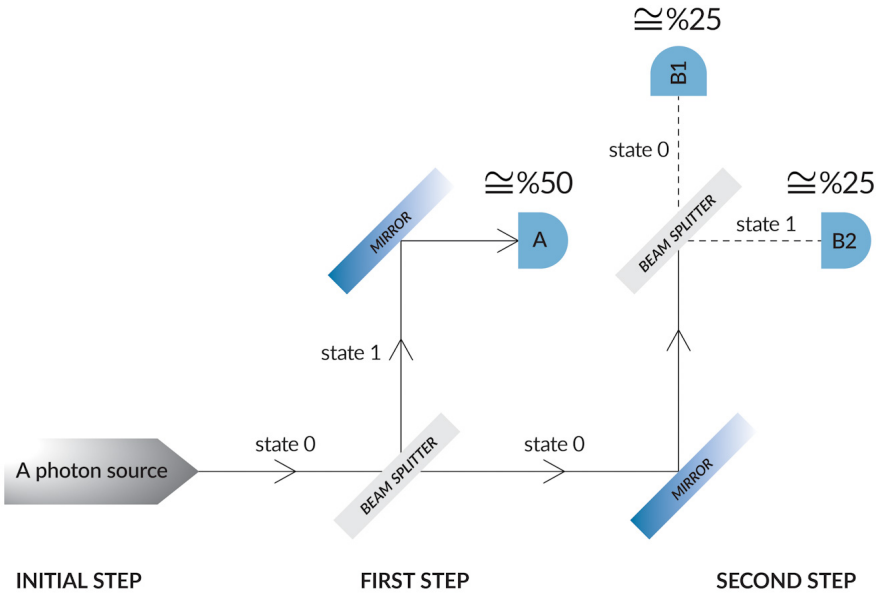

Fig. 7. A basic experiment with photons

The task is to design a quantum circuit with a single qubit. First, we apply a Hadamard operator and measure the qubit. If the outcome is 0, then we apply another Hadamard operator. At the end, we again measure the qubit. The sub-tasks are

- to design the circuit,
- to draw the circuit, and,
- to execute the quantum program 1000 times with checking the statistics.

Up to date, Cirq does not support classically controlled quantum operators where the classical values are obtained by the measurement. This task can be implemented for the other libraries. But, for visualization, only Qiskit and ProjectQ draw the circuit. The former one draws the circuit before executing (see Figure 8).

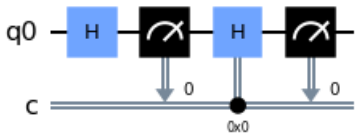

Fig. 8. The quantum circuit for Task 3 in Qiskit

The latter one draws the executed circuits, and so in each execution the number of Hadamard operator(s) can be different (see Figure 9).

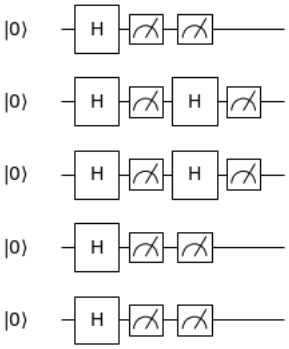

Fig. 9. The quantum circuits (executed 5 times) for Task 3 in ProjectQ

3.4 Three rotations with entanglement

The task is to design a quantum circuit with three qubits, say q1, q2, and q3. We rotate each qubit by angles $\pi/6$, $-\pi/4$, and $\pi/3$, respectively. Then, we apply the controlled-NOT operators CNOT(q1,q2) and CNOT(q2,q3). This time, instead of executing the circuit and collecting statistics, we are interested in reading the quantum state and unitary matrix of the circuit at any point of the design. Thus, we have two sets of sub-tasks. The first set of sub-tasks is

- to read the state vector,
- to obtain the probability of each basis state,
- to read the unitary matrix, and
- to measure all qubits.

The second set of the sub-tasks is

- to draw the circuit and
- to execute the quantum program 1000 times with checking the statistics.

The implementations were successful in each QSDK without any problem. Based on the outcomes, we can represent how the circuit maps the initial state as shown in Figure 10 (using Qiskit).

```
0.306     [[ 0.306 -0.177  0.306 -0.177 -0.53   0.306 -0.53   0.306]     1.0
-0.177     [-0.177 -0.306  0.177  0.306  0.306  0.53  -0.306 -0.53 ]     0.0
-0.53      [-0.53   0.306  0.53  -0.306 -0.306  0.177  0.306 -0.177]     0.0
0.306   =  [ 0.306  0.53   0.306  0.53   0.177  0.306  0.177  0.306]  ,  0.0
0.53       [ 0.53  -0.306  0.53  -0.306  0.306 -0.177  0.306 -0.177]     0.0
-0.306     [-0.306 -0.53   0.306  0.53  -0.177 -0.306  0.177  0.306]     0.0
-0.306     [-0.306  0.177  0.306 -0.177  0.53  -0.306 -0.53   0.306]     0.0
0.177      [ 0.177  0.306  0.177  0.306 -0.306 -0.53  -0.306 -0.53 ]]    0.0
```

Fig. 10. The calculation of a circuit represented by a matrix-vector multiplication

3.5 Quantum teleportation

This is one of the basic quantum protocols, and its implementation requires conditional post-processing.

The task is to design a quantum circuit implementing quantum teleportation including the post-measurement part. The sub-tasks are

- to draw the circuit,
- to read the measurement results, and
- to read the quantum state.

The implementation of quantum teleportation is already publicly available for each QSDK on the internet. So, this task is already implemented in each library. The only incomplete sub-task here is drawing the whole circuit by pyQuil.

The text version of the circuit is concise as shown in Figure 11 (using Cirq). Very recently, Cirq also introduces 3D representation of 2D circuits. We observe this feature with quantum teleportation protocol (see Figure 12).

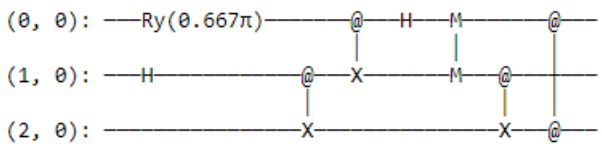

Fig. 11. The text-based circuit for quantum teleportation by Cirq

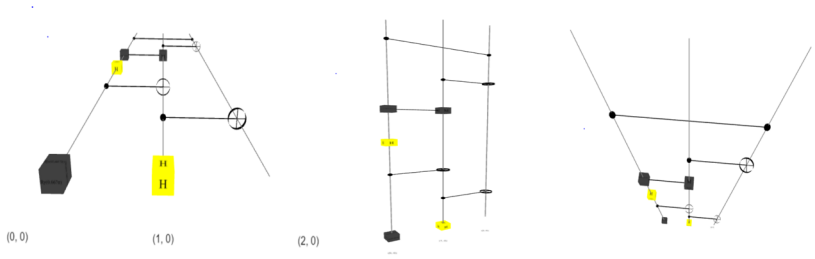

Fig. 12. The 3D circuit for quantum teleportation by Cirq

3.6 Multiple rotations

We have two sets of sub-tasks here. For given four angles $\theta_1, \ldots, \theta_4$, the first sub-task is to design a circuit implementing the unitary matrix

$$U = \left(\begin{array}{c|c|c|c} R_{\theta_1} & 0 & 0 & 0 \\ \hline 0 & R_{\theta_2} & 0 & 0 \\ \hline 0 & 0 & R_{\theta_3} & 0 \\ \hline 0 & 0 & 0 & R_{\theta_4} \end{array}\right),$$

where 0 is (2×2)-dimensional zero matrix and R_θ is the rotation matrix with angle θ in the $|0\rangle$-$|1\rangle$ plane, and then, to read the unitary matrix in order to verify the correctness of the circuit.

The second sub-task is about repetition of the first sub-task: We design two circuits applying U three and five times, respectively, and then, read the unitary matrices in order to verify the correctness of the circuits.

Such unitary matrices can be constructed in different ways, and so up to three different solutions are given for each library.

4 Our findings

We present our findings under three subsections.

4.1 Installation

To implement our tasks, we use jupyter notebooks and we install Anaconda as it comes with widely used python libraries (and so we do not need to install extra packages). Installing Anaconda may cause some problems from time to time[2], but we see it as a default step for each QSDK.

The quantum programming libraries are written in Python. So, their installations are easily done by using a single command on jupyter notebooks, e.g., we can use the following command for Qiskit:

```
!pip install qiskit[visualization]
```

Among all, Forest has extra difficulties for the new learners. It has separate tools for compiling and executing the quantum programs to be more efficient. But, their installations should be done separately, and each of these tools should be run separately in parallel to coding quantum programs in Python. We suggest that Forest may provide Python modules of these tools as well besides the external ones, at least for pedagogical purposes.

4.2 Visualization

Visualization of quantum circuit is pedagogically very helpful. The QSDKs have been improving in this sense, but there are still certain issues.

QSDKs offer three types of basic solutions when drawing quantum circuits:

1. outputting text such as using ASCII Art,
2. using python libraries such as Matplotlib, or
3. outputting LaTeX code and then visualizing the compiled LaTeX code.

Cirq uses text outputs, and it works fine. Qiskit supports all solutions, and colorful circuits can be obtained when using Matplotlib. ProjectQ uses a solution based on Matplotlib. Lastly, pyQuil uses a LaTeX solution.

Among all, Qiskit and ProjectQ work very well in the visualization tasks. When drawing the circuit with classically controlled quantum operators, we could not draw the circuits by the other two libraries. Except this, Cirq works very well. The library pyQuil also fails when drawing the whole circuit for quantum teleportation protocol.

[2] Private communication with the organizers, lecturers, and mentors of QBronze workshops. The list of these workshops are at `https://qworld.net/workshop-bronze/`.

In general, the LaTeX-based solutions may not work properly as they require non-python installations and execution of external tools. Besides, their representative power may be limited. Text outputs are simple and come with minimal dependencies. Matplotlib is a python library that can be used to get user-friendly circuits.

Very recently, Cirq introduced 3D visualization of 2D circuits by using Typescript. We test this feature by drawing the circuit for the quantum teleportation protocol. We believe that such functionality would be useful for 2D circuits, and certain improvements should be done to increase the readability of gates.

4.3 The main functionalities

For the rest of tasks, almost all were implemented in each QSDK. The only exception is that Cirq does not support storing intermediate measurements classically, which can be used later to apply certain quantum operators. But, we should note that we can still implement quantum teleportation in Cirq.

5 Concluding remarks

There are several QSDKs (WEB, h) and they are being improved day to day. In this work, we focus on Qiskit, Cirq, Forest (pyQuil), and ProjectQ, and our motivation is to determine how suitable they are to be used for introductory level of education. Our findings are summarized in Table 1.

Among all, Forest has certain issues to be solved regarding the installation and drawing the circuits. Qiskit seems the most convenient one, but Cirq and ProjectQ are also ready to work as we report only minor issues.

Our methodology can be a base for further studies. We use only six tasks, and, as future work, we plan to extend our task list, such as with operations on Bloch sphere and its visualization.

Besides task-based approach, as pointed out by one of the anonymous reviewers, the study can be extended with different criteria such as (i) the available documentation for each QSDK, (ii) executing programs on real machines, and (iii) how friendly the program syntax is. Our observations are as follows:

1. Each QSDK has its own documentation. Besides there are different online platforms where the users can get help.
2. QSDKs have certain APIs to access different hardware such as IBM's quantum computers, IonQ hardware, AWS Braket, AQT devices, etc.
3. We do not have any problem or difficulties to remark regarding the syntax of each library. But, different usability tests can be performed among different groups when comparing the usability of QSDKs.

As a final remark, each of QSDKs has some different features, which may be used to develop pedagogically interesting applications.

Functionalities	Qiskit	Cirq	Forest	ProjectQ
Randomly generating very simple quantum circuits (using classical conditionals and loops)	✓	✓	✓	✓
Executing quantum circuits and reading the measurement results	✓	✓	✓	✓
Visualizing the quantum circuits	✓ (1,2,3)	✓ (1,*)	✓ (3,*)	✓ (2)
Simplifying the design by using classical subroutines	✓	✓	✓	✓
Reading the quantum state	✓	✓	✓	✓
Reading the unitary operator of the circuit	✓	✓	✓	
Applying controlled quantum operators	✓	✓	✓	✓
Implementing intermediate measurements	✓	✓ (**)	✓	✓
Applying classically controlled quantum operators	✓	✓	✓	✓
Implementing quantum teleportation as a sophisticated protocol for the introductory level	✓	✓	✓ (*)	✓

(1) Using ASCII Art; (2) Using Matplotlib; (3) Using external LaTeX tools;
(*) Problem with drawing of circuits with classically controlled quantum operators;
(**) Problem with storing measurement outcomes classically and then using quantumly.

Table 1. The summary of our findings

6 Acknowledgements

Yakaryılmaz was partially supported by the ERDF project Nr. 1.1.1.5/19/A/005 "Quantum computers with constant memory" and the project "Quantum algorithms: from complexity theory to experiment" funded under ERDF programme 1.1.1.5.

We thank the anonymous reviewers of KKIO2021 and BJMC for their very helpful and detailed comments and suggestions. We thank Vishal Sharathchandra Bajpe for his remarks about ProjectQ. We thank Agnieszka Wolska for preparing Figures 6 and 7.

We used Qiskit, Cirq, ProjectQ, and Forest to create the circuits in the paper.

References

Abbas, A., Andersson, S., Asfaw, A., Corcoles, A., Bello, L., Ben-Haim, Y., Bozzo-Rey, M., Bravyi, S., Bronn, N., Capelluto, L., Vazquez, A.C., Ceroni, J., Chen, R., Frisch, A., Gambetta, J., Garion, S., Gil, L., De La Puente Gonzalez, S., Harkins, F., Imamichi, T., Jayasinha, P., Kang, H., Karamlou, A.h., Loredo, R., McKay, D., Maldonado, A., Macaluso, A., Mezzacapo, A., Minev, Z., Movassagh, R., Nannicini, G., Nation, P., Phan, A., Pistoia, M., Rattew, A., Schaefer, J., Shabani, J., Smolin, J., Stenger, J., Temme, K., Tod, M., Wanzambi, E., Wood, S., Wootton, J. (2020). Learn Quantum Computation Using Qiskit, available at: `http://community.qiskit.org/textbook`.

Endo, S., Benjamin, S. C., Li, Y. (2018). Practical quantum error mitigation for near-future applications software framework for quantum computing. Physical Review X 8, 031027.

Endo, S., Cai, Z., Benjamin, S. C., Yuan, X. (2020). Hybrid quantum-classical algorithms and quantum error mitigation. Tech. Rep. 2011.01382, arXiv.

Hassija, V., Chamola, V., Saxena, V., Chanana, V., Parashari, P., Mumtaz, S.,Guizaniz, M. (2020). Present landscape of quantum computing. IET Quantum Communication (2), 42–48.

Kaye, P., Laflamme, R., Mosca, M. (2006). An Introduction to Quantum Computing. Oxford University Press.

LaRose, R. (2019). Overview and comparison of gate level quantum software platforms. Quantum 3, 130.

Nielsen, M. A., Chuang, I. L. (2000). Quantum Computation and Quantum Information. Cambridge University Press.

Salehi, Ö., Dimitrijevs, M., Gábris, A. (2021). Silver, available at: `https://gitlab.com/qworld/silver`.

Salehi, Ö., Seskir, Z., Tepe, İ. (2021). A computer science-oriented approach to introduce quantum computing to a new audience. IEEE Transactions on Education pp. 1–8 , available at: `https://doi.org/10.1109/TE.2021.3078552`.

Steiger, D.S., Häner, T., Troyer, M. (2018). ProjectQ: an open source software framework for quantum computing. Quantum 2, 49.

Vietz, D., Barzen, J., Leymann, F., Wild, K. (2021). On decision support for quantum application developers: categorization, comparison, and analysis of existing technologies. ICCS 2021, Vol. 12747 of LNCS, 2021, pp. 127–141.

WEB (a). The Quantum Katas, available at: `https://github.com/microsoft/QuantumKatas`.

WEB (b). Qiskit, available at: `https://github.com/Qiskit`.

WEB (c). Cirq, available at: `https://github.com/quantumlib/Cirq`.

WEB (d). PyQuil, available at: `https://github.com/rigetti/pyquil`.

WEB (e). Huawei HiQ, available at: `https://github.com/Huawei-HiQ`.

WEB (f). ProjectQ, available at: `https://github.com/ProjectQ-Framework/ProjectQ`.

WEB (g). LaRose, R.: Practical Quantum Computing with Cirq, available at: `https://quantumcomputingreport.com/review-of-the-cirq-quantum-software-framework/`.

WEB (h). List of QC simulators, available at: `https://quantiki.org/wiki/list-qc-simulators`, September 2021.

Wootton, J.R., Harkins, F., Bronn N.T., Vazquez, A.C., Phan, A., Asfaw, A.T. (2020). Teaching quantum computing with an interactive textbook. Tech. Rep. 2012.09629, arXiv.

Yakaryılmaz, A., Bajpe. V. S., Šćekić, M., Salehi, Ö., Dimitrijevs, M. (2021). Bronze-ProjectQ, available at: `https://gitlab.com/qworld/bronze-projectq`.

Yakaryılmaz, A., Salehi, Ö., Dimitrijevs, M. (2021). Bronze-Qiskit, available at: `https://gitlab.com/qworld/bronze-qiskit`.

Received November 3, 2021 , revised February 28, 2022, accepted March 16, 2022