University of Waterloo

Faculty of Science

Designing a Quantum Programming Language for QuantumION

Author
Ruhi Shah

Program and Term
Mathematical Physics 2B

University of Waterloo

Faculty of Science

Designing a Quantum Programming Language for QuantumION

IQC, Waterloo, ON, Canada

Author
Ruhi Shah
20764515

Program and Term
Mathematical Physics 2B

November 23, 2021

Ruhi Shah, 20764515
116 Westover Crescent
Ottawa, ON, K2T0K6

Brian McNamara, Department Chair
Physics and Astronomy,
Faculty of Science,
University of Waterloo
200 University Ave West
Waterloo, ON, N2L 3G1

Dear Professor McNamara,

This report, titled "Designing a Quantum Programming Language for QuantumION", was prepared after my 2B term in Mathematical Physics, and was my first work term report for the IQC. The purpose of this report is to explain the process of creating a new circuit-level programming language for the QuantumION computer.

The Institute for Quantum Computing is located at the University of Waterloo, and has the main goal of developments in the field of quantum computing. My job at this organization was to create a Python based circuit-level programming language, personalized for the QuantumION computer. I was on the QuantumION controls team, and my supervisor was Virginia Frey and Crystal Senko.

I would like to thank my supervisor, Virginia Frey, for providing inspiration, encouragement, and direction to write this report, and always providing incredibly useful feedback on all of my work. I would also like to thank the QuantumION team for their support. I have read over and formatted this report. This report was written entirely by me and has not received any previous academic credit at this or any other institution.

Sincerely,
Ruhi Shah
20764515

Contents

List of Figures

List of Tables

1 Summary

A new Python based circuit-level module is created for the QuantumION computer. The existing controls system and User Language are described, and requirements for a circuit-level language for QuantumION are expressed. The syntax and functionality of existing Python based circuit-level languages (Qiskit, Cirq, pyQuil, ProjectQ, and braket) are surveyed in relation to the requirements for QuantumION. The structure and syntax of the new Python based circuit-level language for QuantumION, called qion, is described. Finally, circuits for the teleportation protocol, the five-qubit code, and fault-tolerant state preparation are coded in qion as examples.

2 Introduction

Anybody who keeps up with popular science will know that quantum computing has been at the forefront of research for the past decade. There has been an enormous amount of money and effort being poured into the field. From large corporations to start-ups, everybody is trying to hit the jackpot of building the most successful quantum computer [Gil, 2020]. Quantum computing has been used for a variety of applications from algorithms for integer-factorization, to drug formation [Gossett, 2020].

There are a couple of things needed to build a quantum computer. First, a viable physical system that satisfies DiVincenzo's Criteria [DiVincenzo, 2000]. The second, which will be the focus of this paper, a way to control the physical system in a meaningful way.

For any quantum computer, the task of turning a user's instructions into instructions that can be applied to the physical system requires a lengthy and often complicated translation process that the controls system is responsible for.

The particular quantum computer that is the subject of this paper is QuantumION, the ion-trap based quantum computer at the Institute for Quantum Computing in Waterloo, Canada. The objective is to create a suitable high-level user control for QuantumION. In

this paper, first, QuantumION is described, with emphasis on the existing controls system. Then, a need for a circuit-level user language is expressed. For the rest of the paper, an appropriate circuit-level language for QuantumION is designed.

3 Controls System for QuantumION

3.1 Quantum Computing with Ion Traps

Ion traps satisfy the DiVincenzo's criteria in the following way. This description will be brief, sufficient for the purposes of this paper.

In an ion trap, the qubits are represented by the energy levels of the ions. An ion with its electron in the ground state, is represented as $|0\rangle$, and the excited state is labelled with $|1\rangle$. Qudits are defined as quantum systems with more than two-levels. Hence, it is easy to see how further excited states can be harnessed to create qudits, which would be ions with more than two states. These ions are initialized into a qudit state using cooling and optical pumping techniques. In terms of operations, single qudit gates are then performed using Rabi oscillations using a laser. Two qudit gates can be done using the motional states of the ions. Measurement is done by detecting fluorescence from the ions. Finally, the decoherence times are sufficiently long enough and the gate execution times are sufficiently short enough for the computer to perform meaningful computations [Häffner et al., 2008].

Hence, connected to this ion trap, a suitable system of controls is needed to allow a user to specify instructions to the ion trap to perform actions.

3.2 Controls System Stack

In the literature, the most popular quantum algorithms are encoded as quantum circuits. A quantum circuit provides a universal way to describe quantum algorithms. Generally, a quantum circuit consists of a register of qubits, a set of gates that act on them, a set of measurements that happen to them, and potential conditional gates that act on them given

the results of these measurements [Nielsen and Chuang, 2011].

It is then evident to see that performing a quantum circuit on an ion trap will require extensive translation between a circuit description of the algorithm, to instructions that can be given to the various hardware components. This includes equipment such as the lasers, detectors, and memory.

In order to perform quantum computations on QuantumION, the user needs to manipulate lasers, detectors, and memory. Most commercially available quantum computers out there do not expose the user to these low-level controls, [Heim et al., 2020]. The user is able to input a circuit description of the algorithm they wish to run, and the translation down to the instructions to the hardware is done behind the scenes. What is special about QuantumION, is the level in the chain of translation that the user language is able to act on (Figure 1). Users are able to directly input low-level controls, through what is known as the QuantumION User Language [Rademacher, Richard, 2020].

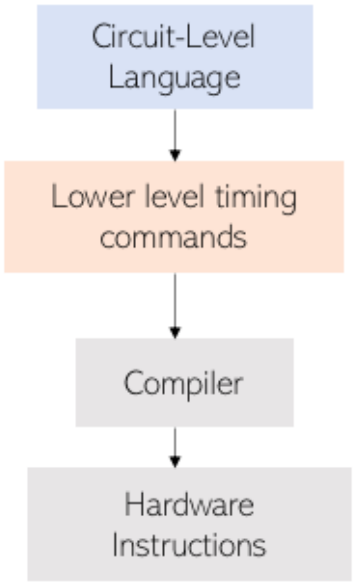

Figure 1: A flowchart representing a potential controls stack. The box shown in blue is where most available quantum computing languages interact with the user. The box shown in orange is where QuantumION allows users to interact from.

3.3 The User Language

The User Language of QuantumION is an XML based language, along with a "binding language" written Python that allows the user to give incredibly specific instructions to the hardware components directly [Frey and Rademacher, 2020b]. The "binding language" refers to the elements of the XML language mapped one-to-one to a Python package, since Python is more popular and convenient for users than XML [Frey and Rademacher, 2020a]. The User Language consists of two layers: the timing layer and the gate layer. The timing layer allows the user to control the different hardware channels at specific times. The gate layer is built on top of the timing layer and allows the user to specify sequences of gates [Frey, 2021]. The timing layer needs to be quite verbose and expressive in order for the user to specify the individual instructions to each channel. The gate layer also follows similar syntax and semantics, and hence ends up being much more verbose than necessary. See Figure 2 for an idea of the verbosity of the gate layer. The top shows a CNOT gate in Qiskit [Abraham et al., 2019], while the bottom shows a CNOT gate in the User Language's gate layer. The syntax is much more verbose in the User Language. As mentioned earlier, most quantum algorithms are specified with circuits. Hence, there was a need to create a layer on top of the gate layer of the User Language, a circuit-level language, for users that wanted to execute simple circuits.

```
# Qiskit
c = QuantumCircuit(2)
c.cx(0, 1)
```

```
# QuantumION User Language
qi.GateCall("TwoQubitCNOTGate",
    assignments=[
        qi.Assign(port="Control", ion="ion1"),
        qi.Assign(port="Target", ion="ion2")])
```

Figure 2: A code snippet comparing a CNOT gate in Qiskit and in the User Language. The left is the gate in Qiskit, and the right is the gate in the User Language

4 Circuit-Level Languages

With the rise of quantum computers, circuit level languages have also been gaining a lot of traction [Heim et al., 2020]. In this paper, we focus on the Python based circuit level modules available.

4.1 Requirements for a Circuit-Level Language for QuantumION

There are already a variety of circuit-level languages out there, so the question begs to ask: does QuantumION need one of its own? First, the requirements for a circuit-level language for QuantumION are defined.

Starting with the basic building blocks, the ions in QuantumION can be two dimension qubits, or multi-dimensional qudits. Hence, the circuit level language needs to allow the user to manipulate and specify qudits. In order to organize qudits in a convenient and meaningful way, we need registers. These are essential when working with large quantum circuits, as they help the user keep track of the qudits. Moving up, we also require a convenient way for users to specify operations. These include all standard gates that may be used in a circuit, along with measurement. Next, we require a way for users to specify when they wish these operations to occur. We want users to have the ability to not only specify on which qudit, but also when in time the operation should occur. Finally, we want QuantumION users to have the ability to make error-correction circuits, and hence we want conditional logic to be supported. In summary, we have the following minimum requirements for a circuit-level language for QuantumION:

- Support qudits (more than just two-dimensional qubits).
- Have registers (for organization of qudits).
- Have all standard operations, including mid-circuit measurement.
- The ability to specify when (in time) an operation occurs.

- Ability to execute conditional logic as used in error-correction.

Finally, it is important to note that the User Language, which exists in XML, currently has a "language binding" in Python [Frey and Rademacher, 2020a]. Hence, we would like the circuit-level language to also be Python based.

Now, the existing Python based circuit-level languages can be explored to see if any of them meets the requirements in a portable manner. In the case that a new circuit-level language is created for QuantumION, the existing languages also should be surveyed to ensure that the circuit-level language created for QuantumION is able to encompass all the features of the existing languages.

4.2 Surveying Existing Circuit-Level Languages

In this section, five different Python based circuit-level languages are explored, the strengths and weaknesses of each one evaluated. The five that are explore are: Qiskit [Abraham et al., 2019], Cirq [Developers, 2021], pyQuil [Smith et al., 2016], ProjectQ [Steiger et al., 2018] [Häner et al., 2018], and braket [Amazon, 2021].

Since the goal of this analysis is to explore the syntax and functionality of the language, the `Circuit` object in each language is built from the bottom up to compare the different languages. First, the language elements which include: qubits and registers, standard gates, measurement, conditional operations, and the organization of the gates in time are all described. Then, drawbacks are discussed and the requirements for a circuit-level language for QuantumION are referred to in the discussion.

4.2.1 Qiskit

When creating a circuit in Qiskit, registers are specified first. Each `Qubit` object must be attached to a `Register` object. Every register can potentially have a name. It is convenient to have a name feature when keeping track of a circuit with several qudits.

```
# A length 5 QuantumRegister with name "Ancilla"
qr = QuantumRegister(5, "Ancilla")

# The last qubit in qr
q = Qubit(qr, 4)
```

Each `Circuit` object can then be initialized with one or more `Register` objects. `Register` objects can be both `ClassicalRegister` or `QuantumRegister` objects. It is also possible to initialize a `Circuit` object with two integers, the first representing the number of qubits, and the second representing the number of classical bits.

```
# A length 5 QuantumRegister
qr = QuantumRegister(5)

# A length 3 ClassicalRegister
cr = ClassicalRegister(3)

# A circuit with the two registers
circuit = QuantumCircuit(qr, cr)

# The same circuit, initialized with integers
circuit = QuantumCircuit(5, 3)
```

Once a `Circuit` has been initialized with `Register` objects, gates can be added. These gates are added through methods to the `Circuit` object, and specify the `Qubit` to act on through integer index of the `Register` objects.

```
# Initializing a circuit
c = QuantumCircuit(5, 3)

# Adding an X gate on qubit 1
c.x(1)

# Adding a CNOT gate on qubits 0, 4
c.cx(0, 4)
```

Measurement is specified through the `measure` gate. The first argument is the index of

the qubit in the quantum register that needs to be measured. The second argument of the `measure` gate is the index of the bit in the classical register that the measurement result should be stored in.

```
# Initializing a circuit
c = QuantumCircuit(5, 3)

# Adding a measurement on qubit 1, and storing the result in bit 0
c.measure(1, 0)
```

Conditional gates are added one at a time through the `c_if` method, that is applied after a gate is added. The first argument is a classical register, and the second argument is the value that the register needs to be in order for the gate to be executed.

```
# Initializing a circuit
c = QuantumCircuit(5, 3)

# Adding measurements
c.measure(1, 0)
c.measure(2, 2)

# Adding an X gate on qubit 3 if the 0th bit is 1
c.x(3).c_if(0, 1)
# Adding a H gate on qubit 3 if the 2nd bit is 1
c.h(3).c_if(2, 1)
```

Qiskit will automatically "slide" gates that are being added to the circuit horizontally. A gate added to a specific wire will take the earliest possible position in horizontal time for execution. It is only possible to prevent this sliding through `barrier` objects.

```
# Initializing a circuit
c = QuantumCircuit(5, 3)

# Adding an X gate on qubit 1
c.x(1)

# Adding a CNOT gate on qubits 0, 4
```

```
c.cx(0, 4)

# A Y gate on qubit 3 will happen at the very beginning
# If we require it to happen after the CNOT, we insert a barrier
c.barrier()
c.y(3)
# This moves the location of the Y gate to after the CNOT gate
```

The documentation can be referred to for further details [Asfaw et al., 2020].

The main drawback to Qiskit is the lack of ability to specify qudits. QuantumION would definitely like the user to specify qudits, and hence Qiskit is not a viable language to use for QuantumION. There are also a few other drawbacks as well. Qiskit requires circuits to be of a fixed size, this can be inconvenient when a user does not know the number of qubits in the circuit to begin with. It can also be harder to add a circuit to another, without creating a new circuit object. The conditional logic also allows the user to only add one gate at a time based on a condition, while it is known that several algorithms (particularly error-correction ones) require an entire sequence of gates, with potential future conditions, to be executed given a condition is true. An issue that many users of Qiskit face is the automatic horizontal compressing of the circuit object. The gates are added sequentially, on each wire, and it is only possible to specify that gates happen at the same time through `barrier` objects. This often gets confusing, as gates end up not where the user thought they should go. Finally, we hope for the high-level QuantumION circuit language to support ion assignment. This means allowing the user to specify which ion in the chain refers to which qubit object. This would not be possible with Qiskit.

Although Qiskit is incredibly popular, and has wonderful documentation to support the user, it is not a viable option for QuantumION mainly due to its lack of qudit support.

4.2.2 Cirq

Cirq, on the other hand, does support qudits. The `NamedQudit` object in Cirq contains a dimension, and an optional name attribute. `Register` objects are then collections of `Qudit`

objects. The user has the option of specifying them in a line or in a grid.

```
# A Qutrit named Alice
q = NamedQudit(name="Alice", dimension=3)

# A Register of 3 4-d qudits
qr = LineQid(3, dimension=4)
```

The `Circuit` object in Cirq, can then be initialized as an empty object. Gates are added through the `append` method. Gates can also directly be added to the `Circuit` when initializing, as arguments.

```
# An empty circuit
c = Circuit()

# Adding an X gate to q defined earlier
c.append(X(q))

# Adding CNOT gate to 2nd and 3rd qubits in qr defined earlier
c.append(CX(qr[2], qr[3]))
```

Measurements are added through the `measure` gate. Mid-circuit measurements are allowed, however, the results are not stored for later conditional use. There is no conditional logic possible.

```
# An empty circuit
c = Circuit()

# Adding a measurement to q defined earlier
c.append(measure(q))
```

Cirq organizes its `Circuit` through the `Moment` object. This is how the time a gate is executed is specified. If the user wishes to specify that collections of gates happen at the same time, then they can all be added to a `Moment`, and the `Moment` added to the `Circuit`.

```
# Defining a Moment
# All of these operations will happen at the same time
m = Moment([X(q), CX(qr[0], qr[1])])

# Adding the Moment m to a circuit
c = Circuit(m)
```

The documentation can be referred to for further details [Google, 2021].

For Cirq, the main drawback is the lack of conditional logic. Other than this, the existence of qudits is ideal. The circuits and registers are all flexible objects, and the existence of `Moment` objects allows the user to control the circuit. However, the lack of conditional logic is the reason that Cirq cannot be used as a circuit-level language for QuantumION.

4.2.3 pyQuil

Circuits in pyQuil are known as `Program` objects, and are initialized to be empty. Gates are added through `+=` syntax, or through the `.inst` method. In pyQuil, qubits are not objects at all, and registers do not exist. Qubits are referred to in circuits through their integer index, which is parsed as an argument to the gate that is being applied to the qubit. Classical bits are specified to a circuit through the `.declare` method.

```
# Creating an empty circuit
c = Program()

# Adding classical bits to the circuit, and storing them in a variable
cr = c.declare('cr', memory_type='BIT', memory_size=3)

# Adding an X gate to qubit 0
c+=H(qubit=0)

# Adding an CNOT gate to qubit 1, 2
c.inst(CNOT(1, 2))
```

Measurement is specified through the `MEASURE` object. The first argument is the index of the qubit to be measured, and the second is the place in the memory to store the result.

```
# Adding a measurement to the circuit c above
# Measuring the 1st qubit and storing the result in the 0th bit defined earlier
c.inst(MEASURE(1, cr[0]))
```

Conditional operations are added to a circuit through the `.if_then` method. The first argument is the classical bit, and the second is the `Program` object to execute. If the value of the single classical bit is 1, the circuit is executed, otherwise, it is not.

```
# Applying an X gate on qubit 2 if cr[0] is 1, and a Y gate otherwise
c.if_then(cr[0], Program(X(1)), Program(Y(1)))
```

Lastly, just like Cirq, pyQuil has `Moment` objects that allow the user to specify the time at which the gates are being executed.

The documentation can be referred to for further details [Rigetti, 2019].

The lack of support for qudits removes pyQuil as a possibility of being used for QuantumION. There are also a couple more drawbacks. First, the lack of registers forces the user to keep track of all the different indices of qubits, and this can get confusing with large circuits. There are also limitations in the implementation of conditional operations. For example, a user may want to execute a sequence of gates given the value of various bits is 101. However, because of the boolean nature of the `.if_then` method, this would simply not be possible. Conditional logic in pyQuil can only be executed on a single bit.

4.2.4 ProjectQ

In ProjectQ, a circuit is called an `Engine` object. The circuit is initialized with a purpose, for example, `CircuitDrawer`. The addition of qubits is done through a method `allocate_qubit`. Each `Qubit` object has an `Engine` and a unique `ID`. They are referred to through the name of the qubit variable.

```
# Creating an empty engine object (circuit), with purpose of CircutDrawer
c = MainEngine(CircuitDrawer())

# Allocating a qubit to c, and storing it in variable q
q = c.allocate_qubit()
```

Since each qubit is attached to an engine, gates are directly attached to qubits. Measurement is a gate that can be applied to a qubit variable. The result of the measurement is then stored in the qubit variable itself. Once the qubit is measured, it is essentially "collapsed". This means further computations cannot be done with the qubit after measurement.

```
# Adding an X gate on qubit q defined earlier
X | q

# Measuring the qubit q defined earlier
MEASURE | q
```

Conditional operations are done using `with` blocks. The first argument for the `with` block is the circuit, or the `Engine` object, and the second is the qubit to control the block with. Just like pyQuil, if the qubit has value 1 after measurement, then the block is executed. This is better than Qiskit or pyQuil, since the user is able to execute an entire block of gates upon a condition being true, not just a single gate.

```
# Allocating new qubits, q0 and q1 to c
q0, q1 = c.allocate_qubit(), c.allocate_qubit()

# Conditional block of operations on q0, if the result of q is 1
with Control(c, q):
    H | q0
    CX | (q0, q1)
```

Finally, ProjectQ has `Moment` objects like in Cirq and pyQuil that allow the user to specify the time at which the gates are being executed.

The documentation can be referred to for further details [ProjectQ, 2017].

ProjectQ is not a feasible circuit-level language for QuantumION, mainly because it does not support qudits. Another drawback is the conditional operations restrictions that existed in pyQuil, since the block of gates can only be executed based on the value of a single qubit, blocks of gates cannot be conditioned on multiple qubits. A different drawback is the inability to use a qubit in the circuit after it has been measured.

4.2.5 braket

In braket, there are no qudits. Furthermore, the qubit objects in braket also do not exist, like in pyQuil, and just exist as integer arguments to gates. `Gate` objects are added like in Qiskit, through methods acting on the `Circuit` object. There is no mid-circuit measurement in braket, and hence there are no classical registers, or conditional operations either.

```
# Initializing an empty circuit
c = Circuit()

# Adding H to qubit 0, and X to qubit 1
c.h(0).x(1)

# Adding CNOT to qubit 1 and 2
c.cnot(1, 2)
```

Finally, braket has `Moment` objects like in Cirq, pyQuil, and ProjectQ that allow the user to specify the time at which the gates are being executed. These `Moment` objects are also visualized in the circuit using vertical lines.

The documentation can be referred to for further details [Amazon, 2021].

Hence, braket does not support qudits, and has limitations in its conditional operations, removing it as a possibility for a circuit-level language for QuantumION.

4.3 Summary of Surveying Languages

From the previous section, it is evident that none of the languages specified are suitable to use as a circuit-level language for QuantumION. Recall the requirements for a circuit-level language for QuantumION:

- Support qudits (more than just two-dimensional qubits).
- Have registers (for organization of qudits).
- Have all standard operations, including mid-circuit measurement.
- The ability to specify when (in time) an operation occurs.
- Ability to execute conditional logic as used in error-correction.

To summarize, we go through each of these requirements, and see which languages support them, and which ones don't. The results of surveying the languages can be summarized in the table below.

Requirement	Qiskit	Cirq	pyQuil	ProjectQ	braket
Support for qudits	No	Yes	No	No	No
Ability to organize qubits in registers	Yes	Yes	No	Yes	No
Mid-circuit measurement	Yes	Yes	Yes	Yes	No
Existence of "vertical" slices of a circuit	No	Yes	Yes	Yes	Yes
Execute conditional logic	Yes	No	Limited	Limited	No

Table 1: A table summarizing whether several circuit-level languages fulfill the requirements of a circuit-level language for QuantumION.

Hence, in order to fulfill all the requirements for an optimal circuit-level language for QuantumION, a new language needs to be created.

5 A Circuit-Level Language for QuantumION: qion

Considering the drawbacks and benefits of each of the languages in the previous section, as well as the specific requirements for QuantumION, a new Python based circuit-level

language, called "qion" was developed [Frey et al., 2021].

5.1 Language Elements

In this section, a summary of all the different language elements in qion is given. Overall, qudits and cdits (classical "dits") are organized in registers. Operations take qudits and cdits as input, and are organized in cycles. Cycles represent a vertical slice through time; all operations contained within a cycle happen at the same time, where "time" here is considered to be on the horizontal axis of a circuit. These cycles are organized time-wise into segments, with cycles that are added first happening first. Segments also contain potential conditions that instruct the flow of the circuit, and which segment to go next. A collection of segments then forms a circuit. Figure 3 can be referred to as a visual aid to understand how the elements work together.

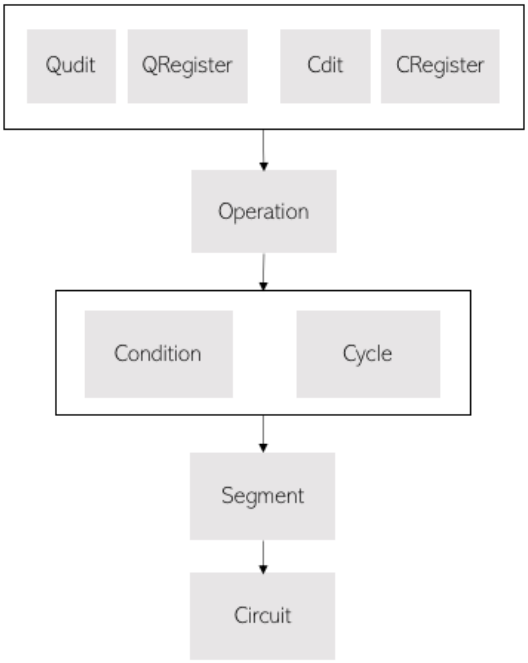

Figure 3: A flowchart representing the structure of qion.

5.1.1 Dits and Registers

In qion, a `Dit` object refers to an object that has a string name, and an integer dimension. A `Qudit` object is inherited from a `Dit`, and contains a string name, and an integer dimension. This object is used to represent qudits in a circuit. A `Cdit` object also inherits from a `Dit`, and contains a string name, an integer dimension. If the `Cdit` has been used to measure a `Qudit`, the `Cdit` object's result attribute is updated from `None` to the measured `Qudit`. There are also `Qubit` and `Cbit` objects, with fixed dimension attribute 2 for convenience.

```
from qion import Qudit, Cbit

a = Qubit(name="Alice")
b = Qudit(dimension=3)

ca = Cbit(name="Alice")
cb = Cdit(name="Bob", dimension=3)
```

A `QRegister` object then inherits from `list`, and contains non-repeating `Qudit` objects. Similarly, a `CRegister` object also inherits from `list`, and contains non-repeating `Cdit` objects. `Register` objects may also have a name.

```
from qion import QRegister, CRegister

# Quantum register of length 3
r = QRegister(length=3)

# Qudits
a = Qudit(name = "Alice")
b = Qudit(dimension = 3)

# Adding a and b to the register
r.append(a)
r.extend([b])

# Similarly for classical registers
cr = CRegister(2)

ca = Cbit(name = "Alice")
```

```
cb = Cdit(dimension = 3)

cr.append(ca)
cr.extend([cb])
```

These list-type registers offer all the different types of methods available to manipulate lists.

5.1.2 Operations

An `Operation` object in qion is one that contains a single or a list of `Dit` objects. A `Gate` then inherits from `Operation` and contains `qudits`, `gate_symbol`, and `matrix_representation`. All the different types of named gates such as `X`, `CX`, and so on, inherit from `Gate`. Each `Operation` object can have some `Dit` objects attached to it. For example, an `X` gate on a `Qudit`, will update the `X` gate's qudit attribute to that `Qudit` object.

```
from qion import X, CX, Measure, QRegister, CRegister

r = QRegister(2)
cr = CRegister(1)

# A single qubit gate
x = X(r[0])

# A two qubit gate
cx = CX(r[0], r[1])

# A Measurement
m = Measure(r[0], cr[0])
```

5.1.3 Cycles

A `Cycle` object contains a list of `Operations` that are supposed to be executed at the same time. Hence, `Operations` on the same `Qudit` object cannot overlap in a `Cycle`, since a qudit can only have one thing done to it at a time. Each `Cycle` also has a `QRegister` and

`CRegister` which contains all the qudits and cdits that are being acted upon in the `Cycle`, respectively.

```
from qion import X, CX, Measure, QRegister, CRegister, Cycle

r = QRegister(3)
cr = CRegister(1)

# A single qubit gate
x = X(r[0])

# A two qudit gate
cx = CX(r[1], r[2])

# A Measurement
m = Measure(r[0], cr[0])

cy = Cycle(x, cx)
```

5.1.4 Conditions

A `Condition` object contains three things. First, a tuple of `cdits` to check. Second, a `pattern` that the `cdits` should match to. Finally, a destination `Segment` object to go to if the `cdits` match the `pattern`. `Segment` objects will be explained further in detail next, but for now, they are essentially a list of time-ordered `Cycle` objects.

```
from qion import Measure, X, QRegister, CRegister, Condition, Cycle
from qion import ConditionalCircuitSegment

r = QRegister(3)
cr = CRegister(1)

# A Measurement
m = Measure(r[0], cr[0])

# The desitination segment (an X gate on first qubit)
destination = ConditionalCircuitSegment(Cycle(X(r[1])))
cond = Condition(cdit=(cr[0],), pattern="1", destination=destination)
```

5.1.5 Segments

A `Segment` object, referring to the `ConditionalCircuitSegment` object, contains a time ordered list of `Cycle` objects and a list of `Condition` objects.

When a `Segment` object is executed, first all the `Cycle` objects are run. Then, the `Condition` objects are evaluated in order to decide which `Segment` to execute next. Recall that each `Condition` object has a destination `Segment`. If the `cdits` in the `Condition` match the `pattern` specified, the corresponding destination `Segment` will be executed. Otherwise, the next `Condition` will be checked. Finally, if none of the cdits of the conditions match their respective patterns, the circuit terminates, since there is no destination segment to go to next.

```
import qion as q

qr = QRegister(1)
cr = CRegister(1)

# A Measurement
m = Measure(qr[0], cr[0])

# The destination segment (an X gate on first qubit)
destination1 = ConditionalCircuitSegment(X(qr[0])
destination2 = ConditionalCircuitSegment(Z(qr[0]))
cond1 = Condition(cdit=(cr[0],), pattern="1", destination=destination1)
cond2 = Condition(cdit=(cr[0],), pattern="0", destination=destination2)

# First the Y gate is executed on qubit 0, then the cond1 is checked, followed by cond2
seg = ConditionalCircuitSegment(Y(qr[0]), conditions=(cond1, cond2))
```

5.1.6 Circuits

It is clear to see that a `Circuit` object is then just a list of `Segment` objects, with the first `Segment` being executed automatically, and the remaining ones executed based on `Condition` objects in other `Segments`.

```
from qion import Cycle, ConditionalCircuitSegment, Circuit

cyc = Cycle()
seg = ConditionalCircuitSegment(cyc)
circuit = Circuit(seg)
```

The documentation can be referred to for further details [Shah and Frey, 2021].

5.2 Translation to the User Language

Although a circuit-level language has been built, it is essential to connect it to the rest of the controls system stack. In Figure 1, this would refer to the arrow connecting the orange box to the blue. A translation needs to be made between the circuit-level language in Python, qion, and the User Language from Section 3.3. Each object in a `Circuit` in qion must be translated to an `Experiment` object in the User Language. Since this would require briefing the reader on the structure of the User Language, which drifts away from the purpose of this paper, the details of this translation are omitted. Further information regarding this can be found on the qion documentation [Shah, 2021]. It is just important to note that a translation down the stack needs to be considered when designing a circuit-level language.

5.3 Example Circuits in qion

In this section, a few popular circuits are implemented in qion in order to further exemplify the language. For the purposes of this paper, the background behind the circuits and algorithms themselves are not described in detail, but merely implemented. However, references are given for those interested.

5.3.1 Teleportation Circuit

The teleportation circuit, or teleportation protocol [Bell, 1964], is an incredibly popular circuit that allows the state of an arbitrary qubit to be teleported from one wire to another

using a Bell pair. We start with three qubits, the first is the qubit we wish to send, and the second and third form a bell pair.

First, three qubits are created and a classical register of length 2 is created.

```
# Initialize our qubits and cbits
a = Qubit("Alice")
b = Qubit("Bob")
m = Qubit("Message")

cr = CRegister(2) # CR of length 2
```

Next, an empty circuit is initialized. Then, the `Message` qubit must encode some state. Let us use the RZ gate to initialize this state in some angle `psi` chosen earlier.

```
# Then an empty circuit is initialized
teleportation = Circuit()

# Add an RZ gate to qubit m
psi = 2.3
teleportation.add(RZ(m, rotation_angle=psi))
```

Next, the Alice and Bob qubits are entangled. This is done using a Hadamard gate on qubit Alice followed by a CNOT gate with control Alice and target Bob.

```
# Add an H gate to qubit a, and a CX to a and b
teleportation.add(H(a), CX(a, b))
```

Now a measurement is done on the Message and Alice qubits in the Bell basis. This measurement is stored in a classical register `cr`, of length 2. In order to do the basis change to the Hadamard basis, a CNOT gate is applied with the Message as control and Alice as target, followed by a Hadamard gate on the Message. Then both qubits are measured, and the results stored in the classical register.

```
# Add a CX gate to qubit m and a, and an H to m
teleportation.add(CX(m, a), H(m))
teleportation.add(Measure((m, a), cr))
```

Finally, depending on the measurement results in the classical register, an X gate and a Z gate are applied conditionally on the Bob qubit. These conditions are attached to the 0th segment of the circuit, `circ[0]`.

```
# Conditional Operations
# If cr[0] is 1, apply the X gate to qubit b
teleportation.add_if(q.Condition((cr[0], cr[1]), "1X"), q.X(b), teleportation[0])
# If cr[1] is 1, apply the Z gate to qubit b
teleportation.add_if(q.Condition((cr[0], cr[1]), "X1"), q.Z(b), teleportation[0])
```

We can visualize the circuit with `draw_circuit` as seen in Figure 4.

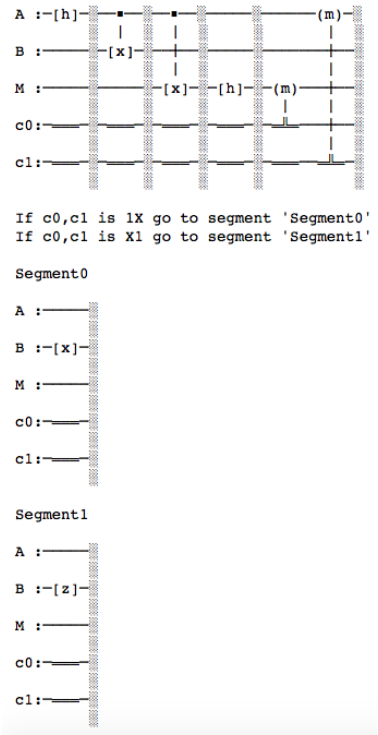

Figure 4: A circuit diagram of the teleportaiton circuit generated using qion.

5.3.2 Five Qubit Code

The five qubit code is the simplest possible fault-tolerant error-correction code [Chao and Reichardt, 2018]. It consists of 5 code qubits, 1 syndrome ancilla qubit, and 1 flag ancilla qubit. It also consists of two subroutines, the first is the fault tolerant syndrome measurement circuit (`ftsmc`), and the second is the non-fault tolerant syndrome measurement circuit, or just the syndrome measurement circuit (`smc`). The `ftsmc` circuit can be run up to 4 times, each time with different gates. Hence, the `ftsmc` circuit is indexed from 0-3. At the end of each `ftsmc` iteration, the flag ancilla is measured. If the flag ancilla is 1, we do not run the remaining iterations of the `ftsmc`, and instead run the `smc` circuit. Either all four `ftsmc` iterations are run, or some `ftsmc` iterations are run, followed by a single `smc` is run. After these two scenarios, a set of correction gates are run that depend on the measurement result of measuring the syndrome ancilla. Figure 5 is a diagrammatic representation of the protocol. Each blue box in the figure represents a `ConditionalCircuitSegment` object.

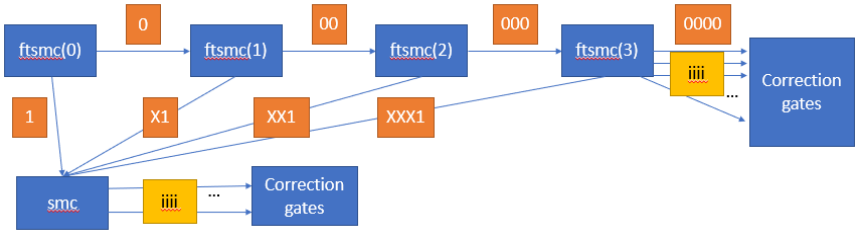

Figure 5: A flowchart representing the five qubit code. The blue boxes are segments, the orange are flag measurements, and the yellow are syndrome measurements.

To code this protocol in qion, we first create our registers.

```
import qion as q

# This is where the classical measurement is stored
# The syndrome and flag measurements are stored in this
c = q.CRegister(8, prefix="c")
# The syndrome measurements are stored here if we get a "1" on the flag
ce = q.CRegister(4, prefix="ce")

# This is where the code qubits are stored
```

```
qc = q.QRegister(5, prefix="qc")

# This is where the ancilla qubits are stored
qa = q.QRegister(2, prefix="qa")
```

Then, we define the `ftsmc(i)` function that returns the i^{th} iteration of the `ftsmc` circuit as a `ConditionalCircuitSegment`.

```
# We run this circuit 4 times,
# each time with slightly different positions,
# provided that the measurement after each iteration is 0

def ftsmc(i): # i here is the positions of our loop iteration, goes from 0 to 3
    """This function will return the circuit required at the
    ith position of our loop."""

    # Define positions based on i
    pos = [i % 5 for i in range(i, i + 4)]

    # Initialize an empty segment
    fteci = q.ConditionalCircuitSegment()

    # To change basis back after previous measurement
    fteci.add(q.H(qa[1]))

    # Add gates that copy the error
    fteci.add(q.H(qc[pos[0]]))
    fteci.add(q.CX(qc[pos[0]], qa[0]))
    fteci.add([q.H(qc[pos[0]]), q.CX(qa[1], qa[0])])
    fteci.add(q.CX(qc[pos[1]], qa[0]))
    fteci.add(q.CX(qc[pos[2]], qa[0]))
    fteci.add([q.CX(qa[1], qa[0]), q.H(qc[pos[3]])])
    fteci.add(q.CX(qc[pos[3]], qa[0]))
    fteci.add(q.H(qc[pos[3]]))
    # To change basis before measurement
    fteci.add(q.H(qa[1]))

    # Measure the ancillas and store in a register based on i
    fteci.add([q.Measure(qa[0], c[2*i]), q.Measure(qa[1], c[2*i+1])])

    # Return the segment
    return fteci
```

Then we define a function that returns the `smc` segment.

```
# If the measurement after a given iteration is not 0,
# we run the smc circuit below

def smc():
    """ This function will return the circuit required at the ith position
    of our loop."""

    smc = q.ConditionalCircuitSegment()
    for j in range(0, 4):

        pos = [j % 5 for j in range(j, j + 4)]

        # Add gates that copy the error
        smc.add(q.H(qc[pos[0]]))
        smc.add(q.CX(qc[pos[0]], qa[0]))
        smc.add(q.H(qc[pos[0]]))
        smc.add(q.CX(qc[pos[1]], qa[0]))
        smc.add(q.CX(qc[pos[2]], qa[0]))
        smc.add(q.H(qc[pos[3]]))
        smc.add(q.CX(qc[pos[3]], qa[0]))
        smc.add(q.H(qc[pos[3]]))

        # Measure the ancillas
        smc.add(q.Measure((qa[0]), (ce[j])))

    # Return the segment
    return smc
```

We initialize an empty fault tolerant error correction, or `ftec` circuit and add the `ftsmc` and `smc` segments.

```
# Initialize an empty circuit
ftec = q.Circuit()

# Initialize a list of ftsmc segments
ftsmc_segments = []

# Add the 0th segment to the list
ftsmc_segments.append(ftsmc(0))
```

```
# Add the 0th segment to the circuit
ftec.add(ftsmc_segments[0])

# Store the smc segment as a variable
smc_segment = smc()

for i in range(1, 4): # i=1,2,3
    # Add the ftsmc segment to the list
    ftsmc_segments.append(ftsmc(i))

    # If not, we add the next ftsmc circuit,
    # condition attached to the ftsmc segment before i
    ftec.add_if(q.Condition(c[2*i - 1], "0"), ftsmc_segments[i], ftsmc_segments[i-1])

    # If the previous measurement was 1,
    # we run the smc circuit with condition attached
    # to the ftsmc segment before i
    ftec.add_if(q.Condition(c[2*i - 1], "1"), smc_segment, ftsmc_segments[i-1])

# Finally, we check if the last measurement was a 1, and if so, we add the else circuit
ftec.add_if(q.Condition(c[7], "1"), smc_segment, ftsmc_segments[3])
```

Now, we add the correction gates for both the `ftsmc` and `smc`. The corrections for the `ftsmc` need to be specified in the conditions of the last `ftsmc` segment. The `smc` corrections are attached to the end of the `smc` segment.

```
# Adding the conditional gates that come after the last ftsmc
corrections = {
    1:q.X(qc[0]),
    2:q.Z(qc[2]),
    3:q.X(qc[4]),
    4:q.Z(qc[4]),
    5:q.Z(qc[1]),
    6:q.X(qc[3]),
    7:q.Y(qc[4]),
    8:q.X(qc[1]),
    9:q.Z(qc[3]),
    10:q.Z(qc[0]),
    11:q.Y(qc[0]),
    12:q.X(qc[2]),
    13:q.Y(qc[1]),
    14:q.Y(qc[2]),
```

```
    15:q.Y(qc[3])}

# iterating through the corrections and adding each correction gate
# as a segment through the add_if method
for i in range(1, len(corrections)+1):
    pattern = "{0:b}".format(i)
    pattern = "0"*(4 - len(pattern)) + pattern
    ftec.add_if(q.Condition((c[0], c[2], c[4], c[6]), pattern),
                corrections[i],
                ftsmc_segments[-1])

# Adding the correction gates that come after the smc_segment
# with explicit add_if statements
ftec.add_if(q.Condition(ce, "0100"), [q.Z(qc[2]), q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "1100"), [q.X(qc[1]), q.Z(qc[2]), q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "0001"), [q.Z(qc[1]), q.Z(qc[2]), q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "1001"), [q.Y(qc[1]), q.Z(qc[2]), q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "0110"), [q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "1010"), [q.X(qc[2]), q.X(qc[3])], smc_segment)
ftec.add_if(q.Condition(ce, "1000"), [q.Y(qc[2]), q.X(qc[3])], smc_segment)
```

And that completes the entire fault tolerant error-correction procedure.

5.3.3 Fault Tolerant State Preparation

There are several algorithms that require a "repeat until success" philosophy. Fault tolerant state preparation is one of them [Chao and Reichardt, 2018]. However, in order to not overload the quantum computer and create unnecessary issues with memory allocation, the user needs to determine the maximum number of times the algorithm may be repeated. This number, `n` is predefined, and used throughout the algorithm. The subroutine we are repeating is called a fault toleration state preparation (ftsp). Since with each iteration that we run it, we store the results in a new classical register (classical registers cannot be overwritten), the function `ftsp` takes in an integer that represents the index of iteration, and returns the `ftsp` segment for that particular iteration.

First, we initialize our registers.

```
import qion as q

n = 5 # Number of tries
c = [q.CRegister(6) for _ in range(n)]
# Code qubits
qc = q.QRegister(5, prefix="qc")
# Syndrome ancillas
qs = q.QRegister(3, prefix="s")
# Syndrome flags
qf = q.QRegister(3, prefix="f")
```

Then, we define the i^{th} iteration of the `ftsp` circuit.

```
def ftspi(i):

    ftsp = q.ConditionalCircuitSegment()

    ftsp.add([q.H(qc[j]) for j in range(5)])
    ftsp.add([q.H(qs[j]) for j in range(3)])
    ftsp.add([q.CZ(qc[0], qc[1]), q.CZ(qc[2], qc[3])])
    ftsp.add([q.CZ(qc[1], qc[2]), q.CZ(qc[3], qc[4])])
    ftsp.add(q.CZ(qc[0], qc[4]))
    ftsp.add(q.CX(qs[0], qc[0]))
    ftsp.add(q.CX(qs[0], qf[0]))
    ftsp.add(q.CZ(qc[1], qs[0]))
    ftsp.add([q.CX(qs[0], qf[0]), q.CZ(qc[0], qs[1])])
    ftsp.add([q.CZ(qc[4], qs[0]), q.CX(qs[1], qf[1])])
    ftsp.add(q.CX(qs[1], qc[1]))
    ftsp.add(q.CX(qs[1], qf[1]))
    ftsp.add(q.CZ(qs[1], qc[2]))
    ftsp.add(q.CX(qs[2], qc[3]))
    ftsp.add(q.CX(qs[2], qf[2]))
    ftsp.add(q.CZ(qc[2], qs[2]))
    ftsp.add(q.CX(qs[2], qf[2]))
    ftsp.add(q.CZ(qc[4], qs[2]))

    ftsp.add([q.H(qs[j]) for j in range(3)])

    # c[i] is the ith classical register
    # the only time the i argument is used
```

```
    ftsp.add(q.Measure(tuple(qs) + tuple(qf), c[i]))

    return ftsp
```

Finally, we initialize the final conditions, and add the segments and conditions required.

```
# Initialize an empty circuit
ftspf = q.Circuit()

# Add the 0th iteration
ftspf.add(ftspi(0))

# Loop through the other iterations
for i in range(1, n):
    # Add the ith segment
    ftspf.add(ftspi(i))
    for j in range(6):
        # If all measurement results are 1,
        # go to the segment just added (ith segment)
        # Add this condition to the segment two iterations
        # from the end (the i - 1th segment)
        ftspf.add_if(q.Condition(c[i-1][j], "1"), ftspf[-1], ftspf[-2])
```

6 Conclusion and Discussion

A new Python based circuit-level module for QuantumION called qion was created by first listing requirements for such a language, and then surveying existing circuit-level languages against these requirements. Then, the structure of the language was described, and simple examples were given to explain syntax. Finally, more complicated examples such as the teleportation circuit, the five qubit code, and fault tolerant state preparation were written up in the new language.

For the future, features that still need to be added to qion, are the ability for users to enter custom gates through a matrix with potential parameters. Another feature would be to display the decomposition of each gate into the standard gate set for QuantumION through a

method. The circuit visualization can also be improved by providing the option for a LaTeX circuit as output.

Although there are so many existing languages like qion out there, the field of quantum programming languages is new enough that what works best is still unknown. Hence, it is of the utmost importance, especially now, to innovate and create in this field.

References

[Abraham et al., 2019] Abraham, H. et al. (2019). Qiskit: An open-source framework for quantum computing.

[Amazon, 2021] Amazon (2021). Amazon braket python sdk. https://amazon-braket-sdk-python.readthedocs.io/en/latest/.

[Asfaw et al., 2020] Asfaw, A. et al. (2020). Learn quantum computation using qiskit.

[Bell, 1964] Bell, J. (1964). On the einstein podolsky rosen paradox. Physics Physique Fizika, 1:195–200.

[Chao and Reichardt, 2018] Chao, R. and Reichardt, B. W. (2018). Quantum error correction with only two extra qubits. Physical Review Letters, 121(5).

[Developers, 2021] Developers, C. (2021). Cirq. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

[DiVincenzo, 2000] DiVincenzo, D. P. (2000). The physical implementation of quantum computation. Fortschritte der Physik: Progress of Physics, 48(9-11):771–783.

[Frey, 2021] Frey, V. (2021). Quantumion python module. http://quantumion-nas.iqc.uwaterloo.ca/www/docs/index.html.

[Frey and Rademacher, 2020a] Frey, V. and Rademacher, R. (2020a). Quantumion bindings. https://github.com/quantumion/QuantumIon-Bindings.

[Frey and Rademacher, 2020b] Frey, V. and Rademacher, R. (2020b). Quantumion schemas. https://github.com/quantumion/QuantumIon-Schemas.

[Frey et al., 2021] Frey, V., Shah, R., et al. (2021). Quantumion qion. https://github.com/quantumion/QuantumIon-qion.

[Gil, 2020] Gil, D. (2020). Quantum computing may be closer than you think.

[Google, 2021] Google (2021). Cirq basics. https://quantumai.google/cirq/tutorials/basics.

[Gossett, 2020] Gossett, S. (2020). 8 quantum computing applications 'i& examples.

[Heim et al., 2020] Heim, B., Soeken, M., Marshall, S., et al. (2020). 'quantum programming languages'. Nature Reviews Physics, 2(12):709–722.

[Häffner et al., 2008] Häffner, H., Roos, C., and Blatt, R. (2008). Quantum computing with trapped ions. Physics Reports, 469(4):155–203.

[Häner et al., 2018] Häner, T., Steiger, D. S., Svore, K., and Troyer, M. (2018). A software methodology for compiling quantum programs. Quantum Science and Technology, 3(2):020501.

[Nielsen and Chuang, 2011] Nielsen, M. A. and Chuang, I. L. (2011). Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, USA, 10th edition.

[ProjectQ, 2017] ProjectQ (2017). Projectq. https://projectq.readthedocs.io/en/latest/index.html.

[Rademacher, Richard, 2020] Rademacher, Richard (2020). Design of a real-time embedded control system for quantum computing experiments. Master's thesis, University of Waterloo.

[Rigetti, 2019] Rigetti (2019). Welcome to the docs for the forest sdk! https://pyquil-docs.rigetti.com/en/stable/#.

[Shah, 2021] Shah, R. (2021). Translating from qion to quantumion. http://quantumion-nas.iqc.uwaterloo.ca/www/docs/qion/translation˙to˙quantumion.html.

[Shah and Frey, 2021] Shah, R. and Frey, V. (2021). Circuit-level programming with qion. http://quantumion-nas.iqc.uwaterloo.ca/www/docs/qion/index.html.

[Smith et al., 2016] Smith, R. S., Curtis, M. J., and Zeng, W. J. (2016). A practical quantum instruction set architecture.

[Steiger et al., 2018] Steiger, D. S., Häner, T., and Troyer, M. (2018). Projectq: an open source software framework for quantum computing. Quantum, 2:49.