Names: Jorden Garcia undergraduate
Bishoy Hanna undergraduate
Timmethy Tran undergraduate

Quantum Programming Languages

In the world of quantum computing, we know all sorts of algorithms and strategies to actually use a quantum computer, even though we don't currently have the mechanical means of producing one. Fortunately, there are a few developments in computer science that allow us to simulate one, effectively testing what we have learned and built from raw math. These "quantum programming languages" give us some insight to how a real quantum computer could be coded based in the real, classical world we are currently limited to. To be clear, they are not real quantum languages that will be implemented with a quantum computer, but a mock of what could be done using some basic and complex functionality we have developed for current languages.

The first language to dive into is Quipper, a scalable quantum programming language focused on accuracy and algorithmic complexity. One of the earlier examples of a strong quantum language, being released in 2013, it's based in Haskell, which takes advantage of strong functional programming and unit typing style. Quipper then uses an idealized form of quantum machine called QRAM (Quantum Random Access Machine) which would be paired with a classical processor piece to simulate a qubit. Using Haskell's typing style, it then created QCData in order to represent quantum information to be manipulated with gates. More information will be explained in further detail later, but this language is known widely as an accurate source of quantum program testing.

The other language we will be discussing is ProjectQ, an open source framework for a wide range of quantum program testing and manipulation. This language was formally made open source on GitHub in 2016 and is open for anyone to contribute to, so long as they follow the submission rules. ProjectQ is written in Python, a relatively simple programming language with strong typing features, which ProjectQ can take advantage of for creating quantum types. Instead of using the QRAM model, ProejctQ takes advantage of the built in random number generator library in Python. A neat thing about ProjectQ is its diversity of functionalities it provides through many support backends, also submitted by users. Some examples are resource use estimation, creating and drawing circuits, and estimated outcomes of algorithms by taking a few shortcuts (to allow user to see what could happen given a standard deviation rather than waiting for the whole program to execute). These estimations are provided and necessary to a

degree because being coded in Python has a drawback of compile and run time. The compiler is not the strongest and some algorithms may become very costly, in which case there are provided alternative compilation options, the most popular being a C++ compiler hooked to the end of the code.

We chose these languages based on their recent release and use, diversity in strategy, and available documentation. Out of the languages presented to us by Professor Fang, we decided to go with these because in general we felt them to be more interesting. In particular, we found ProjectQ to be fascinating because it was easy to pull and test with it being open source on GitHub. We did not imagine that quantum programming languages would be so easy to implement ourselves. The wide range of support and backends also gave us more areas to explore into lightly, diversifying our knowledge on quantum computing. Lastly, the applicability across multiple platforms and functions was helpful for us to learn and play with the language.

Quipper on the other hand was most relatable to what we discussed and practiced in lectures and homeworks, as it tended to go more into detail about each individual algorithm and its implementation in a coding basis. The alternate representations and functionalities that we learned because we are trying quantum algorithms in a different environment actually helped some of us with the topics studied in class, quantum teleportation in particular. The documentation also provided very explicit details about the workings of the language through qubit, gate, and circuit notation that was relatable to class. The language and usage of tools was very similarly represented in class as well, such as the use of oracles and primitives.

There were a few common themes we noticed between the two languages. At a high level, ignoring the individual styles, both languages took advantage of random number generating. This is explicitly told in ProjectQ, but at the core of the idealized QRAM model, it does the same thing to a high degree. Since qubit spinning inherently relies to percentages when measured, RNG's help us start that process by introducing the 50/50 spin on a bit. From there, gates, circuits, and other quantum computations then alter that 50/50 spin in an identical way that we learned in class, only now idealized and testable in a machine. In essence, this is the greatest common feature among the languages.

Other commonalities are the implementation of individual algorithms we learned in class. For example, both languages heavily use circuits in their algorithms, as they are easier to code than some other tools we learned in class. The CNOT gate in particular is a popular target for use in many of the algorithms shown in the documentation.

Also, both languages take advantage of the unique typing each host language has to offer. Python, being a simplistic language with nonspecific variable declaration allows for the easy implementation of a new quantum type, the qubit, which turned out to be easily manipulatable.

Quipper on the other hand took advantage of Haskell's dynamic type creation. As mentioned before, this allowed for the creation of QCData, which is a host for many of the QC types that were created with quantum properties in mind.

Lastly, both languages attempt to mock what we were able to do on paper, and then play with other aspects of the algorithm. These languages were not designed to create breakthroughs in creating and solving new or old problems. They were designed with the purpose of putting something real and physical to all the theoretical math we feel we have solved on paper. This essentially is a preliminary test for what is to come, and it turns out to be fun, helpful, interesting, and even inspiring, seeing as so many approaches were taken to quantum languages.

A few key differences are apparent in these languages however. The overall goal to prove and test algorithms may be the same, but the implementation on how they are done are quite different with these languages. The aura about them is also quite different which is apparent by their coding style and contribution.

Quipper is a more closed on language, used mostly for show and has a harder time of getting your hands on. Not listed as an open source, it seems to be the more professional approach to creating a quantum language, and rightfully so for the goals the designers have in mind. Quipper is a language focused on accuracy and in depth analysis, which is apparent in the documentation. Being written in a functional language also helps to improve this as they tend to be more accurate in some implementations, especially compared to Python. It also provides us with a different window on how to take coding a theoretical science. Quipper also uses QRAM, their version of creating qubits, which in theory seems easy to translate to a real quantum machine, so already it's looking towards the future.

Contrasting Quipper in more ways than not, ProjectQ aims to satisfy a larger audience and area of study in quantum computation. Using Python, a more laid back and easier to use language gives users an easier time implementing what they are actually trying to prove, rather than wrestling with the coding framework. Also, being open source on GitHub allows for anyone to contribute, whether it be ideas or actual code. This means that there is a lot less focus on topics, unlike Quipper. That's why there are so many different functionalities that can be sued in ProjectQ, from estimations, calculations, backends, or other supporting platforms. ProjectQ doesn't attempt very hard to visualize what the actual model will look like however. It's written pretty clear how they manage qubits and it is less sophisticated than Quipper's QRAM model. But again, for the overall goal of usability and friendliness, these attributes are important for ProjectQ to have.

Overall, at a high level, these tools have a similar goal but accomplish it in different ways, and as such it allows you to pick and choose how you would like to go about attempting to

code a quantum algorithm. Quipper provides a much more dense yet accurate and documented framework for stricter application and testing. ProjectQ provides a more open approach to solving a variety of problems with an even larger repertoire of tools and functionalities. It all depends on how the user sees their own style and problem. Next, we will explain in more detail about each language. The GitHub repository for ProjectQ is also linked below, ready for cloning and use.

https://github.com/ProjectQ-Framework/ProjectQ

"Quipper: A Scalable Quantum Programming Language"

Introduction:

This article is talking about how we can use the programming language Haskell in order to represents qubits and quantum computing. In addition, it discusses the introduction of the programming language, and it is used to solve some of the complex computing devices, by using some of the algorithms in which the humans can understand, then translating these things into a form to be executed by the machine.

Quantum Computing:

The law of quantum physics is the law which is responsible for storing and manipulation of the data. Also, any idealized quantum device can be defined in two terms: its state and its operations.
The quantum bit is considered the smallest unit of information in the quantum computing, and it also has other name which is qubit while the state of one qubit is a really sophisticated linear combination of two bases vectors which are ket zero and ket one |0> and |1>. In a similar way, the state one of the two qubits can be described as a linear combination of four basic vector {|00>,|01>,|10>,|11>}; in a general way the state of an n qubits can be defined as a linear combination of 2^n basis vector. The gates are a quantum device which has some built-in set of elementary unitary transformations. In addition, we can measure many qubits in a multi-qubit state by using an analogous rule.

Quipper: A Scalable Quantum Language

As we currently do not have the capability of creating and using real quantum computers due to the lack of technology capable of handling quantum physics, Quipper is a great way to simulate quantum computation. This is a made up language to provide a "scalable, expressive, functional, higher-order quantum programming language." This language is capable of quantum computation on classical computers through using over a trillion gates. Quipper was not imagined on its own, rather a compilation and improvement over other attempts at quantum

computation on classical devices, such as QCL, one of the oldest concrete and C-based languages.

Basic Component

In Quipper, QRAM is the device being used to run the quantum algorithms. The device works by attaching itself to a classical processor, and holds individually accessible qubits. You can then control the QRAM by applying a few kinds of instructions including unitary operations, which we will go into further detail later. Another instruction that can be applied is initialization, which essentially just resets the qubit to 0. Many other techniques can be used here like oracles, circuits, primitives, etc.

A) Interacting with a quantum device

Any n individually addressable qubits can be held by a device in which the n is a constant number. The quantum device's operation can be controlled by two kinds of instructions. The first one are the unitary operation in which they take the form of "apply the built –in unitary gate U to the qubit k" also "apply the gate V to qubits j and K."

B) Basic properties

The implication of the quantum information cannot be repeated by the laws of quantum mechanics, and this is called no cloning property of quantum mechanics. In order to apply a 2-qubit quantum gates to qubits k and k, this would not be defined as physically possible. Quantum programming language makes sure that there are no physical operations are used, and this type of property can be either checked at the compile time or at the run time.

C) Hardware independence

The actual quantum computer can be difficult to control as it has some relative short life span of the quantum states in experimental setting, and there are several layers of quantum error collection and control which is required for making the quantum computation be able for having a meaning. The quantum circuit can be described as a sequence of pre-computed gates, and this kind of quantum computation can be defined as circuit model. There is one operation which can be done by QRAM model, but in the circuit model, it currently cannot be done. Also, the sequence of the quantum gates can be varied in order to respond to the previous measurement results.

Techniques used in quantum algorithms

A good quantum programming language has to be consistent enough in order to make the quantum algorithms be expressed at a high or low level of abstraction which can be as close as possible to the algorithm of the human design intention.

A) Quantum primitives

There are two algorithms which are using the primitive building blocks, and they are: the quantum Fourier transform, and phase estimation.

- Quantum Fourier Transform: is a unitary change of the analog basis to the classical Fourier transform, and it can be used in many quantum algorithms as finding the period of a periodic function.
- Phase estimation: it is a technique which is used for estimating the eigenvalues of a unitary operator. There is a method which is called state distillation, and this method is used in the large noise of quantum states, and gradually narrows them to smaller set of states.

B) Oracles

An oracle is a classical function, and it can be described as f: $Bool^n$ -> $Bool^m$, and it describes some sides of the input to the algorithm as the graph's edges, the winning possible of the game, arithmetic or number-theoretic functions, and so on. In order to use this function in the quantum computing, the oracle must be reversible. This can be done by rising the function such that f $Bool^{n+m}$ -> $Bool^{n+m}$: which can be described as f(x,y) = (x,y (xor) f(x)). The oracle can use quantum register instead of using data types as integers, real numbers, and edges of graph.

A Quantum Data Type

In order for some quantum algorithms to be run correctly, we need some quantum data types. Quipper can provide this in the form of QCData, a data type based of the Haskell's version of a type class. Type classes can be described as a wrapper class in that they are inductively defined by property in which a type can satisfy, which will then come with its own functions. Based on this description, QCData is an overall controller for quantum data, which has two basic data types, a qubit and bit. Other types of data covered under QCData include QShape and QDInt. In order to initialize these data types, Quipper uses qinit and measure as basic quantum manipulation of its data. Some examples the use of this code are shown below:

```
(p,q) <- qinit (False,False)
measure :: QShape b q c => q -> Circ c
```

Quipper will use its extensive built in circuit generating functions to use these functions, which we will discuss later. It's worth noting that these data types are merely used to represent quantum data and aren't actual qubits inside the QRAM, therefore we will be limited on larger computations.

Quipper

Quipper is defined as an embedded functional programming language for the quantum computation, it also can support a unified general programming framework. This provides three things, quantum circuit notation, quantum algorithm notation, and circuit transformation notation.

A) Quipper as an embedded language

Haskell is the most nearest language which can describe Quipper, and it is chosen because Quipper has many higher-order and overloaded operators, whose implementation uses heavy advanced features as Haskell's type system which includes many GHC extensions. Both Haskell and Quipper are strongly functional programming language; thus, they can adapt with each other.

B) Quipper as extended circuit model

The quantum circuit model can be presented with unitary gates and circuits.
There are some aspects for the extended circuit as:

- Ancillas and scope: it makes the qubits whose state is |0> be outside certain well-defined areas where ancilla is being used. Also, all its gates must be unitary. For the compiler of the quantum programming languages, there are many benefits to go over the ancillas explicitly.
- Assertive termination: It is a termination in which the gate -| 0 terminates a qubit through asserting that, and it is in state |0>. This is called the assertive termination, and it is used to differentiate it from the ordinary termination which indicates -|. The goal of the assertive qubit termination allows some aspects. The first one: is noting that it is the programmer and not the compiler who is making sure that the qubit is in the state |0> before being terminated. The general idea is that the correctness of such an assertion relays on the complexity of the specific algorithm, and it is not something which can be verified by the compiler automatically. Thus, the programmer is responsible to make sure that the only correct assertions are made. In this case, the compiler is free to depend on this assertions as applying the optimizations which are only correct if these assertions are valid and correct. The second thing is making sure about the initializations of the circuits which has the qubits, and the assertive termination can't be in a mixed state, and it should be in an appropriate sense as the unitary and reversible.

C) The two run times

The quipper has three different phases of execution, compile time, circuit generation, and circuit execution.

1) **Compile time:** Same compile time as Haskell
   The input: source code, and compile time parameters. The output: Executable object code.
2) **Circuit generation time:** Takes place on a classical computer in an on-line environment.
   The input: Executable object code, and circuit parameter. The output: representation of the quantum circuit
3) **Circuit execution time:** Takes place on a physical quantum computer in an on-line real-time Environment.
   The input: quantum circuit, and some circuits input. The output: circuits output.
   The circuit generation time, and circuit execution time are considered as the "two run-times"

In the circuit generation time, the word "parameter" is used to indicate a value which is known, while in the circuit execution time, the word " input" or "state" is used to indicate a value. The difference between the inputs and parameters must be taken seriously, and they require particular programming language to support them. For example, in the circuit generation, the

inputs are unknown, so in the “if-then-else” operation can be put under a condition as a boolean input. After that, the then-part and the else-part must be generated in the circuits.
The quipper language has three fundamental basics types for bits and qubits.

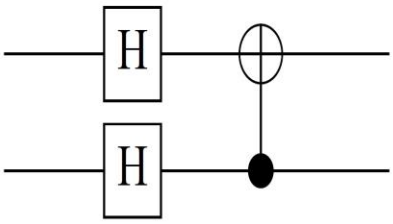
1) Bool: a boolean parameter which is known at the circuit generation time.
2) Bit: a boolean input; for example, a boolean wire in a circuit.
3) Qubit: a qubit input ; for example, a quantum wire in a circuit.

D) Circuit as described language

Quipper is a quantum programming language which operates through sending gate-by-gate instructions in real time to some physical quantum device. The quantum algorithm can be explained as high conceptual level, and there are many tasks in the construction of the algorithm which requires some manipulations at the level of the entire circuits rather than the individual gates; for example, the operations which include inversion, iteration, ancilla managements, and circuit transformations.

There are aspects related to the circuits as the paradigm, block structure, circuit operators, and the run functions.

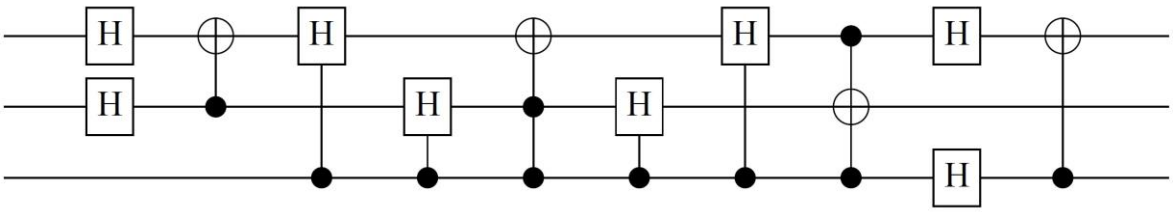
1) **The paradigm**: The basic idea of the paradigm is that the qubits are stored in variables and gates are applied to them in one time. Quipper can offer the basic abstraction, and it is the quantum operation which is a function that has an input for some quantum data, do some state changes on it, and output this quantum data which was changed. This operation is called Circ. One example is the CNOT gate.




```
mycirc :: Qubit -> Qubit -> Circ
(Qubit, Qubit)
mycirc a b = do
a <- hadamard a
b <- hadamard b
(a,b) <- controlled_not a b
return (a,b)
```

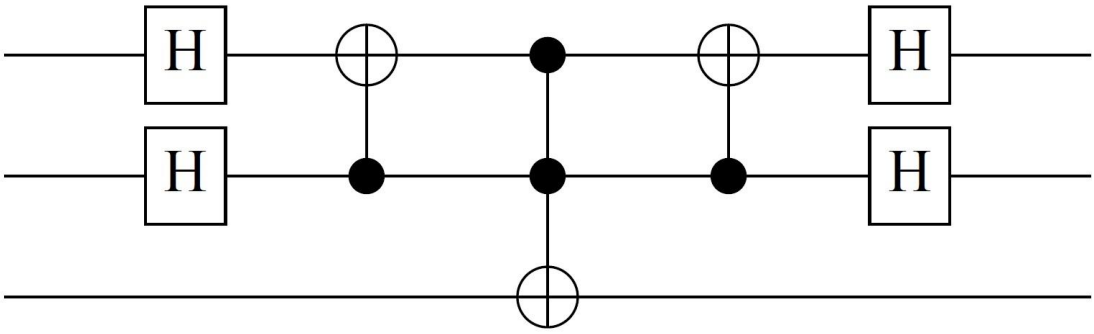
2) Block structure: Quipper supplies operators in order to define block structure into circuit. For example, the operator **with_controls :: Qubit -> Circ a -> Circ a**; in the operator, the qubit controls the entire block of gates. There is an example which explains how the subroutines can be used for building up complex circuits from simpler one.

```
mycirc2 :: Qubit -> Qubit -> Qubit
-> Circ (Qubit, Qubit, Qubit)
mycirc2 a b c = do
mycirc a b
with_controls c $ do
mycirc a b
mycirc b a
mycirc a c
return (a,b,c)
```



3) Circuit operators: The Quipper supplies the circuit with powerful higher-order operators which does operation on the whole quantum functions. Beside the high level operators which are supplied by the Quipper, these operators are responsible for reversing, iterating, and transforming quantum procedures the same as the general mechanism for changing classical boolean procedures to quantum oracles. One example is the reverse-simple operator which it takes a quantum function and returns its inverse.

```
timestep :: Qubit -> Qubit -> Qubit
-> Circ (Qubit, Qubit, Qubit)
timestep a b c = do
mycirc a b
qnot c ‘controlled‘ (a,b)
reverse_simple mycirc (a,b)
return (a,b,c)
```



4) Run functions: It is a subroutines used to run a circuit on a quantum device, or construct and manipulate it in memory. The things which are done to the circuit is determined by using different run functions for the Circ monad as the function prit-generic which is used for printing a circuit in a number of outputs formats that is available. Also, there is another function which is run-generic that is supplied by the Quipper, and this function is used for simulating a circuit.

E) Quipper as extended quantum data types

The abstract view of the quantum data notion is supplied through using the Haskell’s type classes, and the strength of this classes type can be defined by induction on the structures of types. The most fundamental basics members of this class type are Qubit and Bit which represent a quantum bit and a classical bit in a circuit.
For example:

```
instance (QCData a, QCData b) => QCData (a,b) where ...
instance (QCData a) => QCData [a] where ...
```

Other example is the controlled-not which is built in the Quipper function

```
controlled_not :: (QCData q) => q -> q -> Circ (q, q).
```

In addition, Quipper supplies a type class which is QShape, and it works by taking three arguments and representing the relation between the quantum input, and the classical parameter versions of type.
For example:

```
instance QShape Bool Qubit Bit
instance (QShape b q c, QShape b' q' c')
=> QShape (b,b') (q,q') (c,c')
instance QShape IntM QDInt CInt
```

F) Oracles in Quipper

They are boolean functions which represent reversible quantum circuits. One of the algorithm which uses the oracles is the Shor's factoring algorithm, and this algorithm depends on an oracle in order to calculate and compute the modular exponentiation f(x) = □□ (□□□ □) , where N is the integer to be factored.

The implementation of the quantum oracle can be done by hand by using four steps:

1) Define the oracle as a classical program acting on classical data types.
2) Translating this program to a classical circuit for the size of the given input.
3) Convert the classical circuit to the quantum circuit.
4) Making this quantum circuit reversible by using the standard trick for replacing the function x -> f(x) by using the reversible function (x,y) -> (x,y xor f(x)).

There is example which shows a simple oracle that takes a list of boolean as input, and output their parity (even or odd). This can be expressed by using this functional program:

```
build_circuit
f :: [Bool] -> Bool
f as = case as of
[] -> False
[h] -> h
h:t -> h `bool_xor` f t
```

The Triangle finding algorithm in Quipper

A) Background

This algorithm is given by an undirected graph G which has exactly one triangle. This graph is given by an oracle function, and two nodes v,w of graph G, so f(v,w) = 1 if (v,w) is an edge of the graph G and f(v,w) = 0 otherwise. In order to solve this problem, we need to find a set of vertices (e1,e2,e3) which form the triangle by querying f.
In addition, the triangle finding algorithm works through forming a Grover-based quantum walk on a large graph H which is called Hamming graph associated to G. There is also, oracle function which inserts graph G into the space $\{0.1\ldots\ldots\ldots, 2^{\square} - 1\}$ of l-bit integers, and each oracle call requires the use of modular arithmetic to be extended. The whole algorithm is parameterized on integer l,n, and r. It depends more specifically on the length l which represents the integers used by the oracle, $2^{\square}$ nodes of G, and $2^{\square}$ which represents the size of Hamming graph tuples.

B) Top level structure

The Quipper implementation of the Triangle Finding algorithm consists of six modules:

1) Definitions: global definitions used in the algorithm.
2) QWTEP: the quantum walk algorithm and its subroutines.
3) Oracle: the oracle and its subroutines.
4) Main: a command line interface.
5) Simulate: a test suite for the oracle.
6) Alternatives: alternatives and/or generalizations of certain algorithm.

All these modules can be compiled into an executable program tf.

C) Code samples

The quantum algorithm consists into twenty subroutines while the oracle function consists of eight. The code can be represented by using one of each: o4_POW17 and a6_QWSH.

1) The subroutine o4_POW17: It is an arithmetic function which is used by oracle. It is responsible for computing the seventeenth power of a quantum integer and stores the result in a new integer register. It is doing this process by raising the input x to the 16th power through using the repeated squaring subroutine and then multiply x and $\square^{16}$ in order to get the wanted result.

2) The subroutine a6_QWSH: This subroutine is responsible for implementing a walk step on the Hamming graph. The nodes of the Hamming graph which are connected to the graph G are the tuples of nodes of G; if these two tuples are different in exactly one coordinate, then they are adjacent. a6-QWSH is proceeded in two steps; the first one can be done by choosing an index i and a node v of G. The second step can be done by exchanging a Hamming tuple T by an adjacent one $\square'$ through exchanging the ith component of T with v, and modifying the register which contains the edge information related to the nodes in $\square'$.

D) Aggregate gate counts

There is a command line

```
./tf -f gatecount -O -o orthodox -l 31 -n 15 -r 9
```

This command is computing the gate count for oracle only with parameter values n = 15, l = 31 and r = 9. It can count the 2051926 total gates and 1462 qubits.

```
./tf -f gatecount -o orthodox -l 31 -n 15 -r 6
```

This command is producing the gate counts for the entire algorithm, which includes repeated quantum steps with inlined oracle invocations. By using standard laptop, it can run to completion in two minutes and count approximately 30 trillion gates and 4676 qubits.

“ProjectQ: An open source software framework for Quantum Computing”

Introduction:

ProjectQ is a open source software framework that is a high-level quantum language that is implemented as a domain specific language embedded in Python. These characteristic make it one of the easiest, widely understandable, and cheapest (open source) quantum languages currently available. Being embedded in Python really helps it's chances of being picked up early, as we just recently learned that IBM and Google are in competition in make a usable QC, in the lifetime of QC as Python is usually an introduction language for computer science students in college.

ProjectQ Characteristics:

To enable fast prototyping and future extension the compiler is implemented in Python and makes use of novel meta functions like `Compute/CustomUncompute` which adheres to the law of QC that every ancilla qubit must be clean at the end of the computation, or before reuse as scratch space. Annotating the compute section allows the uncomputation to be done automatically by the compiler. There are a few other meta functions like this such as `control`, `Loop`, and `Dagger`. ProjectQ can be compiled in a different language such as C++ using `pybind` if certain compiler components prove to be a bottleneck. As the name implies ProjectQ is able to support both near-term testbeds and future large-scale quantum computers like the IBM and Google ones mentioned earlier. In the backend ProjectQ integrates a quantum emulator allowing it to simulate quantum algorithms by taking classical shortcuts and obtaining speedups of large orders of magnitude. ProjectQ has it's own simulator at the low level which has been tested with actual hardware and allows the backend to run on the IBM Quantum Experience. ProjectQ does take advantage of high-performance C++ code for some of the high-performance kernels, but hides them behind a Python interface.

ProjectQ features:

At the beginning of every ProjectQ implementation the main engine must be imported into the Python code using the `MainEngine` class that contains all the compiler components as well as the back end. If the engine is created without arguments then the default compiler engines are used with a simulator backend. The arguments that can be passed in are the part of the compiler that you want to have run, this includes a list of things you want to keep as part of the compiler pipeline and the backend you wish to use if it's not the simulator. Of course every quantum algorithm operates on qubits which are created by calling the `allocate_qubit` function. Gates like the Hadamurd gate or measuring can be done on a qubit by first importing the gates with import and their respective library names. To apply them to a qubit we use the syntax:

```
H | qubit1
Measure | qubit1
```

Fig. 1

This syntax is how we simulate work on a qubit for example our Hadamurd gate usually looks like H |qubit1> and this is equal to application in Fig 1. After a measurement we can convert the qubit into an int to see the final result of our algorithm.

High-level language

The basics

Just like any other high level language ProjectQ has logical qubits from which more complex types can be built like quint, qufixed, and qufloats which are quantum integers, quantum fixed point number, and quantum floating point numbers, respectively. All of which are similar to their classical counterparts with the underlying contents of a list of qubits, quantum registers or qureg, being the main interpretation difference. To acquire an instance of n-qubit quantum registers we call the function:

```
qureg = eng.allocate_qureg(n)
```

Fig. 2

from the `MainEngine` with the number of qubits as a parameter. The function returns a quantum register of length 1, or a single qubit and they can be allocated at any point in the program so user doesn't need to specify a max number of qubits needed in any part of the code. The compiler takes care of deallocating qubits by exploiting Python's garbage collection and allow for the reuse of qubits, however the user can still deallocate the qubits themselves with Python's `del` statement. The compiler also takes care of the lifetime of a qubits allowing for parallelization for back-ends featuring more qubits than the minimal circuit width. The simulator itself can be used as a debugging tool to validate uncomputed sections since the compiler throws an exception when qubits in superposition are deallocated. This means that qubits have to be measured or uncomputed prior to deallocation. Certain subroutines, like multi-controlled NOT, that do not required clean ancilla qubits in a defined computational basis state but works with borrowed qubits in an unknown arbitrary state, and that can guarantee that after the completion of the circuit the "dirty" ancilla qubits have returned to their starting state can be optimized by the compiler by the allocation of these ancilla qubits by providing a qubit which is currently unused, independent of its state with a state like:

```
qubit = eng.allocate_qubit(dirty=True)
```

Fig. 3

Quantum gates and functions

Operation on quantum data types can be performed using either normal Python functions or with ProjectQ gate. Python functions implementing one of these operations applies other gates or functions to achieve the desired effect. So the function will bring in a qubit and use the syntax we seen earlier to operate on it. With ProjectQ gates we can use that same syntax to call a function that does the samething as our Python function. For example here are the two functions and function calls side by side for comparison:
Python function:

```
def add_constant(quint, c):
  QFT | quint

  # addition in the phases:
  phi_add(quint, c)

  get_inverse(QFT) | quint
```

Fig. 4

Python call:

```
addConstant(myQubit, 11)
```

ProjectQ function:

```
class AddConstant(BasicGate):
  def __init__(self, c):
    self.c = c # store constant to add

# provide one possible decomposition:
register_decomposition(AddConstant,
    add_constant_decomposition1)
```

Fig. 5

ProjectQ call;

```
AddConstant(11) | myQubit
```

The ProjectQ custom gates involve more code but is superior to function-base implementation and results in cleaner user code. The gate based approach provides better optimizations at higher levels of abstraction since the function call executes the individual operations right away.

Metafunctions

Metafunctions allow complex gate operations to be modified. They facilitate optimization processes in the compiler and help the user to write cleaner more concise code. The control metafunction conditions an entire code block one or more qubits being in state |**1**> passed into the function. Dagger is the same dagger that is common in quantum mechanics, it inverts an entire unitary code block. Loop runs a blocks of code for a number of iterations passed into the function. The backends that support the loop instruction will receive the loop body only once, otherwise the loop is unrolled. The compute/uncompute functions annotates section of code. This allows to optimize the condition execution of such sections. Each of these functions bring in the engine that was created for the code.

Backends

The best feature of ProjectQ is the many backends it supports. For example the ProjectQ emulator allows for complex algorithms like Shor's factoring on large numbers in 3 minutes on a typical laptop. As mentioned earlier ProjectQ has it own high-performance simulator which supports AVX(Advanced Vectors extension) instructions and OpenMP threads. This simulator has been test on the the IBM hardware and outperforms all other existing quantum simulators The simulation of quantum circuit at the level of gates is very useful in simulating the effects of noise. Now back to emulation, it's achieved by the simulator as well. There are shortcuts in the

emulator that all it to have faster execution by orders of magnitude. For example the AddConstant gate, with a small modification can be carried out directly without having to decompose it into a QFT and phase shift gates. Basic math function like the one in BasicMathGate are executed by gates and provide a Python function mimicking this behavior in the __init__ function of these functions. Many of these function require large amounts of ancilla qubits to perform the computation and these extra qubits don't need to be simulated when this kind of shortcut is used. The ProjectQ emulation feature uses the same code that is used for resource estimation with only a change to one line of code.The resource estimation backend is not currently ready but the developer plan to have it out soon. However, the developers do have an idea of how this backend will be implemented. It keeps track of all gate encounters and measures the maximum circuit width at the level of the point it is inserted into the compiler chain, considering it can be insert at any point in it. Now for the most useful back-end in the bunch the circuit drawing one. This back-end allows any quantum program to be drawn out as circuit diagrams for publication. By using the get_latex() member function the programmer gets back a LaTex code that is ready for publishing. This backend will be extended with more feature in the future allowing the user to have more power over the mapping and drawing process.

Reference:

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoˆıt Valiron (2013). “ Quipper: A Scalable Quantum Programming Language.” 1304.3390v1 [cs.PL] 11 April 2013

Steiger, Damian S., Thomas Häner, and Matthias Troyer. "ProjectQ: An Open Source Software Framework for Quantum Computing." *[1612.08091] ProjectQ: An Open Source Software Framework for Quantum Computing*. N.p., 23 Dec. 2016. Web. 22 May 2017.