Christopher J. Wright

ID: 42807671
Supervisor: Prof. Thorsten Altenkirch
Module: G53IDS

2018/19

G53IDS Dissertation

Quantum Computing in Haskell

Submitted April 2019, in partial fulfilment of the conditions for the award of the degree BSc Computer Science

Author: 42807671, Christopher J Wright, psycjw
Supervisor: Prof. Thorsten Altenkirch, psztxa

School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text.

Signature

Date 12/04/2019

Abstract

Quantum computing is growing daily as new quantum computers are created, and new research is done. There are already many desirable uses of quantum computing, such as factoring integers in $O(logN)$ [Sho94] and searching unsorted databases in $O(\sqrt{N})$ [Gro96]. There are even quantum games being created, such as Quantum Battleships [Woo17].

As the field expands, more people will find interest in the field and attempt to learn and pursue a career concerning quantum computers, and so suitable learning and experimental platforms need to be present to help this happen. This is the main inspiration for this project; create a stable up-to-date platform for users to run quantum computations on a classical computer via quantum simulation, while also improving my own understanding and experience of the subject for the future.

This dissertation explains the basics of quantum computing, and provides possible methods for the implementation of certain quantum algorithms on a classical computer using Haskell, with a quantum-computation simulating package called QIO [AG09]. This package already includes structures necessary for quantum computations, such as the qubit and unitary functions, however does not work on the latest Haskell update (v8.4.3) and lacks complex quantum algorithms. Therefore, the package is first updated, and then two quantum algorithms are implemented. Following this, a quantum circuit builder is designed and implemented in the package, allowing users to create and test quantum circuits using a intuitive GUI.

Each part of the project, namely the implementation of two algorithms, Grover's algorithm and Shor's algorithm, and the quantum circuit builder, is completely separate and functions independantly. Due to this, I split the project up into three parts. Each part is designed, implemented, tested and evaluated before moving to the next part - avoiding confusion between parts and ensuring one part is complete before attempting the next.

Contents

1 Introduction to Quantum Computing

1.1 Brief History

The idea of a quantum computer was first introduced in a talk in 1981 by Nobel prize-winning physicist Richard P. Feynman. In his talk, reproduced in the *International Journal of Theoretical Physics* in 1982 [Fey82], he proposed that it would be impossible to simulate a quantum system on a classical computer efficiently. At the same time, physicist Paul Benioff demonstrated that quantum mechanical systems can model classical Turing machines [Ben82], and created the first recognizable theoretical framework for a quantum computer.

This resulted in David Deutsch's creation of the first universal quantum computer [DP85]. This computer could do things that a universal Turing machine cannot, such as generate truly random numbers, and simulate finite physical systems.

Over time, quantum algorithms have been developed which solve problems faster and more efficiently than classical algorithms can, demonstrating the main use for quantum computers. The main examples of such algorithms would be the Deutsch-Jozsa algorithm [DJ92], Shor's algorithm [Sho94] and Grover's algorithm [Gro96].

1.2 QIO

QIO is a Haskell package built by Alexander S. Green and Prof. Thorsten Altenkirch [AG09], which allows users to simulate a quantum computer on their own classical computer. Typical quantum operations are built up of unitary (reversible) functions under a monoid `U`, and can then be run using a monad `QIO` and the function `applyU`. This `QIO` monad also has functions `mkQbit` and `measQbit`, which can be used to create a new qubit and measure an existing qubit respectively.

This leads to an intuitive platform which can be used to create quantum computations, and then run/simulate them with 2 very simple functions `run :: QIO a -> IO a` and `sim :: QIO a -> Prob a`, where `run` is used to execute a quantum computation and return the value of qubits measured based on their probabilities and a pseudorandom number generator. The `sim` function is similar, however after executing a quantum computation, it returns the probability distribution for the qubits measured, showing the possible states and their probabilities when measured.

Throughout this dissertation, this package is frequently used to help introduce and teach quantum computing, providing relevent code segments to improve clarity, and create programs which highlight certain features of quantum computations.

1.3 The Qubit

A classical computer uses a bit to store data. This bit can only be in one of two states, 0 or 1. Strings of bits are then used to represent data and instructions, forming the basis of classical computation.

However, a quantum computer uses a quantum bit, known as a qubit. This qubit can simply be in the classical states 0 or 1, or in a superposition of both, i.e., be in both states at once. This means an operation performed on a qubit effectively acts on both 0 and 1 at the same time.

A qubit can be described by a linear combination of $|0\rangle$ and $|1\rangle$,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha^2| + |\beta^2| = 1$$

where $|\alpha^2|$ represents the probability of measuring the 0 state, and $|\beta^2|$ the 1 state. Hence the normalization condition $|\alpha^2| + |\beta^2| = 1$. This condition also corresponds to the fact that $|\psi\rangle$ is a unit vector in the complex vector space.

Measuring a qubit will collapse its wave function due to Born's rule [Bor]. This means that, when measured, a qubit will collapse to be either $|0\rangle$ or $|1\rangle$. For example, if we take the qubit

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

and measure it, the probability of getting $|0\rangle$ or $|1\rangle$ is $\frac{1}{2}$. We can therefore say that the qubit $|+\rangle$ is in an equal superposition.

In QIO, a qubit is referred to as a `Qbit`. One can be created simply using the `mkQbit :: Bool -> QIO Qbit` function mentioned above, and is created based on the boolean value passed in. The function `measQbit :: Qbit -> QIO Bool` can then later be used to measure the state of the qubit passed in.

```
make2qubits :: QIO (Bool, Bool)
make2qubits = do
    q0 <- mkQbit False -- This will create a qubit, q0, in the state |0>
    q1 <- mkQbit True -- This will create a qubit, q1, in the state |1>
    b0 <- measQbit q0 -- This will measure the qubit q0, returning False
    b1 <- measQbit q1 -- This will measure the qubit q1, returning True
    return (b0, b1) -- Returns (False, True)
```

Qubit base states can also be represented as vectors, with $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$. The qubit $|\psi\rangle$ above can be rewritten, introducing a multiplier known as the global phase, making the coefficient for $|0\rangle$ (i.e., α) real and non-negative. For example, the state $\frac{-i}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle$ is the same as $\frac{1}{\sqrt{2}}|0\rangle + \frac{-1}{\sqrt{2}}|1\rangle$ with a global phase of $-i$. This global phase does not affect the amplitudes of the states, and so doesn't change the qubit's value when measured.

This qubit state $|\psi\rangle = \alpha'|0\rangle + \beta'|1\rangle$ can now be written in the form

$$|\psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})|1\rangle, \qquad \theta = 2\cos^{-1}(\alpha'), \quad \phi = \Im(\ln(\frac{\beta'}{\sin(\frac{\theta}{2})}),$$

$$0 \leq \theta \leq \pi, \quad 0 \leq \phi \leq 2\pi$$

From this, it is easy to see that a qubit can be represented geometrically using the angles θ and ϕ, on what is known as the Bloch sphere.

1.4 The Bloch Sphere

The Bloch sphere, named after physicist Felix Bloch, is a geometrical representation of a two-state quantum mechanical system (e.g. a qubit). As shown above, a qubit's coefficients can be expressed in terms of 2 angles, θ and ϕ. This makes it extremely easy to see what happens when operations are performed on qubits, and it is clear that a qubit pointing to a point above the origin has a higher $|0\rangle$ amplitude than $|1\rangle$, and vice versa.

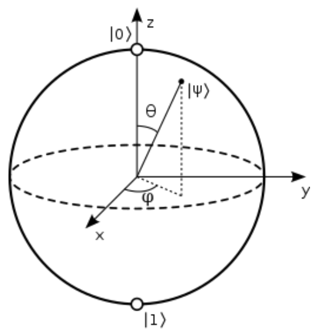


Figure 1: A Qubit represented geometrically on the Bloch Sphere

We can calculate the angle θ using QIO's underlying structure and the `sim` function. The angle ϕ cannot be calculated in this way, as we use the probabilities for each state for the calculation.

The `sim` function returns a value of type `Prob a`, where `a` is given by the quantum computation it is applied to (i.e. applying `sim` to a function of type `QIO Bool`, will return a value of type `Prob Bool`). This `Prob` type is simply a wrapper around vectors given by real probabilities, defined in the `Qio.hs` file:

```
data Prob a = Prob {unProb :: Vec RR a}
```

where `RR` is simply a `Double` value, and `Vec` is a wrapper around a list of pairs, defined in the `Vec.hs` file:

```
newtype Vec x a = Vec {unVec :: [(a,x)]} deriving Show
```

From this, it is easy to see that

```
Prob Bool = Prob {unProb :: Vec RR Bool}
          = Prob {unProb :: Vec {unVec :: [(Bool, Double)]}}
```

Now, we can extract the probabilities for qubits, and calculate θ.

```
quantum :: QIO Bool
quantum = do
    q <- mkQbit True
    measQbit q

theta :: QIO Bool -> Double
theta c = 2 * (acos alpha)
    where
        ps = unVec $ unProb $ sim c
        alpha = sqrt $ snd $ last ps
```

The function `quantum` can be changed to perform any operation on the qubit, and as long as the function's type does not change, then `theta quantum` will always return θ for the qubit measured. E.g.

```
quantum :: QIO Bool        -> theta quantum = 3.141592653589793
quantum = do
    q <- mkQbit True
    measQbit q
```

```
quantum :: QIO Bool        -> theta quantum = 0.0
quantum = do
    q <- mkQbit False
    measQbit q

quantum :: QIO Bool        -> theta quantum = 1.5707963267948966
quantum = do
    q <- mkQbit False
    applyU (uhad q)
    measQbit q
```

1.5 Decoherence

Put simply, decoherence is the loss of coherence. This basically means that the quantum state described by a system is no longer in superposition, and leads to a collapse of the wave function. If this occurs during a computation, all information stored in qubits is lost, and the computation needs to be restarted. Decoherence can occur naturally after a certain amount of time as the system will be entangled with its environment [Zur91], and is the main problem quantum computing faces.

1.6 Basic Operations

1.6.1 1-Qubit Gates

A 1-qubit gate can be thought of as a rotation about the Bloch sphere, and as the state space of a qubit is continuous, there are an infinite amount of 1-qubit gates. Any complex unitary 2x2 matrix represents a 1-qubit gate, and there are 4 main 1-qubit gates: the Hadamard gate, and the 3 Pauli- gates. Note that all quantum gates must be unitary (i.e., is its own inverse).

As mentioned earlier in section 1.2, QIO uses a monoid `U` to form unitary functions. These functions are defined as 5 distinct forms, using recursion to create sequences of functions:

```
data U = UReturn
       | Rot Qbit Rotation U
       | Swap Qbit Qbit U
       | Cond Qbit (Bool -> U) U
       | Ulet Bool (Qbit -> U) U
```

`UReturn` is simply an empty function, and will always be the last unitary in a sequence. For this section on 1-Qubit gates, the `Rot Qbit Rotation U` form is used. QIO defines all single qubit gates as rotations, where a `Rotation :: ((Bool, Bool) -> CC)` represents the 2x2 matrix of the gate. The `rot` function is then used to create a unitary function for a given `Rotation` and qubit.

```
rot :: Qbit -> Rotation -> U
rot x r = Rot x r UReturn
```

The Hadamard Gate Named after French mathematician Jacques Hadamard, the Hadamard gate (also called the Hadamard rotation) takes the base states $|0\rangle$ and $|1\rangle$ into equal superpositions

$$|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

$$|1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

and has the matrix $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$. It is equivalent to the combination of 2 rotations: π about the Z-axis, followed by $\pi/2$ about the Y-axis.

Figure 2: The Hadamard Gate

The gate matrix (i.e. `Rotation`) is given by `rhad` in the `QioSyn.hs` file, along with the unitary function `uhad` which can be used via the `applyU` function.

```
rhad :: Rotation
rhad (x,y) = if x && y then -h else h where h = (1/sqrt 2)

uhad :: Qbit -> U
uhad x = rot x rhad

applyHadamard :: QIO Bool
applyHadamard = do
    q <- mkQbit False
    applyU (uhad q)
    measQbit q
```

The Pauli- Gates The Pauli- gates, created by Wolfgang Pauli, correspond to a rotation of π about one of the 3 axis, and so there are 3 - Pauli-X, Pauli-Y and Pauli-Z (often abbreviated to X, Y and Z respectively). The Pauli-X gate is most common, as it is the equivalent of a classical NOT gate.

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}, \quad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}, \quad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$$

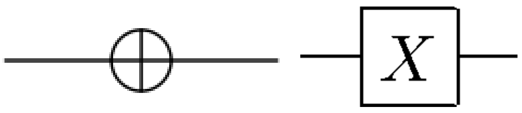


Figure 3: The 2 representations of the Pauli-X (NOT) Gate

The Pauli-X gate is defined in QIO using the `rnot Rotation` and the unitary `unot`. However, these gates can be created simply by first creating the corresponding `Rotation` matrix, and then using the `rot` function similar to how the Hadamard gate is defined.

```
rX :: Rotation                          uX :: Qbit -> U
rX (x,y) = if x==y then 0 else 1        uX q = rot q rX

rY :: Rotation                          uY :: Qbit -> U
rY (False, True) = 0 :+ 1               uY q = rot q rY
rY (True, False) = 0 :+ (-1)
rY (_, _) = 0

rZ :: Rotation                          uZ :: Qbit -> U
rZ (False, False) = 1                   uZ q = rot q rZ
rZ (True, True) = -1
rZ (_, _) = 0
```

1.6.2 n-Qubit Gates

Alongside the gates above, there are 3 other basic quantum gates; Controlled-X, SWAP and Toffoli, which are defined in QIO using the other 3 constructs for `U`.

The SWAP Gate This gate is extremely simply; it just swaps 2 qubits. In QIO, it is defined by the `swap` function.

```
swap :: Qbit -> Qbit -> U
swap x y = Swap x y UReturn
```

The Controlled-X Gate The most common 2-qubit gate, it applies the Pauli-X gate (i.e., a NOT gate) on the second qubit if and only if the first qubit is $|1\rangle$. It is often called the Controlled-NOT or cNOT gate. Note that we can do the same with the other Pauli- gates, however this is rarely used.

The gate is defined using the `Cond Qbit (Bool -> U) U` unitary construct, and the function `cond` which creates the unitary (similar to the functions `rot` and `swap` above). Essentially, when given a qubit and a unitary function which takes in a boolean, the resulting unitary will apply the boolean function to the states the qubit can be in.

This is best demonstrated with an implementation for the cNOT gate. If the qubit is measured to be `True` (i.e. in the $|1\rangle$ state), then the resulting unitary is a NOT gate. Otherwise, nothing will be done and the resulting unitary is empty.

```
cond :: Qbit -> (Bool -> U) -> U
cond x br = Cond x br UReturn

cnot :: Qbit -> Qbit -> U
cnot qc qo = cond qc (\x -> if x then (unot qo) else mempty)
```

The Toffoli Gate This gate acts on 3 qubits, and is often called the ccNOT gate. This is because it applies the Pauli-X gate on the third qubit if and only if the first two qubits are both $|1\rangle$.

```
toffoli :: Qbit -> Qbit -> Qbit -> U
toffoli q1 q2 qo = cond q1 (\x -> if x then (cnot q2 qo) else mempty)
```

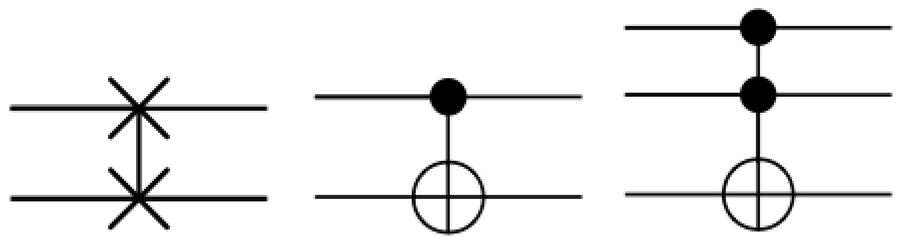


Figure 4: The SWAP gate (left), the Controlled-X (cNOT) gate (middle), and the Toffoli gate (right)

The C^n-NOT Gate The idea behind the Toffoli gate above (multiple control qubits, and one output qubit) can be extended to any arbitrary amount, creating a c^n-NOT gate. From the definition of the Toffoli gate in QIO above, it is clear that we can define a gate in QIO which acts on a list of control qubits, and one output qubit.

```
cNnot :: [Qbit] -> U
cNnot [] = mempty
cNnot (q:qs) = cond q (\x -> if x then (cNnot qs) else mempty)
```

However, this only works due to how QIO operates on qubits. On a quantum computer, a gate like this will require a sequence of toffoli gates and ancilla qubits. For example, the cccNOT gate requires an extra ancilla qubit:

```
cccNot :: Qbit -> Qbit -> Qbit -> Qbit -> Qbit -> U
cccNot q1 q2 q3 q4 qa =
    toffoli q1 q2 qa <>
    toffoli q3 qa q4 <>
    toffoli q1 q2 qa
```

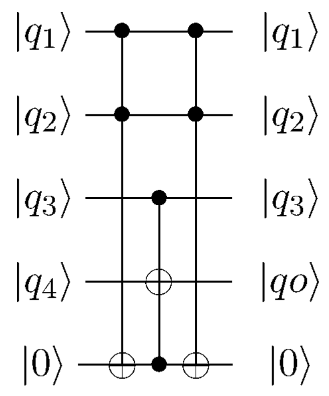


Figure 5: The cccNOT gate, made using toffoli gates and an ancilla qubit, where $|qo\rangle = |q_4 \oplus (q_1q_2q_3)\rangle$

When using ancilla qubits in QIO, we can use the unitary construct `Ulet` and the function `ulet`.

```
ulet :: Bool -> (Qbit -> U) -> U
ulet b ux = Ulet b ux UReturn
```

This construct will create a qubit in the state given by the boolean, for use in the given unitary. We can now execute the cccNot gate defined above. The qubit `q4` will be set to $|1\rangle$ if and only if `q1, q2` and `q3` are all in the state $|1\rangle$.

```
doCccNOT :: QIO Bool
doCccNOT = do
    q1 <- mkQbit True
    q2 <- mkQbit True
    q3 <- mkQbit True
    q4 <- mkQbit False
    applyU (ulet False (cccNot q1 q2 q3 q4))
    measQbit q4
```

1.7 Entanglement

Quantum entanglement is a phenomenon that occurs when the value of one qubit affects another, and forms a large part of many quantum algorithms. As mentioned above, when we measure a qubit it collapses to one of its base states. If another qubit depends on this measured qubit (e.g., via a cNOT gate), then it will also collapse to one of the base states. The 2 qubits are then said to be entangled.

Consider a circuit which starts with two qubits *Alice* and *Bob*, both starting in the state $|0\rangle$. We then apply the Hadamard gate to *Alice*, converting *Alice* into an equal superpositional state of the base states $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, meaning there is a 50/50 chance of *Alice* being 0/1 when measured. Next, we apply a cNOT gate to *Alice* and *Bob*, entangling the two qubits together. Therefore, if we measure *Alice* to be $|0\rangle$, then *Bob* will collapse to the state $|0\rangle$, and the same will occur for the state $|1\rangle$. As entanglement is non-local, *Alice* and *Bob* could be separated by any distance, and this will still occur.

This can be implemented in QIO using the `uhad` and `cnot` defined earlier. The function `entangle` below will create 2 qubits, `qa` and `qb`, and initialize them in the $|0\rangle$ state. We then apply the Hadamard gate to qa, changing its state to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and a controlled-X unitary function, concatenated simply with the mappend operator `<>`. When run using the `run` function, `[True, True]` or `[False, False]` will be returned due to the entanglement. The `sim` function will return `([True, True], 0.5), ([False, False], 0.5)`.

```
entangle :: QIO [Bool]
entangle = do
    qa <- mkQbit False
    qb <- mkQbit False
    applyU (uhad qa <> cnot qa qb)
    ba <- measQbit qa
    bb <- measQbit qb
    return [ba, bb]
```

1.8 The Bell States

The two qubits shown above, *Alice* and *Bob*, together form a Bell state, named after physicist John S. Bell

$$|\Psi_{00}\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

As evident from the equation, if one of the qubits is measured to be $|0\rangle$ or $|1\rangle$, then so is the other. There are 4 Bell states, where each state has 2 entangled qubits, and the measurement of one qubit collapses the state of the other.

$$|\Psi_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad |\Psi_{01}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

$$|\Psi_{10}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \quad |\Psi_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

Each Bell state is created using the same unitary operation, namely a Hadamard gate on the first qubit, and then a cNOT to the second qubit using the first qubit as the control bit, with the resulting Bell state determined by the original state of the 2 qubits.

2 Motivations

2.1 Problems

A quantum computer consists of multiple two-level quantum-mechanical systems, such as multiple electrons, with each electron represents a qubit. The spin of the electron determines the qubit state, with spin up representing $|0\rangle$ and spin down representing $|1\rangle$. The main problem faced here is decoherence [SP05]. The overall system is extremely sensitive to interaction with the surroundings, such as magnetic fields affecting electrons, and must be isolated from its environment. The more qubits we have, the harder it is to maintain coherence.

Due to the difficulties of creating a quantum computer, we need to be able to simulate one on a classical computer. This simulation will not be as efficient as a quantum computer, and so its main use is the testing and creation of quantum algorithms, alongside teaching quantum computing. This means there is a high demand in quantum computing simulators, hence the need for projects like this.

2.2 Simulators

As mentioned above, creating a stable and effective quantum computer is difficult, and so while new ways of implementation are being researched, we can use quantum simulators to create, test and teach quantum computing. As quantum computing grows even larger, simulators need to be available to people in order to help them learn about quantum computations and programs. More quantum simulators across different programming languages will supply a wider range of programmers the ability to practice quantum computing on their own computer, as well as offer different methods of implementing a quantum simulator.

This project aims to add to this group of quantum simulators, offering an up-to-date and reliable Haskell package which enables users to create and run quantum programs. On top of this, they will be able to run quantum algorithms over their own data sets, increasing the teaching ability of the project.

3 Related Work

3.1 IBM Q

The IBM Q Experience is an online platform [ibmd] - found easily with a Google search "ibm q" - which gives its users access to a set of prototype quantum processors. Users can create quantum circuits or programs via a graphical or textual interface, and then either run these quantum operations on an actual quantum computer, or simulate them classically.

It was launched by IBM in 2016 [New16], and started with a 5 qubit quantum processor and matching simulator, alongside a small set of two-qubit operations. Over time, the platform has evolved into a strong stable quantum platform, with the addition of more 2-qubit operations, a simulator over a maximum of 20 qubits and limited beta access to a 16 qubit quantum processor. IBM also released a python-based quantum computing framework, QISKit [Lub17], allowing users to simulate a quantum computer offline, or run quantum programs on the real IBM quantum processors online.

3.1.1 Evaluation

In my opinion, the IBM Q Experience platform is great for all who wish to create quantum programs and just generally experiment with quantum computing. A colourful, intuitive graphical interface can be used to build quantum circuits and then run/simulated with ease - perfect for new users. This still works well for more advanced users, however the textual interface is more suited for advanced use. Complex quantum programs can be created instead of circuits, and the use of the QISKit framework enables users the option of creating and simulating quantum computations without the online platform. However, this does mean users must learn a new programming language, potentially alongside learning about quantum computations, creating quite a large knowledge gap to bridge. Yet this is somewhat diminished by the availability of two user guides, a Beginner's guide [ibmb] and a Full guide [ibmc].

These guides successfully reduce the difficulty in learning the basics of quantum computing without confusing newcomers. The Beginner's guide briefly explains the qubit, and common single qubit gates. It then goes on to hint at some multi qubit gates, and explain the process of entanglement and how it can be used. The Full guide extends these sections, providing a greater depth of knowledge concerning the qubit, such as the Bloch sphere and decoherence, while going into more detail on single and multi qubit gates, with descriptions of quantum algorithms and some quantum error correction methods. More help is then available to users via a community forum [ibma] and in-depth QISKit tutorials [ibme], ensuring users can find solutions to any problems they are having.

3.1.2 Key Points

1. Simple and effective graphical user interface, allowing the user to drag-and-drop quantum gates onto a circuit, with bright colours used to emphasize different gates.
2. Basic textual interface, providing users with an interface they can use to build more complex quantum programs.
3. Multiple user guides to help teach newcomers without confusing them, while still providing aid to more advanced users.
4. Emphasis placed on usability.

4 Project Description

4.1 Aim

The aim of this project is to update and improve an out-of-date Quantum computing package in Haskell, QIO [AG09] - developed by Green and Altenkirch. This will result in an updated, stable package for Haskell v8.4.3, allowing users to simulate a quantum computer, and run various quantum algorithms. It will also provide the user a graphical interface to build and run quantum circuits.

This QIO Haskell package allows a user to create a quantum program, and then run or simulate it. By running, any qubits measured via a function `measQbit :: Qbit -> QIO Bool` will be collapsed into True or False based on their probability amplitudes and a pseudorandom number generator. However, the difference between this package and others is revealed when we simulate the program instead. Then, any qubits measured will be shown as their probabilities, including multiple qubits together. From this we can easily check the program for errors.

The package is currently not working in Haskell v8.4.3, and so first needs to be updated. The update will need to maintain the quantum aspects necessary for quantum computation, such as coherence and the no-cloning theorem, and still be able to run/simulate quantum programs inputted.

Following this, two quantum algorithms (Grover's and Shor's) will be implemented in the package, allowing users to run these algorithms on their own data. There is already an implementation of Shor's algorithm present in QIO, but this needs to be updated. Each implementation will be in a separate Haskell file, and be able to perform a quantum algorithm correctly over its inputs using the rest of the QIO package.

Finally, a quantum circuit builder/interpreter will be built. It will display a graphical interface, and allow the user to drag-and-drop quantum gates onto a circuit of any amount of qubits. It will then display the resulting amplitude(s) and/or value(s) of any of the qubits. This will be similar to the IBM Q Program specified above, only shall be run on a classical computer instead of a quantum one.

4.2 Algorithms

4.2.1 Grover's algorithm

In 1996, Lov K. Grover demonstrated an algorithm which could search a quantum mechanical database quickly [Gro96]. This quantum algorithm is quadratically faster than the fastest classical variant, returning the index of the search item in $O(\sqrt{n})$, instead of the classical $O(n)$. It can also be generalized to find solutions to an unknown predicate function f in the same time. This means that any problem which involves finding solutions to a boolean function can be solved using Grover's algorithm, and so it has a large variety of uses past searching databases, such as finding the mean and median of a set of numbers, or solving the collision problem in $O(n^{1/3})$[Aar05] instead of the classical $\Theta(\sqrt{n})$

The algorithm takes a function f, and after converting it into a unitary function, runs it over a set of qubits representing the data set to search. It then negates solution states in the solution set, and disperses them. This dispersion acts as an inversion around the mean amplitude, meaning the solution state is amplified due to the previous negation. After repeating this a number of times no greater than $\pi/4\sqrt{N}$ times, where N is the amount of variables in the function, there is a high probability that the qubits, when measured, will give a solution to the function f.

This project aims to explore potential methods of implementating this algorithm in the QIO Haskell package, as well as demonstrate other ways of executing the algorithm.

4.2.2 Shor's algorithm

Before Grover's algorithm, there was Shor's algorithm for integer factorization. This quantum algorithm is used to find the factors of any integer N in $O((logN)^3)$ [BCDP96]. The algorithm specifically looks at the case of factorizing an integer $N = pq$, where p and q are both large primes. If an integer b shares no factors with N then $b^r = 1 \pmod N$ for some integer r, and the function $f(x) = b^x \pmod N$ is a periodic function of x with period r. With the period r known, if it is even we can calculate

$$x = b^{\frac{r}{2}} \pmod N$$

Following this, if $x + 1 \neq 0 \pmod N$ we can factorize the input. The probability of choosing a random b which gives a period r with these properties is $> 50\%$ [Gre09]$^{\text{Shor's algorithm continued, page 27}}$.

After constructing a unitary function $U_{f(x)} = b^x \pmod N$, the algorithm applies it over an input state, leaving an equal superpositional state

$$\frac{1}{\sqrt{2^n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n \, |b^x \pmod N\rangle_{n_0}$$

$$n_0 = \log_2 N, \quad n = 2n_0$$

We can then measure the second register, which will give any value for $b^x modN$, leaving the first register in an equal superposition over the values for x. The algorithm then applies the Quantum Fourier Transformation (QFT) to the first register and measures it, giving a value for y where $y = 2^n/r$. Finally, we can check if r is even and if $x + 1 \neq 0 \pmod N$ where $x = b^{r/2} \pmod N$, and if they are both true, then N has factors $(x - 1)$ and $(x + 1)$. We can then find the greatest common denominators of both in N, corresponding to the 2 prime factors of N.

5 The QIO Package

5.1 Underlying Structure

QIO currently uses a HeapMap to store the states of the qubits in the system, and a Qbit type which just acts as an integer reference to this map. Unitary functions take the form of a monoid `Unitary`, and represent an operation on the HeapMap which may produce a new state. We can then restrict unitary functions to be in a few distinct forms, ensuring the functions are indeed unitary - a necessity for quantum computations. These forms are defined in a datatype `U` shown below.

```
data U = UReturn
       | Rot Qbit Rotation U
       | Swap Qbit Qbit U
       | Cond Qbit (Bool -> U) U
       | Ulet Bool (Qbit -> U) U
```

Using recursion, this type dictates is what a unitary operation will be; a string of rotations, swaps, conds and lets, ending in a UReturn. From these 4 operations, we can build any unitary operation possible, and as `U` is also defined as a monoid, we can easily concatenate operations together. Following this, the package defines another datatype `QIO a` - the type of a quantum computation:

```
data QIO a = QReturn a
           | MkQbit Bool (Qbit -> QIO a)
           | ApplyU U (QIO a)
           | Meas Qbit (Bool -> QIO a)
```

This datatype enables us to create and measure Qbits, along with applying unitary operations to them.

From all this, we can build a quantum program which creates qubits in an initial base state (i.e., either $|0\rangle$ or $|1\rangle$), applies any amount of unitary functions to them, and then measures qubits. Now we need to let the user execute a quantum program on a simulated quantum system. To do this, QIO has 2 functions: `run :: QIO a` $\rightarrow$ `IO a` and `sim :: QIO a` $\rightarrow$ `Prob a`.

Run takes a quantum computation in the type defined above, and executes it on a simulated quantum system. When qubits are measured and returned in the computation, the function prints this state value to the console using the IO monad. The state value to return will originally be a quantum state, and so is made up of probability amplitudes. Therefore, the actual state to return will be chosen at random based on these probability amplitudes and a psuedorandom number generator.

Sim, however, is a bit different. It still takes a quantum computation and executes it, however it returns the actual probabilities of the possible states the qubits measured may be in. This makes testing quantum programs fairly simple, drastically increasing the usability of the project.

5.2 Updating QIO

5.2.1 Current Errors

Currently, QIO cannot be installed on the latest Haskell version (v8.4.3). This is because the package depends on the Haskell base package version being ≥ 4.9 and < 4.10. After downloading the package, and manually changing this to allow any base package ≥ 4.9, two errors are shown depicting what is incorrect with the current QIO structure.

```
QIO/QioSyn.hs:39:10: error:
        * No instance for (Semigroup U)
            arising from the superclasses of an instance declaration
        * In the instance declaration for 'Monoid U'
...
QIO/QioSynAlt.hs:118:10: error:
        * No instance for (Semigroup U)
            arising from the superclasses of an instance declaration
        * In the instance declaration for 'Monoid U'
```

From the error messages presented, it is clear that a change to Monoids is causing errors. In the update to base-4.11.0.0, the Haskell Semigroup class was made a superclass of Monoid [bas]. This means that a Semigroup instance for `U` needs to be defined, alongside the Monoid instance.

The current definition for `U` in QioSyn.hs is shown below

```
instance Monoid U where
        mempty = UReturn
        mappend UReturn u = u
        mappend (Rot x a u) u' = Rot x a (mappend u u')
        mappend (Swap x y u) u' = Swap x y (mappend u u')
        mappend (Cond x br u') u'' = Cond x br (mappend u' u'')
        mappend (Ulet b f u) u' = Ulet b f (mappend u u')
```

All that needs to be done to fix this is to include a Semigroup definition for `U` which defines the structure for executing multiple unitary functions, and then to define mempty and mappend for the Monoid instance.

```
instance Semigroup U where
        UReturn <> u
        (Rot x a u) <> u' = Rot x a (u <> u')
        (Swap x y u) <> u' = Swap x y (u <> u')
        (Cond x br u') <> u'' = Cond x br (u' <> u'')
        (Ulet b f u) <> u' = Ulet b f (u <> u')

instance Monoid U where
        mempty = UReturn
        mappend = (<>)
```

After this, the alternate file QioSynAlt.hs needs to be updated also. The definition for `U` is very similar, however uses a non-recursive datatype in its definition.

```
instance Monoid U where
        mempty = Fx UReturn
        mappend (Fx UReturn) u = u
        mappend (Fx (Rot x a u)) u' = Fx $ Rot x a (mappend u u')
        mappend (Fx (Swap x y u)) u' = Fx $ Swap x y (mappend u u')
        mappend (Fx (Cond x br u')) u'' = Fx $ Cond x br (mappend u' u'')
        mappend (Fx (Ulet b f u)) u' = Fx $ Ulet b f (mappend u u')
```

The fix for this is very similar to before:

```
instance Semigroup U where
        (Fx UReturn) <> u = u
        (Fx (Rot x a u)) <> u' = Fx $ Rot x a (u <> u')
        (Fx (Swap x y u)) <> u' = Fx $ Swap x y (u <> u')
        (Fx (Cond x br u')) <> u'' = Fx $ Cond x br (u' <> u'')
        (Fx (Ulet b f u)) <> u' = Fx $ Ulet b f (u <> u')

instance Monoid U where
        mempty = Fx UReturn
        mappend = (<>)
```

5.2.2 New Errors

After updating the files, the package produces 2 new error messages when attempting to build, and the previous errors do not show. The same errors are occuring (lack of a Semigroup instance) but for different files, QioClass.hs and Qio.hs.

```
QIO/QioClass.hs:17:10: error:
        * No instance for (Semigroup UnitaryC)
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid UnitaryC’
...
QIO/Qio.hs:32:10: error:
        * No instance for (Semigroup Unitary)
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid Unitary’
```

Therefore, the fixes are the same as before: create a definition for a Semigroup instance based on the Monoid instance, then define mempty and mappend in the Monoid definition.

```
QioClass.hs:
instance Semigroup UnitaryC where
         (U f) <> (U g) = U (\ fv h -> g fv (f fv h))

instance Monoid UnitaryC where
        mempty = U (\ fv bs -> bs)
        mappend = (<>)

Qio.hs:
instance Semigroup Unitary where
        (U f) <> (U g) = U (\ fv h -> unEmbed $ do h' <- Embed $ f fv h
                                                   h'' <- Embed $ g fv h'
                                                   return h'')

instance Monoid Unitary where
        mempty = U (\ fv h -> unEmbed $ return h)
        mappend = (<>)
```

5.2.3 Testing

After updating QioClass.hs and Qio.hs, the package builds and installs successfully onto Haskell v8.4.3. *Note: The package still installs and runs correctly on the newest version of Haskell, namely v8.6.3.* Running various quantum programs using the updated package gives correct results, and so the package still works as intended after being updated, and so all that is left to do is change the base dependancy of the package to ≥ 4.11.

6 Algorithms

During the following two sections, the algorithms mentioned earlier will be explained in more detail, along with explanations of their implementations in Haskell using the QIO package. The algorithms are each broken down into parts, and each respective part of the implementation is described simultaneously in an effort to maximise understanding.

6.1 Grover's Algorithm

As stated earlier in section 4.2.1, Grover's algorithm can be used to search a quantum database in $O(\sqrt{n})$, and can be extended to solve any blackbox boolean function. This extension reveals the numerous uses Grover's algorithm has, and so will be the main focus for implementation. Allowing the user to input a custom function will abstract them away from the algorithm's complexity, enabling simple testing over multiple use cases - hopefully improving understanding.

In order to find a solution to a blackbox function - which returns 1 for a solution input, and 0 otherwise - a classical computer must repeatedly apply the function to random integers in order to find the solution, and so is $O(N)$. However, a quantum computer can find a solution to the function after applying it no greater than $\pi/4\sqrt{N/s}$ times - where N is the amount of variables in the function, and s is the amount of solutions to the function - with a probability of success close to 1 when N is large. While the algorithm works for all functions which produce either 0 or 1 based on the input, it is often still thought as just an algorithm which searches through an unsorted database faster than any classical method.

Put simply, Grover's algorithm takes in a blackbox function, creates a superposition over all possible function inputs and then emphasizes states which are not solutions. After doing this roughly $\pi/4\sqrt{N/s}$ times, there is a high probability that a solution state is found. The algorithm can be split up into multiple explicit parts:

1. **The Oracle** - Creating a unitary function based on the blackbox function, and applying it to a set of qubits.
2. **Diffusion** - Increasing the amplitude of solutional state(s).
3. **Measurement** - Measuring specific qubits to get a solution.

6.1.1 The Oracle

The blackbox function f returns 0 or 1 based on if the inputted value (i.e. an n-bit integer) x is the solution a;

$$f(x) = 0, \quad x \neq a \qquad f(x) = 1, \quad x = a.$$

This function can be represented in the form of a unitary function U_f, which acts on an n-qubit register containing x, and a single qubit register containing the result $f(x)$.

$$U_f(|x\rangle_n |y\rangle) = |x\rangle_n |y \oplus f(x)\rangle$$

A simple example of this would be a unitary function which returns 1 when 0010 is inputted, and 0 otherwise:

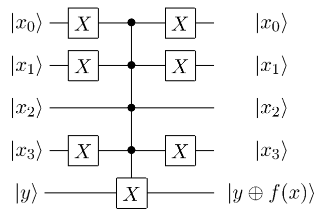


Figure 6: A unitary function, setting the last qubit to 1 when 0010 is inputted, and 0 otherwise

This can be modified so that the overall state's sign is changed if $x = a$ by first setting the output qubit to $|1\rangle$ and applying the Hadamard gate to it before the application of U_f. So, the output qubit is in the state $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$ before application, and after applying U_f the overall state becomes

$$U_f(|x\rangle \oplus |-\rangle) = (-1)^{f(x)}|x\rangle \oplus |-\rangle$$

From this equation it is obvious that applying U_f to the state is the same as doing nothing to the single output qubit, while applying a unitary transformation V to the input register, where

$$V|x\rangle = (-1)^{f(x)}|x\rangle = \begin{cases} |x\rangle, & x \neq a, \\ -|a\rangle, & x = a. \end{cases} \tag{1}$$

If we initially transform the n-qubit input register into a uniform superposition of all possible inputs, i.e.

$$|\phi\rangle = H^{\otimes n}|0\rangle_n = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n$$

Then, after applying V, the component of the state along $|a\rangle$ (i.e. the state we are looking for) will have a negative phase. The unitary V can therefore be written as

$$V|\Psi\rangle = |\Psi\rangle - 2|a\rangle\langle a|\Psi\rangle \qquad \therefore V = 1 - 2|a\rangle\langle a|$$

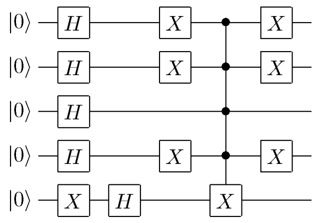


Figure 7: Applying V to an equal superposition of all possible inputs.

Implementation Before actually implementing the oracle used for Grover's algorithm in Haskell, it is useful to first define a few functions to work over lists of qubits - allowing easy expansion to custom oracles later on.

```
mkQbits :: Int -> Bool -> QIO [Qbit]}
mkQbits n b = mkQbits' n b []
  where
    mkQbits' 0 _ qs = return qs
    mkQbits' n b qs = do
      q <- mkQbit b
      mkQbits' (n-1) b (qs ++ [q])
```

As you can see, `mkQbits` is a function which uses recursion to create a list of `n Qbits` in the state given by `b`. This leads on to the creation of another function, `measQbits :: [Qbit] -> QIO [Bool]` which returns a list of boolean values representing the measurements of the inputted list of `Qbits`. The only two remaining functions necessary pertain to applying unitary functions to lists of `Qbits`. More specifically, we need a function which applies a single-qubit unitary operation to all qubits in a list, and another function which applies a single-qubit unitary operation to the last qubit in a list, conditional on the state of each other qubit in said list.

```
unitaryN :: (Qbit -> U) -> [Qbit] -> U
unitaryN _ []       = mempty
unitaryN uf (q:qs) = uf q <> unitaryN uf qs

condN :: (Qbit -> U) -> [Qbit] -> U
condN uf (q:[]) = uf q
condN uf (q:qs) = cond q (\x -> if x then (condN uf qs) else mempty)
```

Now that these functions are created, initializing the qubits and applying the oracle is simple. First, we need to create i qubits for each variable in the function and put them in the state $|+\rangle$, and then create o qubits for each "line" in the function, each put in the state $|-\rangle$.

```
initialize :: Int -> Int -> QIO ([Qbit], [Qbit])
initialize i o = do
  qI <- mkQbits i False
  qO <- mkQbits o True
  applyU (unitaryN uhad (qI ++ qO))
  return (qI, qO)
```

Now we need to create a function representing V, the oracle unitary. For now, this will be for the example above, searching for the state $|110\rangle$.

```
oracle :: [Qbit] -> [Qbit] -> U
oracle qI qO =
  unot (qI!!2) <>
  condN unot (qI ++ [qO!!0]) <>
  unot (qI!!2)
```

6.1.2 Diffusion

Grover's algorithm requires one more unitary W, which does not depend on the oracle function. This unitary transformation changes the sign of the component orthogonal to $|\phi\rangle$, i.e.

$$W = 2\,|\phi\rangle\,\langle\phi| - 1$$

The unitary $-W$ works fine here, as the final state will only differ by an overall minus sign, if it differs at all. From $-W$, and the fact that the Hadamard gate is its own inverse, you can see that we need a gate which does nothing for all states apart from $|00...00\rangle$, which it multiplies by -1.

$$-W = 1 - 2\,|\phi\rangle\,\langle\phi|\,, \qquad |\phi\rangle = H^{\otimes n}\,|0\rangle_n$$

$$\therefore -W = H^{\otimes n}(1 - 2\,|00...00\rangle\,\langle 00...00|)H^{\otimes n}$$

The Pauli-Z gate is the main gate to use here. This gate, with matrix $\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$, multiplies the state $|11...11\rangle$ by -1, and does nothing on all other states. From this, it is easy to see that if we applied a controlled Pauli-Z gate to the qubits, surrounded by n Pauli-X gates, we get the unitary $-W$.

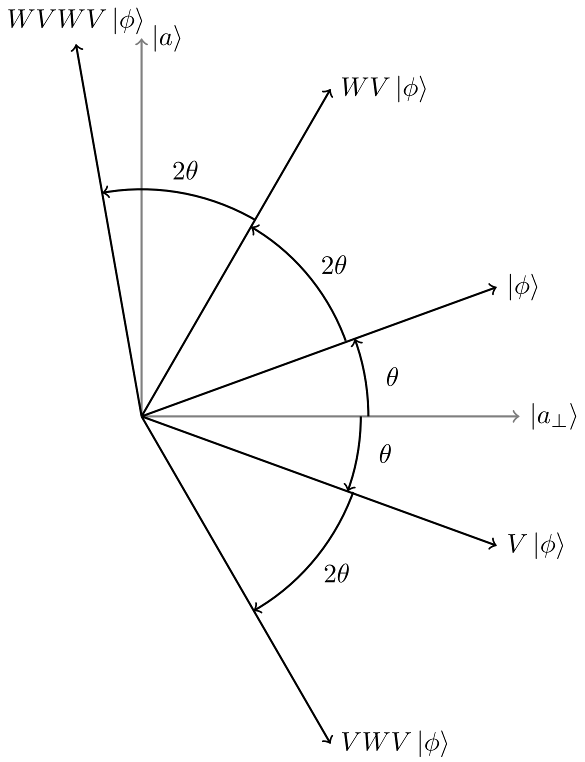
$$-W = H^{\otimes n}X^{\otimes n}(c^{n-1}Z)X^{\otimes n}H^{\otimes n}$$

Implementation Implementing this is extremely simple using the functions created earlier, as shown below

```
diffuse :: [Qbit] -> U
diffuse qI =
  unitaryN uhad qI <>
  unitaryN unot qI <>
  condN (\q -> uphase q pi) qI <>
  unitaryN unot qI <>
  unitaryN uhad qI
```

6.1.3 Application & Measurement

All that is left now is to apply WV repeatedly to the intital state $|\phi\rangle$ enough times for a solution state to be found. This is best described using simple geometry.



The intial state $|\phi\rangle$ will contain one state $|a\rangle$ with amplitude $\frac{1}{2^{n/2}} = 1/\sqrt{N}$, and so starts out at an angle θ from $|a_\perp\rangle$, where $\sin\theta = 1/\sqrt{N}$.

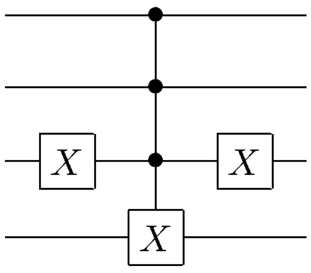
The unitary V negates the state $|a\rangle$ in $|\phi\rangle$, and so on the graph, it reflects $|\phi\rangle$ about the axis $|a_\perp\rangle$. Next, the unitary W is applied which negates any state orthogonal to $|\phi\rangle$ (i.e. any negative state in $|\phi\rangle$, notably the state $|a\rangle$). Graphically, this means the unitary W reflects about $|\phi\rangle$, and so when applied to $V\,|\phi\rangle$, it results in a state at an angle of 3θ above $|a_\perp\rangle$. Therefore, each iteration (application of the unitary V followed by W) rotates the qubits' state by 2θ anti-clockwise, giving us a simple method to find out the amount of iterations, i, required. Note: when N is large, $\sin\theta \approx \theta$.

$$\theta + 2i\theta \approx \pi/2 \qquad \sin\theta = 1/\sqrt{N} \qquad \therefore \theta \approx 1/\sqrt{N}$$

$$\therefore i \approx \frac{\pi - 2\theta}{4\theta} \approx \frac{\pi}{4}\sqrt{N}$$

The amplitude of the state $|a\rangle$ over the qubits is given by the sine of the angle between the qubits and $|a_\perp\rangle$, and so the probability of measuring the solution from the qubits is equal to the square of this.

Example Say we want to search for $|110\rangle$ across a space of 8 (i.e. 3 qubits). The iterations required will therefore be $\approx \frac{\pi}{4}\sqrt{8} \approx 2$. The oracle function will perform a controlled-X gate on its output qubit if its inputs are in the state $|110\rangle$. Then, we need to diffuse the qubits using the unitary W described above. After initializing the qubits, we must apply V, then W, and repeat this again to get a solution.



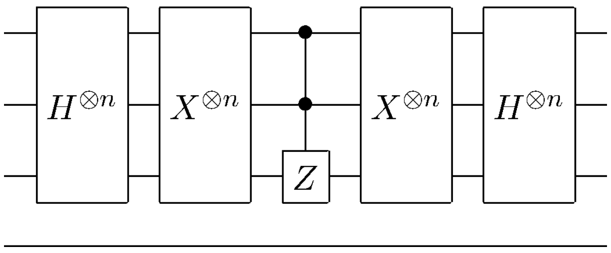


Figure 8: The 2 unitaries V (left) and W (right) used to search for $|110\rangle$.

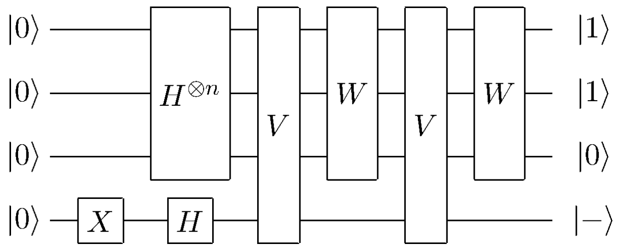


Figure 9: The circuit used to perform Grover's algorithm over 3 input qubits, with solution $|110\rangle$.

Implementation All that remains in the implementation is to combine the functions described above and apply them the correct amount of times (i.e. twice when searching for the state $|110\rangle$).

```
grover :: QIO [Bool]
grover = do
  (qI, qO) <- initialize 3 1
  let groverStep = oracle qI qO <> diffuse qI
  applyU (groverStep)
  applyU (groverStep)
  measQbits qI
```

Testing Testing this is relatively simple: simply `run` the `grover` function, check the result and then `sim` the function, checking the probability of expected results.

```
run grover -> [True, True, False]
sim grover -> [..., ([True, True, False], 0.9453125), ...]
```

with all other states having probability 7.812...e^{-3}

6.1.4 Expansion to multiple solutions

The algorithm above can also search for multiple solutions, the only thing that changes is the amount of iterations necessary. If there are s solutions, then the amount of iterations required becomes $i \approx \frac{\pi}{4}\sqrt{\frac{N}{s}}$.

6.1.5 Implementing a Custom Oracle

In the example above, the oracle is searching for `True, True, False`, i.e. $x1 \land x2 \land \neg x3$. So, if the program takes in a function as a boolean predicate, it can create a unitary function representing the oracle using a list of the input variables in the predicate, and applying 2 NOT gates to negated variables.

The oracle can also search for multiple lines of predicates separated by OR operations; simply have a different output qubit for each line, so that solutions to every line will be negated, allowing easy usage for multiple solutions. For example, if the user wants to apply the algorithm over a function f, where

$$f(x_n) = (x1 \land x2 \land x3) \lor (\neg x1 \land x2 \land \neg x3)$$

then they can input

```
 1 2 3
-1 2 -3
```

So the oracle function needs to be changed to take in a list of a list of `Ints`, i.e. `[[Int]]`, where each inner list represents a line of the function. From this, it will recursively apply the NOT gates over qubits referenced by negative integers, and apply controlled-X gates over the qubits referenced in each line to the output qubits, where each inner list is applied to its own unique output qubit.

```haskell
oracle :: Int -> [[Int]] -> [Qbit] -> [Qbit] -> U
oracle _ [] _ _ = mempty
oracle c (p:ps) qI qO =
  unitaryN unot (unots p qI) <>
  condN unot (conds p qI ++ [qO!!c]) <>
  unitaryN unot (unots p qI) <>
  oracle (c+1) ps qI qO
  where
    unots p qI = map (\x -> qI!!((abs x)-1)) $ filter (< 0) p
    conds p qI = map (\x -> qI!!((abs x)-1)) p
```

The program also needs to know how many solutions there are in order to iterate the application of V and W the correct amount of times, roughly equal to $\frac{\pi}{4}\sqrt{\frac{N}{s}}$. The application can then be done using the `foldr` function, folding with the `mappend` operator `<>`, starting from `mempty`.

The entry point to the algorithm will be using the `main` function in the code. This function, of type `IO ()`, will take in a custom function inputted in the form described above, and then run the algorithm over it. This can also be timed effectively using the `TimeIt` package [Aug09].

6.1.6 Testing

In order to test this, the program needs to be changed slightly to `sim` the algorithm, and print out the probabilities. The above predicate $x1 \land x2 \land \neg x3$, inputted as `1 2 -3` returns the expected result:

```
[...,([True, True, False]), 0.9453125),...]
```

with all other states having probability $7.812...e^{-3}$

A more complicated predicate shown below also returns the correct result, indicating the implementation works correctly.

$$(x1 \land \neg x2 \land x3 \land x4) \lor (x2 \land \neg x3 \land x4) \lor (x1 \land x2 \land x3)$$

Solutions: $0101, 1011, 1101, 1110, 1111$

Output:

```
[([True, True, True, True], 0.1914...),
 ([True, True, True, False], 0.1914...),
 ([True, True, False, True], 0.1914...),
 ...,
 ([True, False, True, True], 0.1914...),
 ...,
 ([False, True, False, True], 0.1914...),
...]
```

with all other states having probability $3.906...e^{-3}$

6.1.7 An Improvement

This algorithm can be altered to actually not use any ancilla qubits in QIO (due to the way QIO executes multi-qubit gates) and use less gates, improving run time. The current implementation uses ancilla qubits to negate solution states in the input qubits. However, this can simply be done using a controlled Pauli-Z gate over the inputs, when the input qubits are initially in the state $|-\rangle$. After this, the diffusion is also simplified to become an inversion about the mean [Gid13]; decrease the amplitude of non-solutional states about the mean amplitude while increasing the amplitude of solutional states. This can be achieved by $n-1$ Hadamard gates surrounding a C^n-NOT gate.

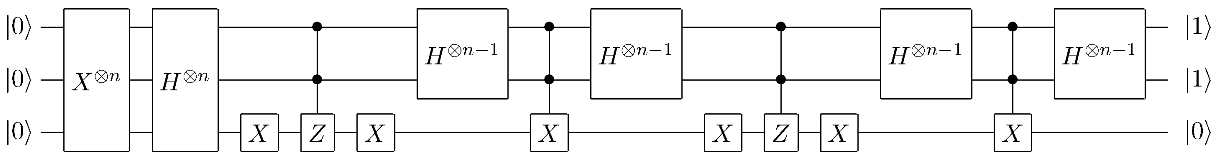


Figure 10: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$

In order to change the code to implement this, the `initialize`, `oracle` and `diffuse` functions need to be altered.

6.1.8 Testing Solutions

The current algorithm works well when solving a function with a known amount of solutions, and returns correct solutions with a high probability as shown. However, the implementation can be changed to measure whether or not the input qubits contain a solution at the end, This will enable the program to return definite, distinct solutions and find solutions to functions with an unknown amount of solutions.

By simply using ancilla qubits to test the state of input qubits with C^n-NOT gates, a final ancilla qubit will be in the state $|1\rangle$ if the input qubit's contain a solution.

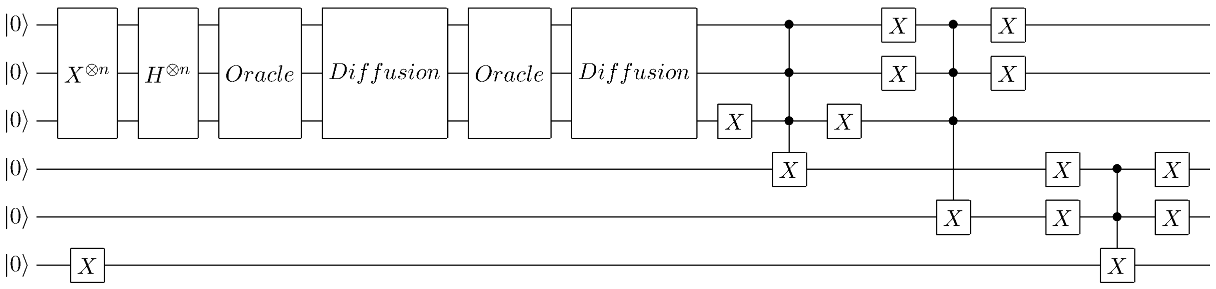


Figure 11: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$ and $|001\rangle$, while also checking n qubits for the solution.

The results of each line of the function (i.e. $x0 \wedge x1 \wedge \neg x2$ and $\neg x0 \wedge \neg x1 \wedge x2$) are saved into two separate qubits. Following this, a third ancilla qubit is set to $|0\rangle$ if both the two ancilla qubits are in the state $|0\rangle$, which would only occur when the input qubits do not contain a solution.

Implementation In order to implement this, the `initialize` function will need create the necessary ancilla qubits, all of which will be in the state $|0\rangle$ except for the last, which will be in the state $|1\rangle$. A new function needs to be created to apply the necessary C^n-NOT gates over the input qubits on distinct ancilla qubits, which will be applied after the algorithm has been executed. Finally, the program will measure the last ancilla qubit in the list, and return this with the measurements of the input qubits. The ancilla measurement can then be checked to determine if a solution was found, allowing easy looping to get multiple, unique and definite solutions.

This will also enable the algorithm to be used for functions with an unknown amount of solutions, as this requires multiple executions of the algorithm with differing amounts of iterations until a solution is found. In this case, the amount of iterations starts as $\frac{\pi}{4}\sqrt{N}$. If this does not result in a solution, the algorithm is executed again with $\frac{\pi}{4}\sqrt{\frac{N}{2}}$ iterations, and then $\frac{\pi}{4}\sqrt{\frac{N}{4}}$ iterations, and so on until a solution is found. The total amount of iterations required is still $O(\sqrt{N})$ as shown below.

$$\frac{\pi}{4}\sqrt{N}\left(1+\frac{1}{\sqrt{2}}+\frac{1}{2}+\frac{1}{2\sqrt{2}}+...\right)=\frac{\pi}{4}\sqrt{N}\sum_{i=0}^{\infty}\frac{1}{2^i}=\frac{\pi}{4}\sqrt{N}(2+\sqrt{2})$$

6.1.9 Probability

We can also print out the probability of finding the solution by running the `sim` function over a function which only measures the last ancilla qubit (which will only be measured to be True if the input qubits contain a solution).

```
probability :: Int -> Int -> [[Int]] -> IO ()
probability n i ps = putStrLn $ show $ sim $ probOfSolution
  where
    probOfSolution = do
      (qI, qA) <- initialize n ((length ps)+1)
      let groverStep = oracle ps qI <> diffuse qI
      applyU (foldr (<>) mempty (replicate i groverStep))
      applyU (testForSolution 0 ps qI qA)
      measQbit $ last qA
```

6.2 Shor's Algorithm

As mentioned in section 4.1, there is already an implementation of Shor's algorithm in QIO, in the file `Shor.hs`. Therefore, this section will be dedicated to explaining the current implementation, and describing any potential improvements.

In section 4.2.2, it was stated that Shor's algorithm is for integer factorization. This is not exactly correct, as the algorithm actually finds the period of a function, which can then be expanded to factorize integers. Let f be a function, periodic on addition, over integers, i.e.

$$f(x) \equiv f(y) \iff x = y + cr, \qquad c \in \mathbb{Z}$$

Classical algorithms for finding the period of such a function take a time which grows faster than any power of the number of bits of r. However, Shor's algorithm does this in a time which scales slightly faster than n^3 [Mer07]$^{\text{Page 63}}$.

Put simply, the algorithm applies the function f over two qubit registers. Measuring the second register, i.e. the register containing the results of $f(x)$, collapses the first into a superposition over all the values which give that measured value. Each of these values in the register will be separated by integer multiples of the period r. Finally, we apply an algorithm known as the quantum Fourier transformation, which extracts the value of the period r from the register. We can then measure this register to obtain a value for the period. Similar to Grover's algorithm, it can be broken down into distinct parts:

1. **Function Application** - Creating a unitary representation for the function f, and applying it to the two qubit registers.
2. **QFT** - Applying the quantum Fourier transformation to the first register after measuring the second.
3. **Period finding** - Measuring the second register and calculating a value for the period r.

In order expand this to factorize an integer $N = pq$, where p and q are 2 prime numbers, the function to apply is

$$f(x) = b^x \pmod{N}, \text{ where } b \text{ is coprime to } N$$

The required amount of qubits in each register is equal to the number of bits in N, namely n_0. However, to efficiently find the period r, the input register needs to actually have twice this amount, ensuring the range of values for x on which $f(x)$ is calculated has at least N full periods of f[Mer07]$^{\text{Page 69}}$. After applying Shor's algorithm with the above function, the period r needs to have certain properties

1. r needs to be even so we can calculate
$$x = b^{\frac{r}{2}} \pmod{N}$$
2. $x - 1 \neq 0 \pmod{N}$ as r must be the smallest integer for which $b^r = 1 \pmod{N}$

From this,

$$x^2 = (b^{\frac{r}{2}})^2 \pmod{N} = b^r \pmod{N} = 1 \pmod{N}$$

$$\therefore (x-1)(x+1) = x^2 - 1 = 0 \pmod{N}$$

So now, $(x-1)$ and $(x+1)$ aren't divisible by N, but $(x-1)(x+1)$ is. As N is the product of 2 primes, p and q, it follows that $(x-1)$ is divisble by p, and $(x+1)$ is divisble by q. Therefore, $p = gcd(x-1, N)$ and $q = gcd(x+1, N)$.

6.2.1 Function Application

Calculating $\mathbf{U_f}$ So, we need a unitary U_f which calculates $f(x) = b^x \pmod N$, i.e.

$$U_f \left|x\right\rangle \otimes \left|y\right\rangle = \left|x\right\rangle \otimes \left|y \oplus f(x)\right\rangle$$

Doing this is actually fairly simple: square $b \pmod N$, then square the result $\pmod N$, then square that etc. calculating the number of powers $b^{2^j} \pmod N$ where $j < n$. So, if initially x is in the input register, 1 (i.e. 00..01) is in the output register, and b is in an additional work register, then the process is simple[Mer07]$^{\text{Page } 84}$:

1. Multiply the output register by the work register using modular multiplication if $x_0 = 1$,
2. Replace the contents of the work register by its modulo-N square,
3. Repeat step 1, now conditional on $x_1 = 1$,
4. Repeat step 2,
5. Repeat step 1, now conditional on $x_2 = 1$, etc.

At the end of this, x will still be in the input register, and $b^x \pmod N$ will be in the output register.

The current implementation of this, found in `QArith.hs`, uses a combination of modular addition operations to form modular multiplication operations, which in turn are combined together to form modular exponentiation operations, calculating $f(x)$. This is explained in the G53/G54NSC "Shor's Algorithm continued..." lecture [Gre09].

Applying $\mathbf{U_f}$ Applying the above unitary to the state $\left|\phi\right\rangle_n \otimes \left|0\right\rangle_{n_0}$, where

$$\left|\phi\right\rangle_n = H^{\otimes n}\left|0\right\rangle = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} \left|x\right\rangle$$

results in the state

$$\frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} \left|x\right\rangle_n \left|f(x)\right\rangle_{n_0}$$

If we then measure the output register to be f_0, the state of the input register will collapse to containing all of the values which give the same value for $f(x)$, i.e.

$$\left|\Psi\right\rangle_n = \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} \left|x_0 + kr\right\rangle_n, \qquad 0 \le x_0 < r, \qquad f(x_0) = f_0$$

where m is the smallest integer where $mr + x_0 \ge 2^n$, and so

$$m = \left[\frac{2^n}{r}\right] \text{ or } m = \left[\frac{2^n}{r}\right] + 1, \text{ where } \left[\frac{2^n}{r}\right] \text{ is the largest integer less than or equal to } \frac{2^n}{r}$$

Measuring this register right now would give us a value for $x_0 + kr$, where x_0 is unknown and random, and so we cannot calculate the value of r yet. We first need to remove the x_0 from the state using the quantum Fourier transform.

6.2.2 The Quantum Fourier Transform

Periodic functions can be decomposed into the sum of simple sine and cosine functions, and this decomposition is calculated by the Fourier transform. The discrete Fourier transform from the vector $x_0, x_1, ..., x_{2^n-1}$ to the vector $y_0, y_1, ..., y_{2^n-1}$ is defined by

$$y_a = \frac{1}{\sqrt{2^n}} \sum_{b=0}^{2^n-1} e^{2\pi iab/2^n} x_b$$

This can be used to our advantage. In order to remove the x_0 value from the state described above, the quantum Fourier transform is used to turn it into a harmless overall phase factor, allowing us to get the value for the period r. The equation for the quantum Fourier transform unitary U_{FT} is shown below

$$U_{FT} \left|x\right\rangle_n = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi ixy/2^n} \left|y\right\rangle_n$$

When applied to a superposition of states $|x\rangle$ with complex amplitudes $\gamma(x)$, U_{FT} produces another superposition with amplitudes related to $\gamma(x)$ by the appropriate discrete Fourier transform

$$U_{FT}\left(\sum_{x=0}^{2^n-1} \gamma(x) \left|x\right\rangle\right) = \sum_{x=0}^{2^n-1} \tilde{\gamma}(x), \left|x\right\rangle \qquad \text{where } \tilde{\gamma}(x) = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi ixy/2^n} \gamma(y)$$

Now, in order to construct the quantum Fourier transform circuit, it is best to define a unitary U_Z, which can be seen as a generalization of the Pauli-Z gate to n qubits, where

$$U_Z \left|y\right\rangle_n = e^{2\pi iy/2^n} \left|y\right\rangle_n$$

From this, it is clear that

$$U_Z^x H^{\otimes n} \left|0\right\rangle_n = U_Z^x \left(\frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \left|y\right\rangle_n\right) = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi ixy/2^n} \left|y\right\rangle_n = U_{FT} \left|x\right\rangle_n$$

In order to make the construction even easier, we can look at a specific case, say $n = 4$, and then generalize for any value of n. So, $n = 4$ means

$$U_{FT} \left|x_3\right\rangle \left|x_2\right\rangle \left|x_1\right\rangle \left|x_0\right\rangle = U_Z^x H_3 H_2 H_1 H_0 \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle$$

If $\left|y\right\rangle_4 = \left|y_3\right\rangle \left|y_2\right\rangle \left|y_1\right\rangle \left|y_0\right\rangle$, then $y = 8y_3 + 4y_2 + 2y_1 + y_0$. Now we can construct U_Z using its definition above.

$$U_Z y = e^{\pi i/8}(8y_3 + 4y_2 + 2y_1 + y_0)$$
$$\therefore U_Z = \exp\left(\frac{\pi i}{8}(8\mathrm{n}_3 + 4\mathrm{n}_2 + 2\mathrm{n}_1 + \mathrm{n}_0)\right)$$
$$\therefore U_Z^x = \exp\left(\frac{\pi i}{8}(8x_3 + 4x_2 + 2x_1 + x_0)(8\mathrm{n}_3 + 4\mathrm{n}_2 + 2\mathrm{n}_1 + \mathrm{n}_0)\right)$$

We can simplify this, combining the coefficients for each n.

$$\begin{aligned} U_Z^x = \exp(\pi i(&\mathrm{n}_3(8x_3 + 4x_2 + 2x_1 + x_0)+ \\ &\frac{1}{2}\mathrm{n}_2(8x_3 + 4x_2 + 2x_1 + x_0)+ \\ &\frac{1}{4}\mathrm{n}_1(8x_3 + 4x_2 + 2x_1 + x_0)+ \\ &\frac{1}{8}\mathrm{n}_0(8x_3 + 4x_2 + 2x_1 + x_0)) \end{aligned}$$

Now, the unitary operation $\exp(2\pi i \mathrm{n})$ is the same as a two Pauli-Z gates, and so is simply an identity operator. This means any $\exp(2^p \pi i \mathrm{n})$ can be removed, allowing us to further simplify U_Z^x

$$U_Z^x = \exp\left(\pi i (x_0 \mathrm{n}_3 + (x_1 + \frac{1}{2}x_0)\mathrm{n}_2 + (x_2 + \frac{1}{2}x_1 + \frac{1}{4}x_0)\mathrm{n}_1 + (x_3 + \frac{1}{2}x_2 + \frac{1}{4}x_1 + \frac{1}{8}x_0)\mathrm{n}_0\right)$$

Following this, we can simply the equation for $U_Z^x H^{\otimes x} |0\rangle_n$. The operators $\exp(\pi i x \mathrm{n})$ and $H|0\rangle$ obey the relation $\exp(\pi i x \mathrm{n})H|0\rangle = H|x\rangle$ over single qubits. For now, lets ignore the terms with fractional coefficients.

$$\begin{aligned}
&\exp(\pi i (x_0 \mathrm{n}_3 + x_1 \mathrm{n}_2 + x_2 \mathrm{n}_1 + x_3 \mathrm{n}_0) H_3 H_2 H_1 H_0 |0\rangle |0\rangle |0\rangle |0\rangle \\
&\quad = [\exp(\pi i x_0 \mathrm{n}_3) H_3 |0\rangle]\ [\exp(\pi i x_1 \mathrm{n}_2) H_2 |0\rangle]\ [\exp(\pi i x_2 \mathrm{n}_1) H_1 |0\rangle]\ [\exp(\pi i x_3 \mathrm{n}_0) H_0 |0\rangle] \\
&\quad = [H_3 |x_0\rangle]\ [H_2 |x_1\rangle]\ [H_1 |x_2\rangle]\ [H_0 |x_3\rangle] \\
&\quad = H_3 H_2 H_1 H_0 |x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle
\end{aligned}$$

When we add in and group the remaining terms we are left with the following

$$\begin{aligned}
U_{FT} |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle = &H_3 \exp\left[\pi i \mathrm{n}_2 \frac{1}{2} x_0\right] H_2 \exp\left[\pi i \mathrm{n}_1 \left(\frac{1}{2}x_1 + \frac{1}{4}x_0\right)\right] H_1 \\
&\times \exp\left[\pi i \mathrm{n}_0 \left(\frac{1}{2}x_2 + \frac{1}{4}x_1 + \frac{1}{8}x_0\right)\right] H_0 \\
&\times |x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle
\end{aligned}$$

The state $|x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle$ is an eigenstate of the number operators n_3, n_2, n_1, n_0 with eigenvalues x_0, x_1, x_2, x_3, allowing us to simplify further.

$$\begin{aligned}
U_{FT} |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle = &H_3 \exp\left[\pi i \frac{1}{2} \mathrm{n}_2 \mathrm{n}_3\right] H_2 \exp\left[\pi i \mathrm{n}_1 \left(\frac{1}{2}\mathrm{n}_2 + \frac{1}{4}\mathrm{n}_3\right)\right] H_1 \\
&\times \exp\left[\pi i \mathrm{n}_0 \left(\frac{1}{2}\mathrm{n}_1 + \frac{1}{4}\mathrm{n}_2 + \frac{1}{8}\mathrm{n}_3\right)\right] H_0 \\
&\times |x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle
\end{aligned}$$

After defining a new unitary operator for the 2-qubit operators, this becomes a lot clearer.

$$V_{ij} = \exp(\pi i \mathrm{n}_i \mathrm{n}_j / 2^{|i-j|})$$

$$\begin{aligned}
\therefore\ &U_{FT} |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle \\
&= H_3 (V_{32} H_2)(V_{31} V_{21} H_1)(V_{30} V_{20} V_{10} H_0) |x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle
\end{aligned}$$

It is obvious now that V_{ij} simply rotates qubit x_j by $e^{\pi i / 2^{|i-j|}}$ when the qubit x_i is in the state $|1\rangle$, allowing us to define a simple rotation operator.

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\pi i / 2^k} \end{bmatrix}, \text{ where } k = |i - j|$$

All that is left is to define a unitary S, where $S |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle = |x_0\rangle |x_1\rangle |x_2\rangle |x_3\rangle$ (made out of multiple SWAP operations), and we have a definition for U_{FT} using 1 and 2 qubit gates.

$$U_{FT} = H_3 (R_1 H_2)(R_2 R_1 H_1)(R_3 R_2 R_1 H_0) S$$

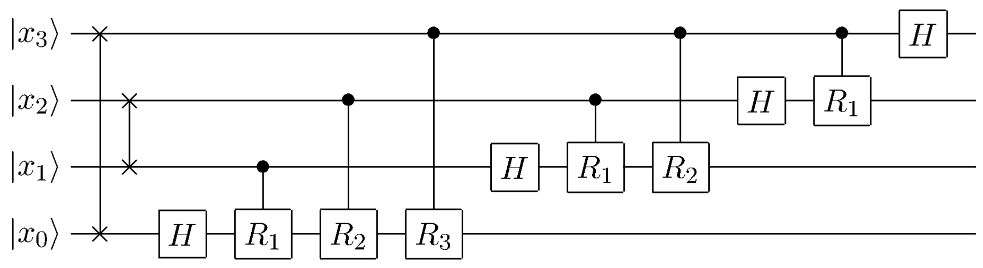


Figure 12: The circuit diagram for the quantum Fourier transform operation over 4 qubits

Implementation The implementation of the quantum Fourier transform in QIO is actually slightly different. By reversing the order of the controlled rotations, and the overall order of the gates, we can remove the need for SWAP operations at the start.

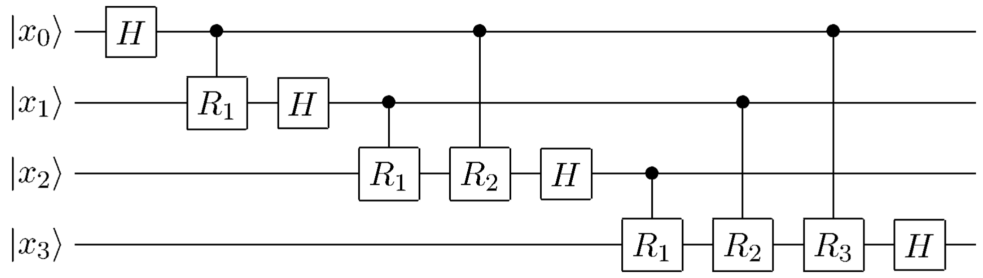


Figure 13: The circuit diagram for an alternate quantum Fourier transform operation over 4 qubits

So, given a list of `Qbits`, we need to apply a Hadamard gate, and then multiple rotation gates for each remaining `Qbit` in the list, where each gate is conditional on a distinct `Qbit`. I.e. given a list of 4 `Qbits`, the unitary to construct is shown below, where `R k xn` is a R_k gate on qubit x_n.

```
uhad x0 <>
cond x0 (\x -> if x then R 1 x1 else mempty) <> uhad x1 <>
cond x1 (\x -> if x then R 1 x2) <> cond x0 (\x -> if x then R 2 x2) <> uhad x2 <>
cond x2 (\x -> if x then R 1 x3) <> cond x1 (\x -> if x then R 2 x3) <>
cond x0 (\x -> if x then R 3 x3) <> uhad x3
```

In order to create such a unitary operation, the implementation uses two recursive functions, `qftAcu` and `qftBase`. An entry function `qft` is then used to apply `qftAcu` over a list of `Qbits`, using a new type class `Qdata a qa | a -> qa, qa -> a`.

This class defines what operations a quantum datatype must implement, and so relates heavily to the main structure of the `QIO` monad.

```
class Qdata a qa | a -> qa, qa -> a where
  mkQ :: a -> QIO qa
  measQ :: qa -> QIO a
  letU :: a -> (qa -> U) -> U
  condQ :: qa -> (a -> U) -> U
```

```
instance Qdata Bool Qbit where
    mkQ = mkQbit
    measQ = measQbit
    letU b xu = ulet b xu
    condQ q br = cond q br
```

As you can see, we have already defined the operations for the relation between `Bools` and `Qbits`.

The class is then used to define other quantum datatypes; pairs of datatypes, lists of datatypes, and most notably, a datatype relating `Ints` to quantum integers, `QInts`.

Using two functions - one which creates a list of `Bool` values representing its `Int` argument, and the other which converts a list of `Bool` values to an integer - the datatype effectively works as an abstraction over a list of `Qbits` qs, stored as `QInt qs`.

The main operation used for the quantum Fourier transform is `condQ`. It is used to perform conditional operations, such as the rotations described above, and enables the function `qft` to send its `[Qbit]` argument to `qftAcu` with the measured values of the qubits.

```
qft :: [Qbit] -> U
qft qs = condQ qs (\bs -> qftAcu qs bs [])
```

From there, `qftAcu` deals with all the operations on a qubit, recursively moving from x_0 to x_{n-1}, while `qftBase` deals with each individual operation, i.e. the Hadamards and conditional Rotation gates.

```
qftAcu :: [Qbit] -> [Bool] -> [Bool] -> U
qftAcu [] [] _ = mempty
qftAcu (q:qs) (b:bs) cs = qftBase cs q ‘mappend‘ qftAcu qs bs (b:cs)

qftBase :: [Bool] -> Qbit -> U
qftBase bs q =  f’ bs q 2
    where f’ [] q _ = uhad q
          f’ (b:bs) q x = if b then (rotK x q) ‘mappend‘ f’ bs q (x+1)
                  else f’ bs q (x+1)
```

6.2.3 Finding the period

So, after applying the unitary U_f to the state $|\phi\rangle_n \otimes |0\rangle_{n_0}$, and measuring the output register, the input register collapses to the state

$$|\Psi\rangle_n = \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} |x_0 + kr\rangle_n, \qquad 0 \le x_0 < r, \qquad f(x_0) = f_0$$

To extract the period r, we can apply the quantum Fourier transformation as shown below.

$$\begin{aligned} U_{FT} |\Psi\rangle &= \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} e^{2\pi i (x_0+kr)y/2^n} |y\rangle \\ &= \sum_{y=0}^{2^n-1} e^{2\pi i x_0 y/2^n} \frac{1}{\sqrt{2^n m}} \left(\sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right) |y\rangle \end{aligned}$$

The probability of getting the result y is therefore given by

$$\frac{1}{2^n m} \left| \sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right|^2$$

which is highest when y is close to integral multiples of $2^n/r$, giving the period as $r = 2^n/y$.

6.2.4 The Implementation - an Alternative

The quantum Fourier transformation has another use here; it's inverse can be used in a procedure known as phase estimation:

If we have a superposition of states

$$\frac{1}{2^{t/2}}\sum_{j=0}^{2^t-1} e^{2\pi i\varphi j}\left|j\right\rangle_n \left|u\right\rangle_m$$

we can apply the inverse quantum Fourier transform to get the state $\left|\hat{\varphi}\right\rangle_n \left|u\right\rangle_m$ where $\hat{\varphi}$ is an approximation of φ up to n bits. We can use this phase estimation procedure to execute Shor's algorithm.

To recap, after applying U_f, we are left with the state

$$\frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}\left|x\right\rangle \left|b^x \pmod N\right\rangle$$

Now, this state is approximately equal to

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}\left(\frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1} e^{2\pi i x\frac{s}{r}}\left|x\right\rangle\left|u_s\right\rangle\right) \text{ where } \left|u_s\right\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{\frac{-2\pi i s k}{r}}\left|b^k \pmod N\right\rangle$$

Applying the inverse quantum Fourier transform to this state gives

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}\left|(\frac{\hat{s}}{r})\right\rangle\left|u_s\right\rangle$$

and so measuring the first register will give us an integer result $\hat{\varphi} = \frac{\hat{s}}{r} \approx \frac{s}{r}$ up to n bits. We can then use the continued fractions algorithm for $\frac{\varphi}{2^n}$ to calculate r.

On to the implementation. After getting a random integer coprime to N - using a quantum random number generator defined in `QIORandom.hs` - the implementation applies Shor's algorithm with the function `shor :: Int -> Int -> QIO Int`, which initializes the qubits, applies hadamard gates to the first register, applies the function $b^x \pmod N$ and then applies the inverse of the quantum Fourier transformation. This is done efficiently using the already-defined quantum Fourier transformation, `qft`, and the function which reverses its unitary argument, `urev`. Finally, the first register is measured and its value returned as an `Int`. Due to the approximation and the small amount of qubits, this integer returned is approximately the period r.

All that is left is to check if r is even and not equal to 0, and that $x - 1 \neq 0 \pmod N$. If all these are true, N is factorized using `gcd (x-1) n` and `gcd (x+1) n`.

7 Quantum Circuit Builder

7.1 Analysis

As mentioned earlier, the quantum circuit builder will be similar to the IBM Q Composer [ibmd]. Emphasis needs to be placed on usability, giving users a stable, reliable and most notably, easy to use graphical platform they can use to create and simulate quantum computations on a classical computer using Haskell and the QIO package.

Therefore, the user should be presented with a readable and intuitive circuit diagram; minimal in design to avoid confusion while also being large enough to convey the details of the circuit. The user needs to be able to alter the amount of qubits on the circuit. Quantum gates will be able to be moved on/off the circuit, enabling easy creation of a new quantum circuit without confusion. Buttons need to be available to run or simulate the created circuit, presenting the results of either to the user in a clear manner. The circuit needs to be able to be edited, allowing the moving of gates off the circuit, and changing the order in which they appear. Gates also need to be able to be made conditional to other specified qubits so that more complex quantum circuits can be built.

7.1.1 Specification

Functional

1. A graphical interface will be presented to the user in its own window upon running the program. Closing this window will end the program entirely.
2. An initial circuit will be shown containing three qubits, initially in the state $|0\rangle$.
3. Buttons will be available to add or remove qubits from the circuit, and all qubits will initially be in the state $|0\rangle$.
4. Quantum gates will be able to be moved on and off the circuit using a mouse.
5. While picked up, gates must be able to be removed, allowing the user to cancel their original intention.
6. Gates will be able to be made conditional on other qubits, which also need to be specified before placement.
7. Gates must be able to be moved around the circuit, while maintaining the layout of the other gates on the circuit.
8. Buttons can be used to run or simulate the circuit, using the appropriate `run` and `sim` functions. The results of these functions then need to be presented to the user clearly.

Non-functional

1. **Usability** Emphasis must be placed on usability. All aspects of the graphical interface must be large enough to read and interact with, while also being clear on their intended use. A user guide will need to be created, explaining in appropriate detail each part of the program, and explain appropriate use cases.
2. **Reliability** The program must be able to be run at any time on any appropriate computer with a Haskell compiler installed. It must not crash at any point, and so should only execute simple functions to present and allow interaction with the interface. However, the running and simulating of circuits will rely on QIO's functions, which are already relatively reliable.
3. **Performance** Due to its graphical nature, the interface needs to be efficient in its rendering and interaction functions in order to minimise user frustration and confusion.
4. **Quality** The functional and non-functional specifications here needs to be met as best as possible to maximise the program's quality. It must be thoroughly tested with both simple and complex quantum circuits to ensure it is robust and can handle the user's needs.
5. **Documentation** As mentioned above, a user guide will need to be created to provide instruction to the user on how to interact with the program. Simple explanation needs to be provided on what each aspect of the interface does, alongside the detailing of common use cases of the program.

7.2 Graphics in Haskell

As the circuit builder will be created in Haskell, a suitable graphics package must be used to deal with rendering as well as user interaction. Therefore, research is first done to find the most appropriate package to ensure program quality alongside an understandable implementation.

7.2.1 Diagrams

Diagrams [Yor16] is "a powerful, flexible, declarative domain-specific language for creating vector graphics, using the Haskell programming language." It's core library aims to be as simple and flexible as possible, built using a small set of primitive types and operations. The main page of the package [Yor16] provides access to a quick start tutorial, giving basic instruction on installation and simple use, along with a full user manual explaining each aspect of the package, and how to create complex images.

Drawing both simple and complicated images is very easy with this library. For example, to draw a circle of radius 1 is done quickly by executing `circle 1`. Shapes can be combined using functions such as `atop`, or placed side-by-side with the operators `|||` and `===` for horizontal and vertical placement respectively. Diagram attributes, such as line width and fill colour, are defined using simple functions taking in the value to set the attribute to, such as `lw veryThick` and `fc red`. Diagrams can also be transformed using `scale`, `rotate` and `reflect` functions. The diagrams form a monoid under `atop` as well, and so two diagrams can be superimposed using the mappend operator `<>` instead of `atop`. Other things, such as transformations and paths, also form monoids allowing easy construction of complex images.

While this package does seem promising, it is not suitable for creating a quantum circuit builder. The simple and effective commands would be extremely useful for rendering purposes, however the program does not need complex shapes to be drawn. The diagrams package also draws to files, such as .svg files using its SVG backend, which is the opposite of what is required by the circuit builder. User interaction is also not supported, reinforcing the idea that this package is not suitable for use in this case.

7.2.2 Gloss

Gloss [Lip18] "hides the pain of drawing simple vector graphics behind a nice data type and a few display functions". The package abstracts the use of OpenGL and GLUT, promising that you can "get something cool on the screen in under 10 minutes." Much like diagrams, it aims to be as simple as possible while also being flexible, allowing simple and complex images to be drawn effectively. Learning the package is not quite as easy to do as with diagrams, as there is no quick start tutorial or user guide. Instruction is instead given using straightforward examples, ranging from drawing simple text on the screen to animating the movement of complex images over a background. Luckily, the package is intuitively designed, and so being able to create images, interact with the user and draw animations, can be learnt with the use of these examples and reading through some of the code.

Much like with the diagrams package, shapes can be drawn extremely easily using functions like `rectangleSolid` and `thickCircle`. Text can also be drawn using a simple function `text` which takes in the `String` to draw. Everything that can be drawn is defined under a `Picture` data type, and attributes of shapes can be changed by functions of the type `Picture -> Picture`. This means such functions can be combined with the $ character. For example, creating a solid red circle with radius 5 at coordinates (100,100) is done with the line `translate 100 100 $ color red $ circle 5`. Pictures can be combined and altered as one using the `pictures :: [Picture] -> Picture` function.

The `main :: IO ()` function is used as the entry point to programs using gloss, which provides multiple functions for different use cases. For example, the `display :: Display -> Color -> Picture -> IO ()` function draws the `Picture` on a display specified by the `Display` type with a background colour specified by the `Color` type. The main function here though is the function `play`.

```
 play :: Display -> Color -> Int
         -> world -> (world -> Picture) -> (Event -> world -> world)
         -> (Float -> world -> world) -> IO ()
```

This function presents a display specified with a given background colour and a framerate. It then uses a function of type `(world -> Picture)` to render a world, starting with the intial world specified, and allows user interaction with a function of type `(Event -> world -> world)`. The final argument is then used to step the world by one frame, allowing animations to be created.

This package is a great fit for use in creating a graphical quantum circuit builder. The simple functions provided to draw basic shapes, which can be extended to draw more complex objects, work perfectly, and the availability of a function to draw text given a string would come in handy. The abstraction of user interaction with an `Event` type is great to create simple yet effective functions which handle all necessary types of user interaction. However, the lack of even a small quick start guide means the, albeit basic, knowledge required to use the package has to be learnt through limited examples, code reading and trial and error.

7.2.3 The Package of Choice

The most appropriate package for use in creating the quantum circuit builder is gloss. The simple abstraction to OpenGL and GLUT means drawing shapes is easily done, effective and efficient. The `play` function means handling of user interaction can be done with one simple function, and is made to be used with one world type in mind. This would enable the implementation to be built around a circuit record, storing the amount of qubits on the circuit, the gates on the circuit etc.

The `pictures` function will allow transformations to be performed on groups of images, such as all the buttons or the rows of qubits, making the implementation simple and easy to understand for others. Rendering of text will be used frequently to intuitively describe buttons and present results of computation to the user.

7.3 Implementation

7.3.1 The Window

To start off, the implementation requires a `main :: IO ()` function to display a window to the user, and provide the necessary functions to the `play` function mentioned above. The window will have the title “Circuit”, with a width of 1024 and height of 720, placed initially at the position (10,10) on the screen.

```
window :: Display
window = InWindow "Circuit" (floor width, floor height) (10,10)

main :: IO ()
main = play window
            white
            30
            initialCircuit
            draw
            handleEvent
            (const id)
```

This will produce a window described above, with a white background colour and an fps of 30. Given an initial circuit, it will draw using the `draw` function and handle user interaction via `handleEvent`. Finally, as animations are not necessary, the last function simply just returns whatever it is given.

Each aspect of the program, specifically the `main` function, the circuit, the rendering and the interaction will be located in its own file, ensuring simple modularity.

7.3.2 The Circuit

As explained above, gloss relies upon a single data type to store all the details of the "world", or in this case, the circuit. Therefore, the implementation requires a data type `Circuit`, which will be a record containing all the circuit details, including the qubits in the circuit and the gates, and their locations, on the circuit.

```
data Circuit = Circuit { qubits :: [Int],
                         gates :: [Gate]
                       } deriving (Eq, Show)

initialCircuit :: Circuit
initialCircuit = Circuit { qubits = [0..2],
                           gates = [] }

data Gate = Null,
          | Had Int [Int]
          | PX Int [Int]
          | PY Int [Int]
          | PZ Int [Int]
          | Swap Int [Int]
          deriving (Eq, Show)

instance Ord Gate where
  compare g1 g2 = compare (getCol g1) (getCol g2)
```

A datatype `Gate` is also created, restricting a Gate to be of a specific type, with a column `Int` and a list of `Int`s representing the qubits it operates on, where the gate is applied to the last qubit in the list conditional on the first $n-1$ qubits before. If only one qubit is in the list, it is not conditional and acts as a standard gate. The only exception here is the SWAP gate, which will only be able to have a list of two qubits and will not be conditional. Gates will also be able to be sorted by their column, allowing easy sorting over lists of gates.

7.3.3 Rendering

The `draw` function is the main entry point in rendering the interface, and takes in a `Circuit` to draw. Every button and gate on the interface will look the same; a simple square white box, with a black outline and a symbol inside indicating what it does. For example, the button representing the Hadamard gate will need to have a "H" inside of it. So, first off, its easiest to define a function `drawBox` which will be used to draw the square box at the position passed in. The coordinates of each item will also be fashioned into a grid, so that the rows and columns on the circuit line up smoothly, with equal spacing between everything. For this, we can use two functions, `calcX` and `calcY` along with a `spacing Float` constant defined at the start.

```
drawBox :: Int -> Int -> String -> Picture
drawBox c q t = translate (calcX c) (calcY q) $
                pictures [ color white $ rectangleSolid (spacing/2) (spacing/2),
                           color black $ rectangleWire (spacing/2) (spacing/2),
                           color black $ translate (-spacing/8) (-spacing/8)
                               $ scale 0.1 0.1 $ text t ]
```

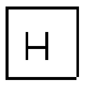


Figure 14: The result of drawBox 0 0 "H"

As can be seen to the left, the box is drawn as required however the text is still off even after moving it, and even then still looks out of place. The lines making up the "H" are not big enough, and are not aesthetically pleasing. Therefore, a function `drawText` is required to handle drawing text onto the buttons.

This function will take in a `String`, such as "H", and draw it using the appropriate rectangles. As the program will only be required to draw certain symbols, pattern matching can be used to handle all specific cases, and then return a blank picture for any unsupported symbol.

From here, all of the buttons can be drawn efficiently using a list containing the text for each button, and mapping over this list with increasing values of x to draw each button. For this, a special case of `drawBox` needs to be implemented which does not translate the drawn picture to an x and y, allowing this to be handled by the calling function. Finally, a thicker line needs to be drawn under the buttons dictating a separation from the buttons and the circuit. However, as thick lines are not possible in gloss, the program simply draws two lines, with one under the other.

```
buttons :: [String]
buttons = [ "+",
            "-",
            "H",
            "X",
            "Y",
            "Z",
            "Swap",
            "Run",
            "Sim",
            "Clear" ]
```

```
drawButtons :: Picture
drawButtons = pictures $
              (map (\(x, b) -> translate (x*spacing) 0 $
                  drawBox (-1) (-1) b) $ zip [0..] buttons) ++
              [ color black $ line [(-spacing/2, -spacing/2),
                                    (width-spacing, -spacing/2)],
                color black $ line [(-spacing/2, -spacing/2-1),
                                    (width-spacing, -spacing/2-1)],
                color black $ line [(3*spacing/2, spacing/3),
                                    (3*spacing/2, -spacing/3)],
                color black $ line [(13*spacing/2, spacing/3),
                                    (13*spacing/2, -spacing/3)]]
```

The circuit now needs to be rendered. Given a `Circuit`, functions are required to draw the qubits and the gates. The qubits are simply black lines with the text "—0¿" in front to show that they are initially in the $|0\rangle$ state. It will also show the number of the qubit, improving the readability of the circuit. All this is done by mapping the drawing of the qubit over the qubit list in the circuit record passed in.

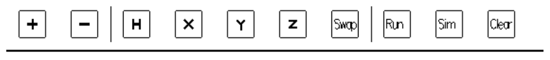


Figure 15: The result of drawButtons

Drawing the gates is slightly more complicated. Due to the design of the `Gate` type, each gate needs to be handled individually. However, each gate is handled pretty much the same, with the only difference being the Controlled-X gate (i.e. when the Pauli-X gate has multiple qubits in its list) and the SWAP gate.

The function `drawGate` will take in a gate with a column value `c` and the qubits it acts on given by `qs`. Black dots representing the conditionality are drawn for each $n - 1$ qubit in the list, and a line is then drawn connecting them to the last qubit. Finally, a box is drawn on the last qubit, with a symbol pertaining to the type of gate. The only complication here is when the conditional qubits in the list are both above and below the last qubit. In order to deal with this, a line has to be drawn between each qubit individually, instead of one long line from the first qubit to the last.

In order to draw a Controlled-X gate, the conditional qubits are dealt with in a similar fashion, however instead of a box containing an "X" on the last qubit, a circle will be drawn with a plus inside; the same as shown in section 1.6.2. Finally, drawing the SWAP gate is simply a case of drawing a cross on each qubit in the list, with a vertical line connecting them together.

Combining all this into one function `drawCircuit circuit = pictures [ drawQubits (qubits circuit), drawGates (gates circuit)]` produces the expected result over an initial circuit with gates already included.

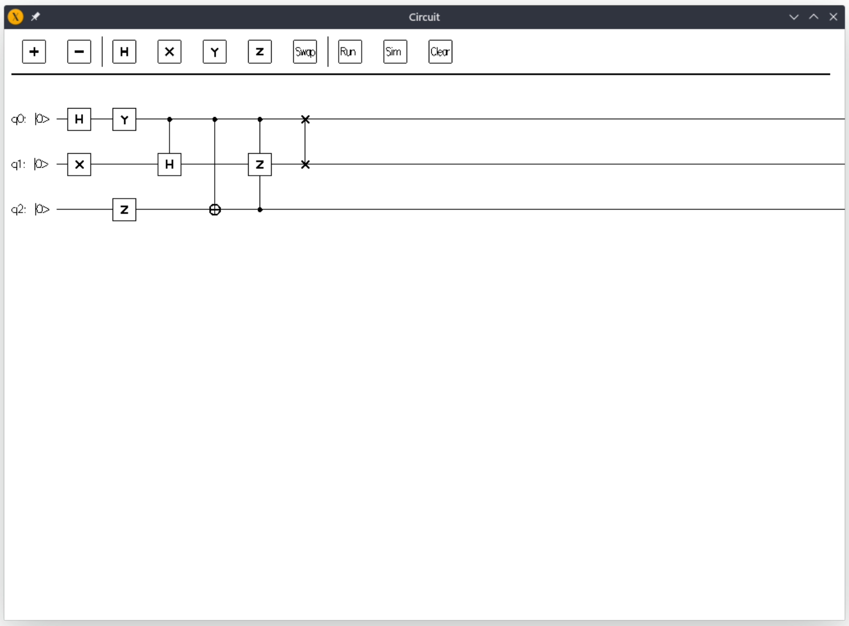


Figure 16: The result of drawing over a circuit with three qubits and gates = [Had 0 [0], PX 0 [1], PY 1 [0], PZ 1 [2], Had 2 [0, 1], PX 3 [0, 2], PZ 4 [0, 2, 1], Swap 5 [0, 1]]

7.3.4 User Interaction

Handling user interaction is done via the `handleEvent` function. This function takes in an `Event` and a `Circuit`, and then returns a `Circuit` which it has altered in some way. Therefore, we can't directly call rendering functions, and instead need a way of storing user interaction details in the `Circuit` type. Simply adding an `edit :: String` and `mouse :: (Float, Float)` will achieve this. `Edit` will store the gate picked up at that moment, i.e. "H" for a Hadamard gate, while `mouse` will store the location of the mouse, allowing the picked-up gate to be seen following the mouse and knowing its location when dropped.

The `Event` type contains all the details of a user event, such as a key press or mouse movement. It can take one of three distinct types:

```
data Event = EventKey Key KeyState Modifiers (Float, Float)
           | EventMotion (Float, Float)
           | EventResize (Int, Int)
```

The `Key` type will contain either the character of the key pressed, the special key pressed (e.g. the Enter key) or the mouse button pressed (e.g. the left mouse button). The `KeyState` dictates if the key has been pressed or released, while the `Modifiers` represent whether the ctrl, shift or alt keys are also down. The `(Float, Float)` in both `EventKey` and `EventMotion` contain the current x and y positions of the mouse in the window.

Upon pressing the left mouse button down, the user may want to either click on a button, click on a location on the circuit to place down a picked-up gate, or click on a gate already on the circuit to pick it up. Using a function `mousePosition :: (Float, Float) -> Circuit -> Circuit`, the program can check which of these has occured. If the right mouse button is instead pressed, any picked up gate is cleared, allowing the user to easily clear gates from the circuit or stop placing a gate down when one is picked up.

Button Press Using the `buttons` list created earlier, the button clicked can be retrieved by finding the column on the grid described earlier which the mouse is inside. If the mouse is indeed over a button, a function `buttonPress :: Circuit -> String -> Circuit` is called to handle which button was pressed. If the `String` passed in is a "+" or a "-", then a qubit needs to be added or removed from the circuit. If the `String` is "Clear", then all gates need to be removed from the circuit. The "run" and "sim" cases will be handled later on. In any other case, the `edit` record in the `circuit` needs to be updated to reflect that a gate has been selected from the buttons.

Adding a qubit is simply a case of appending `[length $ qubits circuit]` to the end of the qubits in the circuit, and removing a qubit is done by applying `init` to the list of qubits as long as the list is not empty. However, at the same time, all the gates on this qubit must also be removed, including gates conditional on the qubit. In order to do this, a function `getQs :: Gate -> [Int]` is used to extract the list of qubits from a gate, and a check is then done to see if the qubit to remove is included in this list. If so, the gate needs to be removed, else nothing is done. All this, combined into a function `remGateOnQubit :: Int -> Gate -> Gate`, is then mapped over the list of gates in the circuit, removing them as necessary. The removal is actually done by setting the gate to `Null` if it needs to be removed, and then applying `filter (/= Null)` over the returned list.

If the Clear button is instead clicked, we simply set the `gates` record in the `circuit` to be `[]`. If a gate button is pressed, the `edit` record in the `circuit` is set to the `String` representing the gate. After this, the rendering functions need to be changed to show that a gate has been picked up.

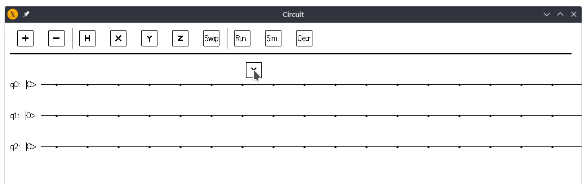


Figure 17: A picked up Pauli-Y gate over an empty circuit.

In the `draw` function described earlier, dots must be drawn on all empty sections of the circuit to highlight that a gate can be placed here. This is done by drawing a black circle at the calculated x and y location on the circuit, and then mapping this over all the qubits on the circuit and a list of `[0..20]`, as roughly 20 columns fit in the window. Following this, the gate selected needs to be drawn under the mouse cursor by translate the result of the `drawBox` function to the mouse coordinates stored in the `mouse` record of the `circuit`.

Placing a Gate If the `edit` record in the `circuit` already has a value, then a gate has already been picked up. Therefore, if the mouse is over the circuit, then the user wants to place the gate down. The function must then calculate the column and row that it is being placed onto and pass it into a function `addGate :: Int -> Int -> Circuit -> Circuit`.

Three things can then occur: the position on the circuit is already taken, the mouse is between two columns, or the position is empty. Dealing with the case of the position being taken is easiest, and just requires nothing to be changed; the gate is still picked up and the circuit is not altered.

If the position is empty, the only thing required to do is to add a new gate to the circuit's list, with the column and row calculated earlier and the type specified by the `edit` record. Then, the `edit` record is cleared to show the gate has been placed down.

If the mouse is between two columns, the user wants to effectively add a new column in between, and place the gate there. Therefore, all the gates with a column value higher than or equal to the position need to be moved right by one column. As there is an instance of `Ord` defined for gates already, the `sort` function can be used to sort by increasing column order. Following this, the `span` function can be used along with a new function `lteCol :: Int -> Gate -> Bool` - which returns True when the gate passed in is in a column less than or equal to the column passed in - the list of gates can be split up into the form `(bG, aG)`, where `bG` contains the gates before the new column and `aG` contains the gates which will be after the new column (i.e. the column value of each gate in this list needs to also be incremented by one). Then, the new gate can be placed in between the two sets, creating a new column for it to be placed onto.

Picking up a Gate Finally, if the `edit` record is empty and the mouse is over the circuit, the user wants to pick up a gate from the circuit. In this case, the function calculates the column and row it is being picked up from and passes it into a function `pickUpGate :: Int -> Int -> Circuit -> Circuit`. This function then calculates whether or not the mouse is in fact over a gate on the circuit, and if it is the gate is deleted (note that the column is not deleted if there is nothing else in it) and then the `edit` record is changed to show the gate has been picked up.

Key Press The number keys will be used to specify the control qubits for a gate which is picked up. For example, while a Hadamard gate is picked up, pressing the '0' key will specify that this gate will be conditional on the qubit "q0". Handling this is relatively simple using the `EventKey (Char c) ...` form of the `Event` type.

If `c` is a number key, then the `edit` record of the `circuit` is updated by appending the a space along with the number pressed as long as a gate is picked up. If the gate is a SWAP gate, then this can only occur once (as the key pressed represents the qubit to swap with when placed on a different qubit), and so the length of the `edit` record must be checked. If the backspace key is instead pressed, causing `c` to equal `'\b'`, then the conditional qubits need to be cleared, reverting the `edit` record back to simply store the gate type picked up.

```
handleEvent (EventKey (Char c) Up _ pos) circuit =
  case head e of
    'H' | c == '\b'                             -> circuit { edit = "H" }
    'H' | c >= '0' && c <= '9'                  -> circuit { edit = e ++ [' ', c] }
    ...
    'S' | c >= '0' && c <= '9' && length e == 4 -> circuit { edit = e ++ [' ', c] }
    ...
  where
      e = edit circuit
```

During the render process, the numbers already pressed can be shown easily using gloss' `text` function. The implementation must now be able to handle placing gates down with conditional qubits. In order to acheive this, a function `getControls :: String -> [Int]` is created. This function removes all spaces from

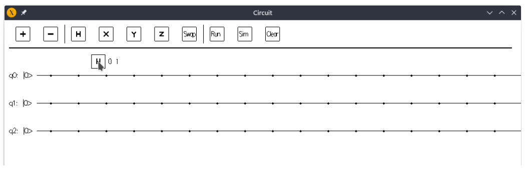


Figure 18: Picking up a Hadamard gate, with control qubits "q0" and "q1".

the String passed in, removes duplicate numbers, and then converts each character into its relevant Int using the read function. Now, new gates can be created by appending the qubit it is placed on at the end of the list of control qubits. If any of the control qubits are equal to the qubit the gate is being placed on, the operation is cancelled to prevent errors from occuring.

7.3.5 Executing the Circuit

In order to execute the circuit, it first needs to be compiled. This is actually fairly simple to do using the list of qubits and list of gates in the circuit. Each qubit in the list needs to be created in the state $|0\rangle$, and then the unitary operation relating to the circuit needs to be executed using the applyU function, and finally the qubits are measured.

```
compileCircuit :: Circuit -> QIO [Bool]
compileCircuit circuit = do
  qs <- mkQbits (length $ qubits circuit) False
  applyU (gatesToU (gates circuit) qs)
  measQbits qs
```

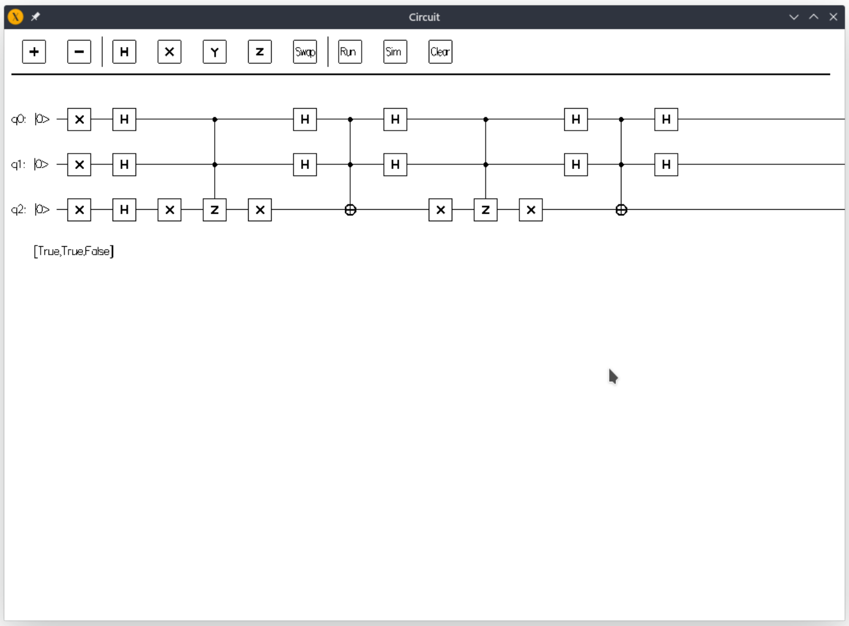
The gatesToU function is used to recursively convert each gate into its corresponding unitary operation, using the function gateToU :: Gate -> [Qbit] -> U, and then combine the operations using the <> operator. Upon being given a gate, the gateToU function will map the relevant unitary operation (e.g. uhad for a Hadamard gate) over the list of qubits, where each qubit is at the index specified by the gate's qubit list. A function condU is used to conditionally apply the operation according to the first $n - 1$ qubits in the gate's qubit list.

When the circuit is compiled, it will be able to be run or simulated using the appropriate functions when the user clicks either the "Run" or "Sim" buttons described above. The result of either is then saved in a new record result :: String inside the circuit type. For simulation, this is done simply using show $ sim $ compileCircuit circuit, as the result of sim can be converted straight into a String. However, running the circuit is more difficult, as run returns the type IO a, in this case IO [Bool].

As an IO a type cannot be converted into a String, as it is wrapped in the IO monad, we must find a way of extracting the values. This can be done using the function unsafePerformIO :: IO a -> a, which does what we want by performing the IO actions as a pure function. Because of this, Haskell will optimize the actions (as it does not realise it is an IO action) and so certain side-effects may not be executed. However, the computation will not produce any side-effects, and its only action is to generate a random number using Random.randomRIO, and so it is safe to use in this instance, producing a String result via show $ unsafePerformIO $ run $ compileCircuit circuit.

In order to render the results, the draw function needs to be altered. The result record in the circuit will contain a String of what to draw, and so gloss' text function can be used to render the text underneath the rows of qubits.

The execution of the built circuit can be made easier changing the handleEvent function, so that when the Enter key is pressed, the circuit is run.



[True,True,False]

Figure 19: Running Grover's algorithm, searching for the state $|110\rangle$, over three qubits.

7.4 Testing

8 Reflections

9 References

[Aar05] Scott Aaronson. Limits on efficient computation in the physical world. 2005. https://arxiv.org/abs/quant-ph/0412143v2.

[AG09] Thorsten Altenkirch and Alexander S. Green. *The Quantum IO Monad*, page 173–205. Cambridge University Press, 2009.

[Aug09] Lennart Augustsson. Haskell package timeit-1.0.0.0: Time a computation, 2009. Last accessed 4/2/2019. http://hackage.haskell.org/package/timeit-1.0.0.0.

[bas] Changelog for base-4.11.0.0 — Hackage. http://hackage.haskell.org/package/base-4.11.0.0/changelog.

[BCDP96] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54:1034–1063, Aug 1996. https://link.aps.org/doi/10.1103/PhysRevA.54.1034.

[Ben82] Paul Benioff. Quantum mechanical models of turing machines that dissipate no energy. *Phys. Rev. Lett.*, 48:1581–1585, Jun 1982. https://link.aps.org/doi/10.1103/PhysRevLett.48.1581.

[Bor] Max Born. Zur quantenmechanik der stoßvorgänge [on the quantum mechanics of collisions]. *Zeitschrift für Physik*.

[DJ92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1992.0167.

[DP85] David Deutsch and Roger Penrose. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070.

[Fey82] Richard P. Feynman. Simulating physics with computers. *Internation Journal of Theoretical Physics*, 21(6/7), 1982. https://people.eecs.berkeley.edu/christos/classics/Feynman.pdf.

[Gid13] Craig Gidney. Grover's Quantum Search Algorithm, 2013. Last accessed 16/1/2019. http://twistedoakstudios.com/blog/Post2644_grovers-quantum-search-algorithm.

[Gre09] Alexander S. Green. G53/G54 NSC Lectures given by Alexander S. Green over the academic years 2009/2010., 2009. http://www.drinkupthyzider.co.uk/asg/NSC/.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. http://doi.acm.org/10.1145/237814.237866.

[ibma] IBM Q Experience Community. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/community.

[ibmb] IBM Q Experience Documentation Beginners Guide. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=beginners-guide&page=introduction.

[ibmc] IBM Q Experience Documentation Full Guide. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=full-user-guide&page=introduction.

[ibmd] IBM Q Experience, the online platform created by IBM allowing users access to Quantum Computers. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/experience.

[ibme] Jupyter Notebook Viewer - Qiskit Tutorials. Last accessed 06/12/2018. https://nbviewer.jupyter.org/github/Qiskit/qiskit-tutorial/blob/master/index.ipynb.

[Lip18] Ben Lippmeier. gloss: Painless 2D vector graphics, animations and simulations., 2018. Hackage: http://hackage.haskell.org/package/gloss Main page: http://gloss.ouroborus.net.

[Lub17] David Lubensky. Quantum Computing gets an API and SDK, March 2017. https://developer.ibm.com/dwblog/2017/quantum-computing-api-sdk-david-lubensky/.

[Mer07] N. David Mermin. *Quantum Computer Science: An Introduction.* Cambridge University Press, 2007.

[New16] IBM Newsroom. Ibm makes quantum computing available on ibm cloud to accelerate innovation. May 2016. https://www-03.ibm.com/press/us/en/pressrelease/49661.wss.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[SP05] D. Solenov and V. Privman. Evaluation of decoherence for quantum computing architectures: Qubit system subject to time-dependent control. 2005. https://arxiv.org/abs/cond-mat/0506286.

[Woo17] Dr. James Wootton. Quantum Battleships: The first multiplayer game for a quantum computer, 2017. Last accessed 14/12/2018. https://medium.com/@decodoku/quantum-battleships-the-first-multiplayer-game-for-a-quantum-computer-e4d600ccb3f3.

[Yor16] Brent Yorgey. diagrams: Embedded domain-specific language for declarative vector graphics, 2016. Hackage: http://hackage.haskell.org/package/diagrams Main page: https://diagrams.github.io.

[Zur91] W.H. Zurek. Decoherence and the transition from quantum to classical - revisited. *Physics Today*, 44:36–44, 10 1991.