Session 3: Using Light to Create a Quantum Computer

Dominic Moylett
Quantum Engineering Centre for Doctoral Training,
University of Bristol
dominic.moylett@bristol.ac.uk

July 25, 2016

Worksheet Details

This worksheet is for the third session of the Quantum Computing day in the 2016 "Quantum in the Summer" school. In the previous sessions, we have seen how a quantum computer works and some of the problems that we have managed to speed up thanks to quantum computing. Now, this session will look at how we might build a quantum computer using photons: particles of light. Work on building such a computer is being completed right now within the Centre for Quantum Photonics at the University of Bristol.

1 Why Light?

Throughout this summer school you have seen some of the quantum properties of photons, which are particles of light. These quantum properties led to one of the first proposals of implementing a quantum gate to be done in linear optics, by Gerard James Milburn in 1989. Since then, designs for general purpose quantum computers have been proposed as early as 2001, and there is a big push right now to develop and implement more practical schemes for a quantum computer.

But why do we want to consider a quantum computer using photons? The answer is that there are a variety of reasons. The first benefit is that photons are already used for communication – we use them in fibre optic cables all the time – so it is convenient for communication between quantum computers.

Another benefit is that it is possible to make optical setup really small, even fitting them onto a single silicon chip. The Centre for Quantum Photonics is researching how we can make practical quantum computers on chips this small so that we can piggyback off the decades of work on silicon chips in the electronics industry.

It is worth noting however that optics also has its disadvantages. In particular, it is hard to make photons interact with one another, resulting in problems which we shall see in the final section. But, work is currently being done on how we can avoid these issues.

2 Qubits and Measurement

So what does an optical qubit look like? There are many ways we can encode a qubit in a photon, but the way we will look at today in particular is called *path encoding*. Essentially, we encode the qubit as a single photon travelling down one of two paths. If the photon is in one path, we say that it is in the $|0\rangle$ state, and if the photon is in the other path we say that it is in the $|1\rangle$ state.

We can implement the above encoding using the QCEngine. To do so, we will need to use the commands `qc.start_photon_sim` and `qc.start_photon_sim` to tell the engine that we are now going to be using linear optics. When working on linear optics, our code will look like it has two qubits but is actually one qubit represented by two paths.

```
qc.reset(2);
qc.write(1);
qc.start_photon_sim();
qc.stop_photon_sim();
```

You can also now see that we have an option called "Fock state". The Fock state of a quantum system is related to linear optics, and describes how many photons are going down a specific path at a given point in time. In the case above, we have one photon in the first path and none in the second, so our Fock state is $|1-\rangle$.

So if this is how we encode a qubit, how do we measure which state it is in? This is done using single photon detectors. A single photon detector will release an electrical signal whenever a photon reaches it. By placing detectors on both the $|0\rangle$ and $|1\rangle$ path and waiting for one of them to register a photon, we can see which path the photon was in and thus measure the state.

To detect which path a photon is in, we can use the `qc.read` command:

```
qc.reset(2);
qc.write(1);
qc.start_photon_sim();
qc.stop_photon_sim();
qc.read();
```

So if we have a photon in the $|0\rangle$ path and measure it, we find that the photon is in the $|0\rangle$ path.

3 Beam Splitters

Now that we have an understanding of how we will be encoding our qubits, we need to apply gates to them. We can do this using a variety of optical components.

The first gate we shall be using is a beam splitter. A beam splitter is a partial mirror, which will with some probability r reflect the photon and with probability $1 - r$ let the photon pass through. The probability of the beam splitter reflecting is known as its *reflectivity*.

We can use this component in our optical quantum computer to generate a superposition:

```
qc.reset(2);
qc.write(1);
qc.start_photon_sim();
qc.dual_rail_beamsplitter(0.5, 1|2);
qc.stop_photon_sim();
qc.read();
```

By clicking back and forth from the detectors, we can see that half of the time our photon is in the $|0\rangle$ path, and half the time our photon is in the $|1\rangle$ path. This is similar to what we had with the Hadamard gate.

But the Hadamard gate wasn't just special for resulting in $|0\rangle$ half the time and $|1\rangle$ half the time. The other important property of the Hadamard gate was that applying it twice resulted in us seeing the original state again. Do we see the same effect when using a 50-50 beam splitter twice?

```
qc.reset(2);
qc.write(1);
qc.start_photon_sim();
qc.dual_rail_beamsplitter(0.5, 1|2);
qc.dual_rail_beamsplitter(0.5, 1|2);
qc.stop_photon_sim();
qc.read();
```

It seems that we do not. If we start in the Fock state $|1-\rangle$ and apply a beam splitter twice, we end up with the state $|-1\rangle$. Likewise if we start in the state $|1-\rangle$ we end in $|1-\rangle$. So applying a beam splitter twice does not give us the original state, but instead flips the qubit[1].

The reason for this is because a beam splitter does not produce the same state as a Hadamard gate does. This is because when a photon is reflected off the beam splitter, a phase is introduced to that photon. So while the state we measure after sending a photon in path $|0\rangle$ through might be $|0\rangle$, the actual state would be $i|0\rangle$, where $i = \sqrt{-1}$.

We can however make the photon come out its original path by placing a phase controller on one of the photon paths between the two beam splitters. In linear optics on integrated circuits, this can be done by just placing a heater next to the photon path:

[1]Applying the operator twice also adds some phase to the overall state, which we will not worry about.

```
qc.reset(2);
qc.write(1);
qc.start_photon_sim();
qc.dual_rail_beamsplitter(0.5, 1|2);
qc.phase(180, 1);
qc.dual_rail_beamsplitter(0.5, 1|2);
qc.stop_photon_sim();
qc.read();
```

The above code will apply a phase of $180°$ to any photons in the $|0\rangle$ path. This is enough to cause the single photons to interfere differently and result in the photon exiting the $|0\rangle$ path. It is worth noting however that this is still not strictly the same as the Hadamard gate.

Task 3.1. *The first argument of the $qc.phase$ argument can be any number from 0 to 360. Try placing different numbers in and seeing how often you measure $|0\rangle$ or $|1\rangle$. Some numbers you try might produce a superposition, so try running them multiple times. This experiment is an example of a Mach-Zender interferometer.*

4 Controlled-NOT

So we can get a superposition with linear optics, but can we generate entanglement? Well, we saw in the first session that if we can generate a superposition and implement a $CNOT$ gate, then we can generate an entangled pair of qubits. We already have one of those, so all we need to do is create a $CNOT$ and then we're done!

Unfortunately, implementing the $CNOT$ is where our current method for implementing a quantum computer using linear optics really starts to struggle. This comes back to the point made in Section 1: Photons do not interact well with each other. So we need to be smarter in order to implement this gate.

To start with, we will implemented a related gate, which we shall call CZ for Controlled-Z[2]. We will implement this gate using six photon paths: Two for the control qubit, two for the target qubit, and two extra paths required for the gate. The gate can be implemented using beam splitters with reflectivity $1/3$:

```
qc.reset(6);
qc.write(4|8);
qc.start_photon_sim();

qc.dual_rail_beamsplitter(1.0/3.0, 1|2);
qc.dual_rail_beamsplitter(1.0/3.0, 4|8);
qc.dual_rail_beamsplitter(1.0/3.0, 16|32);

qc.stop_photon_sim();
qc.read();
```

Task 4.1. *Try running this code a few times. If the operation works correctly then you should see a blue circle with the line pointing downwards in the state $|12\rangle$, indicating that the target qubit's $|0\rangle$ state has phase -1. How often does this happen? Can you see any patterns when it does happen?*

[2]Recall that $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$.

Unfortunately, this gate will not always work. In fact, if you try running it right now, it will almost never work as we want it to.

The cases in which it does work satisfy the following properties:

- There are no qubits in the first and last photon paths,
- and there is exactly one photon in the control qubit paths
- and there is exactly one photon in the target qubit paths.

If any of these cases don't hold, then our gate will not operate as it should. So, let's only perform the rest of the computation when they do hold. This idea is known as *postselection*, where we only consider those cases where the circuit would work as it is intended to. We can add these conditions with the following code:

```
qc.reset(6);
qc.write(4|8);
qc.start_photon_sim();

qc.dual_rail_beamsplitter(1.0/3.0, 1|2);
qc.dual_rail_beamsplitter(1.0/3.0, 4|8);
qc.dual_rail_beamsplitter(1.0/3.0, 16|32);

qc.postselect(0, 1|32);
qc.postselect_qubit_pair(2|4);
qc.postselect_qubit_pair(8|16);
qc.stop_photon_sim();
qc.read();
```

Running this now, we can see that this always produces the state $-|10\rangle$ when starting with the state $|10\rangle$.

Task 4.2. *Try running the Controlled-Z operation on the other states:*

- $|11\rangle$ *corresponds to* `qc.write(4|16)`
- $|00\rangle$ *corresponds to* `qc.write(2|8)`
- $|01\rangle$ *corresponds to* `qc.write(2|16)`

These cases should all produce the original state.

Now we need to implement our $CNOT$. To do this, we shall reuse a trick from session 1. Remember in teleportation when we performed a Controlled-Z operation consisting of a Hadamard, a $CNOT$ and another Hadamard? Well, we can similarly perform a $CNOT$ by performing a Hadamard, a Controlled-Z and another Hadamard. Of course, we have not seen how to implement a Hadamard gate using linear optics. But we have seen a beam splitter, which is close to a Hadamard. So let's try using that instead:

```
qc.reset(6);
qc.write(4|8);
qc.start_photon_sim();

qc.dual_rail_beamsplitter(0.5, 8|16);

qc.dual_rail_beamsplitter(1.0/3.0, 1|2);
qc.dual_rail_beamsplitter(1.0/3.0, 4|8);
qc.dual_rail_beamsplitter(1.0/3.0, 16|32);

qc.dual_rail_beamsplitter(0.5, 8|16);

qc.postselect(0, 1|32);
qc.postselect_qubit_pair(2|4);
qc.postselect_qubit_pair(8|16);
qc.stop_photon_sim();
qc.read();
```

Now, when our control photon is in the $|1\rangle$ path, our target photon will be in the same path it started in, but if our control photon is in the $|0\rangle$ path, the target photon will be in the opposite path to what it started in.

Task 4.3. *Verify the above claim by running the circuit on the different input states. To change the input state, follow the same instructions as for Task 4.2.*

Task 4.4. *How could you use this gate to generate entangled qubits?*