Session 2: Quantum Algorithms

Sam Pallister
Quantum Engineering Centre for Doctoral Training,
University of Bristol
sam.pallister@bristol.ac.uk

July 26, 2016

Worksheet Details

This worksheet is for the second session of the Quantum Computing day in the 2016 "Quantum in the Summer" school. This session is focused on *quantum algorithms*, i.e. getting a quantum computer to compute something useful, in much less time than a regular or "classical" computer would take.

1 Introduction

You've seen in the previous session how fundamental operations of a quantum computer work - how we can take a bunch of qubits, manipulate them with different kinds of gates and then measure them to get a certain outcome. We've seen how, if your device is described by quantum mechanics, there are behaviours that emerge that are unachievable just with classical physics (we talked about two: *superposition* and *entanglement*). We've even looked at examples where these behaviours go from being purely academic, to helping you to construct genuine scenarios where seemingly impossible things (like teleportation) can occur.

Despite this being curious, bizarre and worth exploring in its own right, the claim by quantum physicists is bigger than this - namely, that we can use these operations to build a fundamentally different species of computer. That means cooking up a recipe involving qubits, quantum gates and measurements, the outcome of which is the answer to a pre-specified problem. In short, we're looking for a *quantum algorithm*.

There's a few things worth mentioning before we dive head-first into building some algorithms. The first is that no-one knows how to take an arbitrary problem and speed it up by using a quantum computer (in fact, the opposite is almost certainly true - that there are whole classes of problems where having a quantum computer doesn't help you get to the answer faster). Secondly, there isn't a standard structure to crafting a quantum algorithm; in fact, as you'll see in this worksheet, they frequently work very differently from each other. The upshot of this is that we currently only have a short list of problems that are sped-up by a quantum computer. An (almost) comprehensive list is kept up to date at the *Quantum Algorithm Zoo*:

`http://math.nist.gov/quantum/zoo/`

This shouldn't be taken as bad news, however - it just means that we need lots of curious young minds to work out the details of what else we can program our quantum computer to do!

In the rest of this worksheet, we'll introduce and explore three separate examples of quantum algorithms: Deutsch's algorithm, Grover's algorithm and the Quantum Fourier transform.

2 Deutsch's Algorithm

The majority of this section on Deutsch's algorithm is heavily inspired by Ashley Montanaro's notes *"Winning a game show with a quantum computer"*. A full copy can be found here:

`https://www.cs.bris.ac.uk/~montanar/gameshow.pdf`

We are now ready to see one problem where quantum computers can truly outperform classical ones, first proposed by David Deutsch (and expanded by Richard Josza) in 1992. First, we'll pose the problem in colloquial terms and then we'll deal with the mathematical description of the problem.

Its 2050. The most popular TV show in the UK for the last decade has been the game show Meal or No Meal. The rules of the game are simple. There are 100 rounds, and in each round, the contestant is shown two closed saucepans on a table. Each saucepan may or may not contain a delicious meal (say, a roast chicken), so there are somewhere between zero and two meals on the table. The contestant is allowed to open at most one saucepan, and then has to say whether or not they think there is exactly one meal on the table. If they are right a hundred times in a row, they win a billion dollars (approx 50p in todays money). Nobody has ever collected the prize. Why? Imagine the game show host tosses a coin to decide whether to put a chicken in each saucepan. Now, whichever saucepan the contestant opens, there is a 50% chance that the other saucepan contains a chicken. So, whether the first saucepan had a chicken in it or not, the probability of there being exactly one chicken between the two saucepans is 50%. This means that, whatever the contestant says, they have only a 50% chance of being right each time a probability of 0.5. So the probability of being right a hundred times a row is $0.5 \times 0.5 \times \ldots \times 0.5 = 2^{-100}$; i.e. negligibly small! This chapter will explain how, if a contestant had a quantum computer, they could win the billion-dollar prize with certainty. That is, a quantum computer can tell whether or not there is

exactly one chicken between two saucepans, with only one peek into a saucepan.

Now for the mathematical description of this game - suppose we have a function f, which takes one bit as input and returns one bit as output. In other words, $f(0)$ is either 0 or 1 and $f(1)$ is either 0 or 1. Think of the input to the function as the choice of which saucepan to peek in, and the output is whether that saucepan contains a chicken or not. Deutsch's problem is to determine if $f(0) = f(1)$ - if this statement is true, then we know for sure that there is either 0 ($f(0) = 0, f(1) = 0$) or 2 ($f(0) = 1, f(1) = 1$) chickens in the pair of saucepans. Likewise, if it false there is only one chicken in the pair of pans. This information is exactly what is required to win Meal or No Meal, every time.

On a classical computer, we would need two queries to the function f: One to figure out what $f(0)$ is and one to figure out what $f(1)$ is. This is just a statement that we can't normally guarantee a win at Meal or No Meal without looking in both pans. But can we do better with a quantum version of this circuit? This is the challenge we set ourselves with this section - to cook up a scenario where we only need one quantum input in order to win Meal or No Meal with certainty.

First, we need to construct circuits that carry out the function $f(x)$, for any choice of $f(x)$ that we care to make (there are four possible f's - given an input 0 or 1 the outputs can be 0 and 0, 0 and 1, 1 and 0 or 1 and 1).

One could consider a single qubit circuit to perform f, but instead we'll take a two-qubit circuit. The input and output to the circuit are given by the following diagram, where x is the choice of saucepan to peek into and $f(x)$ denotes the presence of a chicken:

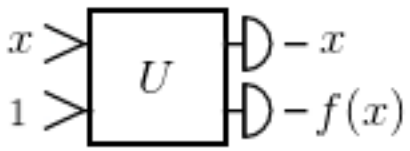

Figure 1: A schematic for the circuit to evaluate the function $f(x)$. Two qubits are initialised in the states $|x\rangle$ and $|1\rangle$, they pass through a circuit (called U, here), and the output are qubits in the state $|x\rangle$ and $|f(x)\rangle$.

We'll now take a look at the four possible choices for the function f. Remember, for the following tasks - the qubit on the bottom rail is prepared in the state $|1\rangle$, not the state $|0\rangle$ - and so you'll either have to write a line of code to prepare it in the correct state, or you'll have to prepare it in the state $|0\rangle$ and then apply a NOT gate to the bottom qubit at the start of the program.

Task 2.1**:** Check that by building a circuit like in Fig 1 above, where the U part is just doing nothing, we have a function where $f(0) = 1$ and $f(1) = 1$. For this choice of function, there is a chicken in each saucepan.

Task 2.2**:** Check that, for the following function, U:

```
qc.not(2);
```

we have that $f(0) = 0$ and $f(1) = 0$. For this choice of function, there are no chickens in either pan.

Task 2.3**:** Check that, for the following function, U:

```
qc.cnot(2,1);
```

we have that $f(0) = 1$ and $f(1) = 0$. For this choice of function, there is only a chicken in the first pan.

Task 2.4**:** Check that, for the following function, U:

```
qc.not(1);
qc.cnot(2,1);
```

we have that $f(0) = 0$ and $f(1) = 1$. For this choice of function, there is only a chicken in the second pan.

Remember - the goal is to tell whether $f(0) = f(1)$ or not (i.e. to tell whether there is exactly a single chicken in the pair of pans). So, we need an algorithm that can always tell apart the cases in Tasks 2.1 and 2.2 from the cases in Tasks 2.3 and 2.4.

While all these circuits do successfully compute $f(x)$, they can't be the full quantum algorithm we're seeking if the input qubit x is just in the state $|0\rangle$ or $|1\rangle$. This is due to the circuit having the same limitation as the classical algorithm; i.e. to check whether the circuit satisfies $f(0) = f(1)$ requires feeding in the input $|0\rangle$ first to find $f(0)$, and then feeding in $|1\rangle$ afterwards to find $f(1)$. So at this stage, we haven't managed to do any better with a quantum computer.

Now, we'll try making full use of some quantum weirdness. Firstly, rather than just preparing the input qubit in the state corresponding to which saucepan we want to look in, we'll do something cleverer. In particular, we know that the input qubits need not be in the states $|0\rangle$ or $|1\rangle$; we can modify the input state so that it is in superposition of both. Also, we have drawn a picture of a circuit with two output bits rather than just one; let's try and make use of this by measuring both qubits and looking at both output bits.

Task 2.5**:** Add a Hadamard gate to the circuit so that top qubit enters in a superposition. What is the outcome of the circuit when we measure both qubits, for all the possible choices for the function from Tasks 2.1 - 2.4? Does this help us win the Meal or No Meal game with only a single input? Remember: the qubits may well be in a superposition just before you measure them, so you might need to measure of few times to see all the possible outputs.

You should notice that starting the top qubit as a superposition doesn't help things - for example, receiving the outcome 0 for the top qubit and 1 for the bottom qubit could mean that the function is doing nothing, like in Task 2.1, or it could be applying a CNOT gate, like in Task 2.3. Failing to tell apart these cases

means that if we get this outcome, we can't win Meal or No Meal - Task 2.1 has $f(0) = f(1)$, whereas Task 2.3 does not.

So, what's the solution? The first step is to apply Hadamard gates to both qubits, putting them both in superposition, before sending them through gates that perform the function $f(x)$. The second step is to "undo" the superposition by applying Hadamards again to both qubits just before measurement.

Task 2.6**:** Have a go at implementing the correct circuit for Deutsch's algorithm. Does it win Meal or No Meal with certainty? If yes, how does it do it? If not, can you make any changes so that it does work?

If you're a little lost, here's a simple circuit for carrying out Deutsch's algorithm in full:

```
qc.reset(2);
qc.write(2);
qc.hadamard(1|2);
// Add your particular choice of function $f(x)$ here
qc.hadamard(1|2);
qc.read();
```

Aside: the extended form of Deutsch's algorithm is called the *Deutsch-Josza algorithm*, which makes the following modification to Meal or No Meal. Suppose now that instead of two closed saucepans, we have some bigger even number, N. The gameshow host promises you that one of two situations can occur: either every saucepan has a chicken in it, or only half do. The prize money is yours if you can tell these two cases apart with certainty, by only looking in a single saucepan. Spoiler alert: there's a quantum circuit that can always win this game with a single input, regardless of what the number N is!

Task 2.7**:** How many saucepans would you normally (i.e. without access to a quantum computer) need to look in, to tell the two cases in the Deutsch-Josza game apart?

Extension Task: Try to implement a circuit that runs the Deutsch-Josza algorithm for four saucepans/qubits, rather than the two saucepans/qubits in Deutsch's algorithm.

3 Grover's algorithm

Grover's algorithm was published by Lov Grover in 1996, and it remains one of the most influential and important quantum algorithms that we know about. In short, it is a quantum algorithm for searching for an item in a list. This might sound pretty mundane, but essentially a regular computer only does a few basic things - basic arithmetic in the CPU, and looking up values of data stored in memory. When you break down a computer this way, the task of searching through a list becomes really quite important. When it was published, Grover was a research scientist at Bell Labs (which was then owned by the American phone company, AT&T). Working as a research scientist for a phone company might be one of the reasons why he framed his problem as that of searching for a phone number in a phone book - if you're interested, you can read his paper here:

`https://arxiv.org/abs/quant-ph/9605043`

We're going to take a slightly different application, however. Here's the scenario: you're playing the board game Guess Who, with a particularly dumb computer/artificial intelligence/robot (have a quick Google if you've never played this board game before!). Guess Who is essentially a game of 20 questions, where each player takes turns to find out which character from a list their opponent is holding. The robot's strategy for beating you at the game is pretty unsubtle - they're going to take each person on the Guess Who board, one by one, and ask you whether you're holding that person. I.e. they'll sequentially ask questions like *"Is it Hillary?"*, *"Is it Donald?"* but not something more subtle, like *"Do they wear pant suits?"* or *"Are they blatantly wearing a wig?"*.

Task 3.1**:** For a Guess Who board with n faces on it, how many dumb questions would the robot have to ask until they got the right answer, on average?

It turns out that, for a simple computer, the best search algorithms don't do much better - they always have to take a number of guesses that is proportional to n, the number of faces on the Guess Who board.

Given the stupidity of the robot's strategy, you'd think you'd have an easy time beating it by asking more sensible questions. But, here's the catch - you've promised to let your new quantum computer play the game for you. Moreover, to stop you cheating, you don't ask the robot questions; the robot has the name of its Guess Who card stored in memory somewhere, and it does something specific to the state of your quantum computer every time you take a turn (which we'll discuss shortly). So, the question becomes: *can you program your quantum computer to beat the robot, before it beats you?*

Let's formulate this problem slightly more mathematically. Rather than referring to names of characters on the board, we'll give each character a number in binary. For example, we could make the following replacement:

Character	Binary address
Donald	00
Hillary	01
Ted	10
Bernie	11

Table 1: An example table of Guess Who characters and addresses.

When the robot takes a turn, it asks whether you are holding a particular character; or, it is giving you a set of binary digits that represent that character (let's call this string of bits x). Likewise, you are sending back either "YES" or "NO" - let's encode this in a single bit, with 0 for "NO" and 1 for "YES". The robot's turn is then, essentially, to get an output to the following function, for a single input:

$$g(x) = \begin{cases} 1 & \text{if } x \text{ denotes the correct character} \\ 0 & \text{if } x \text{ denotes a wrong character} \end{cases}$$

The robot declares that it has the correct answer when the function g returns a 1.

Now, for the quantum algorithm - prior to the start of the game, the quantum computer is set up with n qubits for a game with 2^n characters. Each of the 2^n amplitudes are to be interpreted as the probability of the quantum computer asserting that the robot's held character corresponds to that amplitude.

Task 3.2**:** Set up the quantum computer to play the game with 16 characters. Before starting the game, the computer applies Hadamard gates to every qubit. Add this to your circuit (here's a tip - writing the Hadamard function with just an empty pair of braces applies Hadamards to every qubit in the circuit). Given that each amplitude is the probability of the quantum computer guessing a particular character, and that the quantum computer is yet to ask the robot anything, does this initial state make sense?

Every time the quantum computer takes a turn and asks for some information about the character the robot is holding, it evaluates a function called the "oracle", which by itself does not change any of the probabilities of guessing a particular character. Building the circuit from scratch is a little tedious, so we'll introduce the whole code in one go and then explore what it does. Here's the code needed to set up the problem and carry out one iteration of the "oracle" (i.e. for the quantum computer to ask the robot a single question):

```
qc.reset(4);

// The main function that will take the Guess Who
    turns for the quantum computer
function main()
{
  qc.write(0);
  qc.hadamard();
  for (var i = 0; i < 1; ++i)
  {
    oracle_flip(12);
  }
}

// oracle_flip codes what happens to the quantum
    state each turn
function oracle_flip(value_to_flip)
{
  qc.codeLabel('Oracle');
  qc.not(~value_to_flip);
  qc.cz();
  qc.not(~value_to_flip);
}

main();
```

The structure of this code is slightly more involved than in the previous cases, so if you're not too familiar with coding and you like to step through what the code means, ask a demonstrator.

Task 3.3**:** Check that calling the oracle does not change the probabilities of guessing a certain character. What about if it is called lots of times (i.e. if the index for the loop, i, is increased to say, 3)? If it doesn't alter the probabilities, what does it do? What happens if you change the argument of the function oracle_flip (e.g. replacing oracle_flip(12) with oracle_flip(7))?

Here's the interesting part - in between implementing calls to the oracle, the quantum computer evaluates a different function (which we'll call "Grover", after its originator). So the total algorithm is going to look like repetitions of "ask the robot a question with the oracle, whir through the Grover function, wait; ask the robot a question... etc.". Again, implementing this iterated algorithm from scratch with the Grover function included is a little fiddly, so here's a copy of the code that adds in the Grover function:

```
qc.reset(4);

// The main function that will take the Guess Who
    turns for the quantum computer
function main()
{
  qc.write(0);
  qc.hadamard();
  for (var i = 0; i < 1; ++i)
  {
    oracle_flip(12);
    grover_iter();
  }
}

// oracle_flip codes what happens to the quantum
    state each turn
function oracle_flip(value_to_flip)
{
  qc.codeLabel('Oracle');
  qc.not(~value_to_flip);
  qc.cz();
  qc.not(~value_to_flip);
}

// grover_test is what the quantum computer does
    between turns
function grover_iter()
{
  qc.codeLabel('Grover');
  qc.hadamard();
  qc.not();
  qc.cz();
  qc.not();
  qc.hadamard();
}

main();
```

Task 3.4**:** Implement the code above, and then step through the circuit to examine what the "Grover" function is doing.

Task 3.5**:** Modify the code to take four Guess Who turns for the quantum computer (i.e. run the oracle, Grover pair 3 times - oracle, Grover, oracle, Grover, oracle, Grover, oracle, Grover). What happens to the probability of guessing each of the characters throughout the circuit? Does this make it any easier to puzzle out what the Grover function is doing?

It turns out with some careful mathematical analysis, you only need about $\sqrt{n}$ iterations of the oracle, Grover pair for a game with n characters, in order to get the right answer with high probability - this is much better than the robot (or any classical computer, for that matter) can do! Here's an rough plot of how many turns the robot and the quantum computer need, on average, as a function of the number of characters on the Guess Who board:

`http://www.wolframalpha.com/input/?i=Plot+y%3Dn+and+y%3Dsqrt(n)+for+n%3D0+to+50`

The upshot is, unless you're really unlucky and the robot guesses correctly straight away, you should comfortably win a game of Guess Who by relying on your quantum computer.

Extension Task: Modify the code to run the oracle/Grover pair more than 3 times. What happens to the probability of guessing the correct character? Can you attempt to write down a mathematical description of the probability of guessing the correct character, as a function of the number of iterations of the oracle, Grover pair?

4 The Quantum Fourier Transform

The final algorithm we'll step through is the *Fourier transform*. The Fourier transform is a piece of mathematics discovered by French mathematician and physicist Joseph Fourier and discussed in his *Analytic Theory of Heat* published in 1822. For Fourier, its discovery was almost incidental - he was really interested in exploring a mathematical description of how heat flowed in solids, and the Fourier transform was just a tool he developed in order to get solutions to his equations. Fourier's mathematical hunch was the following trick: take any periodic function (i.e. a function which repeats itself after going through a certain period). Then, this function can be expressed as a (possibly infinite) sum of sines and cosines[1]. For example, take a look at the function in Fig. 2.

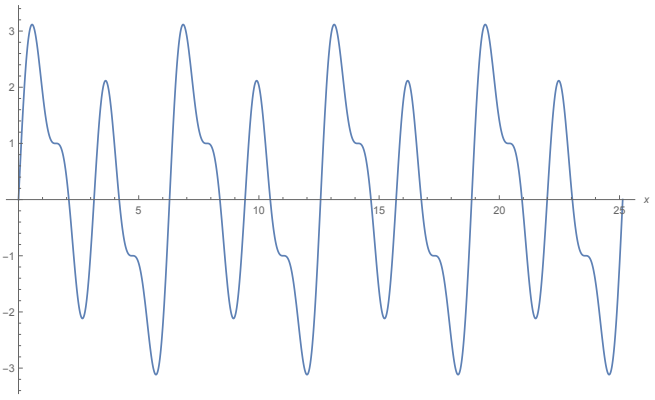

Figure 2: An example of a periodic function that isn't just a single sine wave.

[1]Further inspection by other mathematicians such as Lagrange and Dirichlet show that there are a few caveats to this statement, but we won't go into the details here.

While it may look completely unlike any well-behaved sine curve, this function is actually $f(x) = \sin(x) + 2\sin(2x) + \sin(4x)$, and so can be written as a sum of three sine terms (and no cosines). The Fourier transform takes a periodic function, and converts it to a "bar chart" representing the amount of each frequency sine wave that is used to construct the function. So, the output of the Fourier transform for the function above is the bar chart shown in Fig. 3.

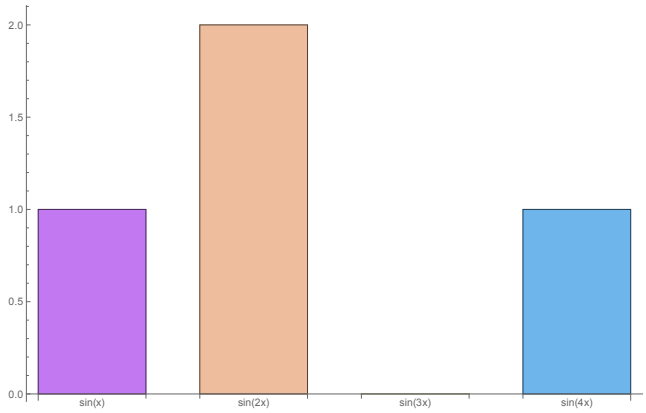

Figure 3: The Fourier components of the function in Fig. 2.

We're going to do something slightly simpler than the cases Fourier was worried about, just for the sake of clarity. Here's our scenario: we're going to imagine working for a music company whose product is a piece of software that records some sound and then tries to match it to an entry in an existing database of songs. However, the company is really new and so the only entries in the database are just pure sine waves, each with its own unique period (what, you kids don't listen to pure sine waves these days?). Moreover, the recording device your boss has given you is 200 years old and only records the amplitude of the sound at, say, 16 discrete time steps over a 10 second window. After this, you run out of memory, tape or whatever the output of your ancient sound recorder is (or maybe the whole thing explodes). In short, if x is the time step and $f(x)$ is the amplitude of the sound you might get an output that looks like this:

x	0	1	2	3	4	5	6	7
$f(x)$	0	2	3	1	0	2	3	1
x	8	9	10	11	12	13	14	15
$f(x)$	0	2	3	1	0	2	3	1

Table 2: An example output from a 200 year old recording device.

Here's a plot of the sound recording, just in case you're sceptical that it is periodic:

`http://www.wolframalpha.com/input/?i=bar+chart+%7B0,1,3,2,0,1,3,2,0,1,3,2,0,1,3,2%7D`

The aim of the game is to extract the period of $f(x)$ (i.e. the number of intervals you need to step through before the amplitudes repeat themselves), in order to match it up to an entry in the database. For the simple examples here, you can probably just read off what the period is:

Task 4.1**:** What is the period of the example output, in Table 2?

We'll step through the quantum algorithm for extracting the period in these simple cases, with the promise that the algorithm is more broadly applicable.

The first step of the algorithm is to set up the quantum computer to work like the recording device; we want to encode a superposition state that has information about both the time steps x and the amplitudes recorded for them, $f(x)$. We'll be switching back and forth between decimal and binary in this section, so just to be clear - if a state is written like $|2\rangle\,|3\rangle$, we actually mean preparing the qubits in the equivalent state in binary: $|2\rangle\,|3\rangle = |10\rangle\,|11\rangle = |1011\rangle$. It's just a change of notation, and nothing else - everything works just as well in decimal, but it makes the explanation of the algorithm in this section neater. Now, when we take a recording, we want to end up with the following state in our quantum computer:

$$|\psi_{input}\rangle = \frac{1}{\sqrt{15}} \sum_{x=0}^{15} |x\rangle\,|f(x)\rangle\,.$$

Task 4.2**:** Write down what this state is for the example output, using the decimal notation.

The second step is to measure the second set of qubits (the ones that encode $f(x)$ - we'll leave the set that encodes x for now). For the sake of argument, let's say we measured the second set and got all zeroes except for the qubit in the second-to-last position (i.e. the second set of qubits is measured to be $|2\rangle$ in decimal, or $|000\ldots10\rangle$ in binary).

Task 4.3**:** Convince yourself that you're left with the following state:

$$|\psi\rangle = \frac{1}{2}[|1\rangle + |5\rangle + |9\rangle + |13\rangle].$$

What is this state, converted back into binary? Can you think of a way to write a circuit to generate this state (don't look below!)?

Here's some code to generate the state above, where we've added a line that changes the colour of each amplitude based on its phase just to make things easier to view:

```
// Exploring a QFT
qc_options.color_by_phase = true;
var bits = 4;
qc.reset(bits);
var qreg = qint.new(bits, qreg);

// Generate the input state
qc.codeLabel('input');
qreg.write(0);
qreg.hadamard(4|8);
qreg.not(1);
```

You should notice that the state we've chosen implictly has some information about the period tucked away inside it - its the gap between all the amplitudes represented in the state.

Task 4.4**:** Suppose we'd measured the $|f(x)\rangle$ qubits and got a different outcome (say, $|0\rangle$). What would the state be now? Would the encoded information about the period still be the same?

So, how do we extract this hidden information about the period? The answer is the *quantum Fourier transform* (or "QFT"). Luckily, there is already a function built into QCEngine to do the Fourier transform for you - here it is tacked onto the code we used to generate our state:

```
// Exploring a QFT
qc_options.color_by_phase = true;
var bits = 4;
qc.reset(bits);
var qreg = qint.new(bits, qreg);

// Generate the input state
qc.codeLabel('input');
qreg.write(0);
qreg.hadamard(4|8);
qreg.not(1);

// Do the QFT
qc.codeLabel('QFT');
qreg.invQFT();
qreg.reverseBits();
```

In principle, the QFT should take the state with the period encoded, and spit out what it thinks the period is.

Task 4.5**:** Implement the circuit above. What is the output, for the state we've put in? You should see that the output state has more than one guess for the period - does this make sense? Why?

This is the entire algorithm for extracting the period from the output of your recording device, which should keep your bosses at the music company happy.

The circuit for the QFT was first written down by computer scientist Peter Shor in 1994. However, folding it into the problem written above, where we're comparing the Fourier transform of a recording with a database of sounds that we already know, is much newer. In fact, the problem has just celebrated its first birthday - the first peer-review paper on this problem (which goes by the technical name "Forrelation" - a portmanteau of "Fourier" and "correlation") was presented at a conference by quantum computing researchers Scott Aaronson and Andris Ambainis in July 2015. This means we've gone from the algorithms explored right at the dawn of quantum computing, right up to the cutting-edge research being explored today. Congratulations!

Extension Task 1**:** Have a go at implementing the algorithm for your own, different recording, $g(x)$. Keep x running from 0 to 15, to keep things simpler. There are only two details you need to make sure the algorithm doesn't go pear-shaped: (1) make $g(x)$ periodic (i.e. $g(x+r) = g(x)$ for a period r) and (2) $g(x)$ can't have repeated amplitudes within a single period (so for example, repeating the pattern 021,021,021 is fine but 101,101,101 is not).

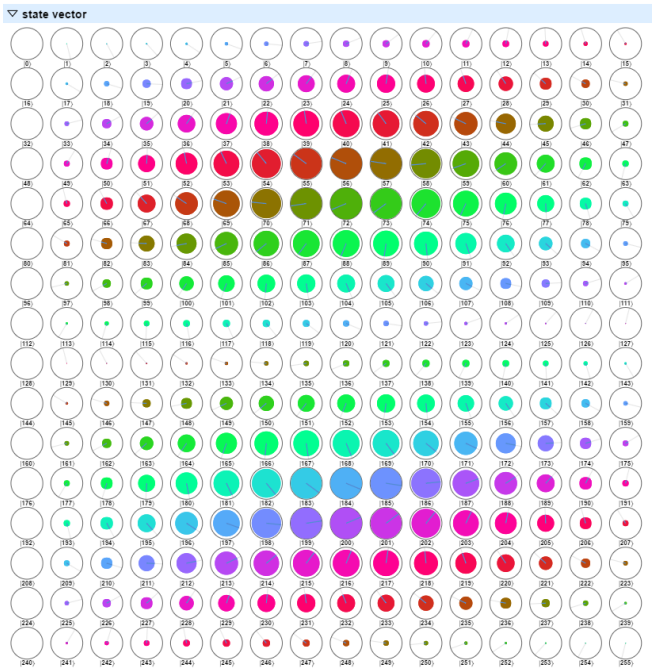

Figure 4: Pretty.

One final thing - the QFT is really just a bunch of quantum gates put together, and it'll spit out some output for any input that you feed in. These inputs don't necessarily need to be representative of some periodic function, like we wrote above; they just need to be some valid quantum state. This is useful for one really important, technical reason: because you can make pretty pictures with it. For example, here's some code that just expands the code above to include more qubits, and fiddles around with the input state:

```
// Exploring a QFT
qc_options.color_by_phase = true;

var bits = 8;
qc.reset(bits);
var qreg = qint.new(bits, 'qreg');

// Generate the input state
qc.codeLabel('input');
qreg.write(0);
qreg.not(1|2|4|8|16|32|64|128);
qreg.hadamard(2|16);

// Do the QFT
qc.codeLabel('QFT');
qreg.invQFT();
qreg.reverseBits();
```

The output state to this code is shown in Fig. 4.

Extension Task 2**:** By fiddling around with the input state for the code above, have a go at creating some pretty outputs like the one shown here.