Intro to Quantum Error Correction

Ben Criger

July 23, 2016

1 Introduction

The purpose of this note is to help introduce quantum error correction [QEC] and (especially) fault-tolerance to people who are not familiar with quantum mechanics, or quantum computing, such as:

- undergraduates,
- electrical/computer/software engineers,
- experimental physicists now trying to implement QEC, who have missed such an introduction.

My goals in writing this note are:

- to keep it short, at the cost of completeness,
- to refer to longer/superior works where possible (often),
- to frequently compare "scary quantum" processes with their classical equivalents, to help de-mystify QEC.

[Insert usual caveat about imprecise notation, i.e.: these notes are free, and you get what you pay for.]

I begin in the following section with an explanation of why error correction is necessary, moving on to describe the difference between error-correction and fault-tolerance, before moving on to the most popular elements of QEC theory (stabilizer codes $\mapsto$ CSS codes $\mapsto$ homological codes $\mapsto$ the surface code).

2 Why Error Correction?

If you meet a software engineer in the street, the need for error correction may not be obvious. In the life of a computer programmer, errors are things caused exclusively by people, and careful thinking and good practices are required to prevent, detect and correct them. [Maybe there's a repetition code at work here, especially with `diff`, but that is an analogy for another day.] However, if you scratch the surface, you find instances of error correction and even (what I'm going to call) fault-tolerance at work.

Take, for example, satellite/spacecraft communications. These spacecraft are very far away, and bits don't make it between the spacecraft and the ground station. Actually, that example is too far outside our daily experience to take, so let's take flash drives, AKA USB keys. The job of these drives is to store bits (0's and 1's), which they manage to do most of the time. However, one bit in every 10^{4-9} is randomly flipped, which in the worst case scenario would lead to an ASCII typo per page of text [or something].

To compensate for this, the USB key records the information redundantly, using more bits than are stricly necessary. When reading, the CPU can then check that different decodings of the redundant information are consistent, and if they're not, can infer a decoding that's likely to be correct. The fact that the CPU is not itself (assumed to be) error-prone is a great boon to us in reading USB keys, otherwise we couldn't trust that any decoding was correct.

There's a reason that CPUs are not considered noisy, by the way. Transistor-transistor logic (TTL for short) defines a standard voltage for its 0 and 1 states, usually $0\,\mathrm{V}$ and $5\,\mathrm{V}$, respectively. These circuits can be designed to output $0\,\mathrm{V}$ for any input voltage between $0\,\mathrm{V}$ and $0.4\,\mathrm{V}$, likewise to output $5\,\mathrm{V}$ whenever a voltage of $2.6\,\mathrm{V}$ or higher is input. This is a (very powerful) error correction mechanism. If noise spreads voltages out according to some unimodal distribution, we can tighten that distribution at every TTL element, paying only a small energy cost to do so. In theory, we could also design an element which outputs a third value whenever the input voltage doesn't correspond to a logical 0 or 1, so that we can detect errors. In practice, we would just increase the supply voltage until we don't have to worry.

2.1 Fault Tolerance

We will aim to replicate this property of TTL logic for quantum states. TTL logic takes a continuous set of values (voltages) and uses them to represent a discrete logical set (bits). This trick has been replicated in QEC, in 'qubit-in-oscillator' codes. While these codes appear simple, and perhaps useful, they're not very popular, so we're going to take large sets of qubits as our input space.

If we want to change the input space, we should have a general idea of how error correction works, so that we can determine which codes work, and how well. Let's define a code to be [blah blah blah, n, k, d].

[Small voltage shifts = correctable errors, large voltage shifts = uncorrectable, even larger voltage shifts = undetectable]

[Knill-Laflamme]

This leaves us with a bit of a problem. Qubit quantum states are represented by complex length-2 vectors with unit norm. However, in order to represent an arbitrary state of two qubits, we need to use tensor products (AKA Kronecker products for our purposes) of these vectors, and the tensor product of an length-a vector and a length-b vector has length $a \times b$. For 2-3 qubits, analytic results can still (sometimes) be obtained, but calculations involving these vectors don't stay doable after that. We'll need to find a family of n-qubit quantum states that can be efficiently represented (i.e. adding a qubit doesn't multiply the size of the vector, or other object we have to keep in memory).

3 The Stabilizer Formalism in a Nutshell

There is at least one well-studied way to do this (there are a few others as well, but they're not as popular). This theory uses operators instead of quantum states as its representation, because tensor-product operators can give rise to states with non-trivial entanglement, which it turns out is necessary to do error correction [Prove this?]. This is the so-called *Heisenberg picture*.

It's also best to use operators that form a group to do this, that way we don't have to keep track of any matrices during a calculation, we can just use the group operation. Our operators have $+1$ and -1 eigenspaces of equal size [Prove this]. This means that, if we want to specify an $n-1$-qubit subspace of a space, we only have to write down one n-qubit operator, given as an $\mathcal{O}(n)$-letter string. This means that, in order to define a subspace with k "logical" or "effective" qubits, we specify $n-k$ stabilisers, using a total of $\mathcal{O}(n^2)$ letters (assuming k doesn't scale with n).

It remains to be seen that subspaces defined in this way can detect/correct errors. Let's show that Pauli errors can be detected if they anticommute with some stabiliser, and that a set can be corrected if any two such errors don't form a logical when you multiply them. [Go ahead and show that].

3.1 CSS codes

These are like easy stabiliser codes.

3.2 Homological Codes

These are like easy CSS codes.

3.3 The Surface Code

This is like an easy homological code.