Intro to Quantum Error Correction

Ben Criger

October 24, 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 there are bits that 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 strictly 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.

The availability of reliable components which can be used to store and process classical information is a great benefit to computer engineers. In the following sections, we'll see which of these properties can also be achieved for quantum systems. But first, some definitions.

2.1 What is a Code?

A code is a subset.

If you have a system, classical quantum or other, which can assume states from some set, you can define any subset of these states to be a code. The states in the code are called the *codewords*.

Most of the time, we will also constrain our codes to be *linear* with respect to some operation (usually denoted $+$), which means that for any codewords a and b, $a + b$ is also a codeword. This is pretty common, and it means we can normally consider state sets which are vector spaces, and codes which are subspaces. This is true in both the classical and quantum case.

A code can be used to fulfil a variety of purposes. Compression, for example, involves codes whose codewords are shorter than the logical messages they represent.

We will mostly concern ourselves with *error-correcting* codes, which can only be analysed using two other mathematical objects as tools. The first of these is the *error model*, the set of possible transformations which can act on a state (codeword or otherwise) without our knowledge. The second is the *recovery map*, which takes as input states from the full set and outputs corresponding codewords. To solidify these ideas, let's consider an example.

2.1.1 Example: 3-Bit Repetition Code

Suppose Alice wants to tell Bob the value of a bit, 0 or 1, but that she has to use a channel which is subject to symmetric bit-flip:

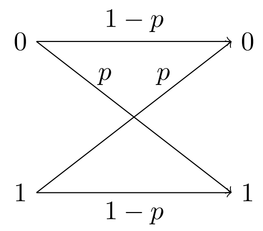

Figure 1: Graphical representation of an error map on a classical bit which flips the bit (taking 0 to 1 and vice versa) with probability p.

Alice, then, would like to send messages from $\mathbb{Z}_2$, so she should design codes that have two codewords. Supposing that Alice has repeated access to the channel, and knows the probability p, she can design codes which use $\mathbb{Z}_2^n$ as the full state space. A simple thing that Alice can do is repeat herself three times (taking $n = 3$) to make herself understood. Bob, then, should infer that 0 is the desired message when he receives a state containing mostly 0s and 1 when he receives a state containing mostly 1s. Let's express these ideas mathematically, so that we can generalize later.

Codewords : $000, 111$

Error Model :

$$000 \mapsto \begin{array}{cl} 000 & (1-p)^3 \\ 001,\,010,\,100 & p(1-p)^2 \\ 110,\,101,\,011 & p^2(1-p) \\ 111 & p^3 \end{array} \qquad 111 \mapsto \begin{array}{cl} 111 & (1-p)^3 \\ 110,\,101,\,011 & p(1-p)^2 \\ 001,\,010,\,100 & p^2(1-p) \\ 000 & p^3 \end{array} \tag{1}$$

Recovery Map : $(000, 001, 010, 100) \mapsto 000$, $(111, 110, 101, 011) \mapsto 111$

We can see that, if there is a weight-two or weight-three error, Bob will infer the incorrect state. The resulting probability of error is $3p^2(1-p) + p^3$.

Exercise: When does this protocol amplify errors? How should we remedy this?

We will go on to construct quantum codes in later sections. But first, I should confess something.

2.2 Fault Tolerance

Let's begin with an overall reminder of what we're trying to accomplish. We want to replicate the properties of TTL logic for quantum states and operations. Not only are we able to interpret the output of a TTL gate using a perfect computer, we also know that TTL operations suppress errors that occur *during the operations themselves*. We therefore have to *encode operations* in some way, using a small subset of operations drawn from some large set to ensure that errors from the operations used to correct the code can themselves be corrected. For the first few examples, we won't focus on this, since it's useful to see that we can correct 'vanilla' errors before moving on to attempt fault tolerance.

3 Quantum Codes

If fault tolerance wasn't a big enough problem, we have another one. Error models for quantum systems are much more complex than those for classical bits, because they're *continuous*. They don't map $|\psi\rangle$ to $|\psi_\perp\rangle$

4 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].

4.1 CSS codes

These are like easy stabiliser codes.

4.2 Homological Codes

These are like easy CSS codes.

4.3 The Surface Code

This is like an easy homological code.