Quantum Error Correction

2021 QCIPU Summer School

Scott Lawrence[*]

21-22 July 2021

Real-world quantum computers, like real-world classical computers, are built from faulty machinery. Unlike classical computers, however, the errors that occur on quantum computers are relatively large and frequent. Quantum errors can be alleviated by using multiple physical qubits to encode a single logical qubit. Such a construction is referred to as a *quantum error-correcting code* (QEC). QECs are the main topic of these lectures. At the end, we will discuss the *threshold theorem*: when the error rate of physical qubits is below a certain threshold, QECs can be used to design logical qubits with arbitrarily low error rates.

These are lecture notes for two one-hour lectures delivered at the Fermilab-run QCIPU summer school in 2021. They are intended as a non-rigorous invitation to quantum error correction and fault tolerance, assuming minimal background in quantum mechanics and computer science. The first lecture is to consist of the first and second sections; the second lecture continues with the third section and introduces the all-important threshold theorem in the fourth section.

These notes assume familiarity with various parts of linear algebra, and a bit of understanding of how quantum computation works. For completeness, a review of linear algebra and density matrices is given in Appendix A.

These notes include some content, including many examples, not covered in the video lectures. This extra content is included as appendices within each section (e.g. 1.A).

[*]scott.lawrence-1@colorado.edu

1. Classical Error Correction

Anyone can make mistakes, but only an idiot fails to correct them.

(Cicero)

While performing some computation, I need to store the information I'm working with. Often, this storage is not perfect: if I store a 0, when I go to retrieve it, I might find that the bit has been flipped to a 1. The physical mechanism for how this happens (thermal noise? a cosmic ray?) isn't important right now. Let's just assume that there's some probablity p that the bit I store gets flipped[1]. What implications does this have for the success of my computation?

If my computation involves only this one bit, then the probability that the result is corrupted is just p — or in other words, the probability that the computation proceeds without error is $(1-p)$. If there are N bits, however, the computation is uncorrupted with probability only $(1-p)^N$. This can be quite small! The probability of success decays exponentially with the number of bits.

For concreteness, let's say that $p \approx 0.01$, and I want to do a computation with $N = 100$ bits. The probability of each individual bit flipping is quite small — seemingly negligible — but the whole computation proceeds uncorrupted only 37% of the time. This is an unacceptable rate of errors, so we must reduce p.

Ordinarily, we might try to reduce p by lowering the temperature, shielding the processor from cosmic rays, and banning the graduate students from eating in the lab. These are all sensible measures, but in practice there comes a point where lowering p by physical means is just too difficult or expensive. Let's say we're stuck with $p \approx 0.01$, and can't afford to (or figure out how to) lower it by better engineering. It turns out there's a way to do the computation anyway.

In the case where we only need to encode one bit, let's imagine that we made a duplicate. So, if we need to store a 0, we'll store 00 on two physical bits, and if we need to store a 1, we'll store 11 on two physical bits. In this way, two *physical* bits are combined to form a single *logical* bit.

At first glance, this doesn't do much good — before, the probability of error was p, but now it's nearly twice that: $1-(1-p)^2 \approx 2p$. However, if we go to read the logical bit and discover that the physical bits are in the configuration `01` or `10`, then we know that an error has occured! It's much better to know that an error has occured than to blithely accept an

[1]In your computer, the probablity of such an error occuring on any given day, on any given bit, is something like $\sim 10^{-10}$. In other words, bit-flips happen, but require some effort to detect.

incorrect computation. We will only be ignorant of the error when *both* bits are flipped. This is much less likely, with probability p^2.

In the case of N bits, this error *detection* scheme has a probability $(1-2p)^N$ of producing the correct answer, a probability of $2Np+O(p^2)$ of detecting an error, and a probablity of $Np^2+O(p^3)$ of yielding a corrupted answer without reporting an error. (The "big-O" notation $O(p^2)$ indicates that terms quadratic in p, or of higher order, have been dropped. When p is small this is a good approximation.) The probability of getting the right answer without trouble has been reduced, but the probability of getting the wrong answer has fallen much more steeply, so this is a net gain.

The shortcomings of this method are due to the fact that errors can be detected, but never *corrected*. We can fix this: instead of 2 physical bits, we'll use 3. The logical bit 1 will be ideally encoded as 111, but physical bitstrings 110, 101, and 011 will also be accepted as corresponding to a logical 1. Similarly, the logical bit 0 is ideally encoded as 000, but any majority-0 physical bitstring is accepted as a logical 0.

Now, there are three times as many chances for bitflip errors, but *two* bitflips (in the same codeword) are required before the error actually matters. As a result, we can calculate the probability of a corrupted computation: $3Np^2$. For small p, this is a dramatic improvement over the original probability of $\sim Np$.

This *triple repetition code* is an example of the general idea of an error-correcting code. By using multiple physical (qu)bits to encode a single logical (qu)bit, the effective probability of errors is squared or better. There may be small factors of 2 or 3 (or 9) introduced, but when the original probability of physical errors is small, these factors will be drowned out by the fact that two or more physical errors are required to produce a single logical error.

1.1. Hamming Codes

Hamming codes are simple, but effective enough to be practical. If you buy RAM and see it labelled "ECC", Hamming codes are probably being used.

In the triple correction code, three physical bits were used to encode a single logical bit. We could reduce the error rate further, from p^2 to p^3 or even lower, by using five or more bits per codeword. Such schemes are very inefficient — to do better, we will work with blocks of more than one logical bit. For the purposes of error correction, each block can be treated as a single entity; we will see that this allows far more efficient schemes.

To understand Hamming codes, we first need the notion of Hamming distance. The *Hamming distance* between two binary strings of equal length is the minimum number of

$$\begin{aligned} \texttt{00} &\mapsto \texttt{000} \\ \texttt{01} &\mapsto \texttt{011} \\ \texttt{10} &\mapsto \texttt{101} \\ \texttt{11} &\mapsto \texttt{110} \end{aligned}$$

Figure 1: Codewords for `Hamming(3,2)`.

bits that must be flipped to get from one to the other. As an example, `0110` and `0100` have a Hamming distance of just 1. Notice that Hamming distance corresponds exactly to our error model: if the distance between two strings is d, then the probability of one being transformed, by bitflip errors, to the other is $\sim p^d$.

Central to Hamming codes is the notion of a *parity bit*. Let's say we want to transmit two bits, and then detect any bitflip error. We could always just transfer the same two bits twice, doubling the length of the message: any single bitflip will be detected by the fact that the first message is not the same as the second. However, we can do much better by simply appending a parity bit — a bit that denotes whether the number of `1`s in the message was even or odd. The four codewords are shown in Figure 1. Note that the minimum Hamming distance between any two codewords is 2, indicating that at least 2 bitflips are required to obtain an undetectable error.

Now we're ready to move to constructing more sophisticated codes. We'll begin with one for encoding 4 logical bits via 7 physical bits, often referred to as `Hamming(7,4)`. This code will have three parity-check bits; one acting on bits $1,2,4$, one on bits $1,3,4$, and one on bits $2,3,4$. The other four bits in a codeword will be unchanged from the logical bits we're seeking to encode.

This code is concisely represented by this *code generating matrix*:

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

A column of G corresponds to a bit of the codeword; a row corresponds to one of the 4 bits we're seeking to encode. Each column can be thought of as specifying the list of bits from which the parity check is computed. Physical bits $3,5,6,7$ contain the original message, and bits $1,2,4$ are the parity-check bits. Symbolically, we have $c \equiv G^T m \mod 2$, where m is a vector of bits in the message, and c is the codeword[2].

Another way of thinking about the structure of the parity-check is as a list of constraints on the set of valid codewords. For instance, you can check for yourself that $w_1 + w_3 + w_5 + w_7 \equiv 0 \mod 2$ for any uncorrupted codeword. Because there are 7 bits in a codeword and only 4 bits in a message, there are 3 such constraints we can write down, summarized by $Hw \equiv 0 \mod 2$, where

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Note that `Hamming(7,4)` compares very favorably to the triple repetition code. Both have the property that single bitflip errors are corrected, but the triple repetition code triples the length of the message, where the Hamming code does not even double it.

A Hamming code is a particular sort of linear code. In general, a linear code is specified by a code generator matrix G (which tells us which parity bits are based on which inputs), from which we can obtain a parity-check matrix H (which mandates linear relations that must hold true for any codeword).

The parity-check matrix given for `Hamming(7,4)` has a peculiar property: except for `000`, every three-bit string is present as a column! The term "Hamming code" describes any linear code whose parity check matrix has this property. So, for instance, you can see that the triple repetition code introduced at the very beginning of this section is also a Hamming code, with these code generator and parity-check matrices:

$$G = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ and } H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Unsurprisingly, the triple-repetition code also goes by the name `Hamming(3,1)`.

A bit more of the theory of linear codes is developed in Section 1.A; it is not a prerequisite for understand the quantum case.

1.2. Other Types of Errors

So far we've considered the performance of error-correcting codes in the presence of single bit-flip errors. More than one bit might be flipped, but with probability falling exponen-

[2]In fact, if we view G as a matrix in the finite field $\mathbb{F}_2$, then we can just write $c = G^T m$. In this form, Hamming codes on bits can be generalized to codes on trits, and so on.

```
111 000 111 111 000 ...          10110...
111 001 111 110 00 ...           10110...
110 011 111 100 0 ...             1110...
```

Figure 2: The degradation of the message `10110`, coded with the `Hamming(3,1)`, under multiple erasures. The first erasure (in the second codeword) has no effect, but the second (in the first codeword) introduces an effective single-bit erasure into the coded message.

tially with the number to bits to be flipped. If we change this model, then the performance of the ECC changes.

For instance, if (due to a quirk of the hardware) there's a substantial probability for all bits in a codeword to be flipped, the triple repetition code will neither correct nor detect the error. Under the previous error model, this sort event only happened with probabiity p^{3L}, and was completely negligible. The triple repetition code requires that assumption to be negligible.

Perhaps a more realistic change to the error model is to allow "erasures" — that is, the deletion of bits from a stream. This is unlikely if you think about RAM, where each bit has a unique index, but is much more plausible if we consider the case of *communication*, where the receiver might blink and miss a bit. An erasure might change `10110` $\rightarrow$ `1010`; here the receiver blinked exactly in the middle of the transmission.

The triple repetition code does not fare so well in the presence of erasures. Remember that the code can handle single bitfips gracefully, and even multiple bitflips as long as no two occur in the same codeword. It turns out that this code can also handle a single erasure, but not more, no matter how far separated the erasures are. This is illustrated in Figure 2.

We won't consider analogous errors even in the quantum case, but these examples should convince you of an important point: the performance of an error correcting code depends strongly on the types of errors to which it will be subjected. In fact, there's no such thing as a completely general code: for *any* code, there exists a class of errors that the code does not reduce. This is true in the quantum case as well.

1.A. Theory of Linear Codes

The Hamming codes described above are a specific type of *linear code*. Linear codes can be defined for any finite field $\mathbb{F}_q$, but we'll remain specialized to the case $\mathbb{F}_2$ (binary linear codes) for simplicity.

A field is a set equipped with two operations: addition and multiplication. They must obey all the usual rules: addition and multiplication are commutative, and multiplication distributes over addition. Additionally, there are two special elements. The additive identity (0) has the property that $0+a=a$ for any a, and the multiplicative identity (1) similarly obeys $1 \cdot a = a$. Finally, for every a, there must be an additive inverse $-a$ and (if $a \neq 0$) a multplicative inverse a^{-1}, so that $a+(-a)=0$ and $aa^{-1}=1$.

The real numbers are the most familiar field, the complex numbers a close competitor. We can construct fields that have a finite number of elements, and the simplest non-trivial case is $\mathbb{F}_2$, the finite field with two elements. The elements are $\{0,1\}$. Addition corresponds to logical `XOR`; in particular, $1+1=0$. Multiplication corresponds to logical `AND`.

Linear algebra makes sense over any field, not just the reals or complexes. As an example, in $\mathbb{F}_2$, we have

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Note that matrix coefficients as well as the vector are valued in the field $\mathbb{F}_2$.

A codeword is a sequence of n elements of $\mathbb{F}_2$ (bits), which once decoded will represent k logical values (also elements of the field[3]). In the triple repetition code above, we had $n=3$ and $k=1$, but any values are possible so long as $n \geq k$. In the case $n=k$, of course, no error correcting can be done. In general, n is termed the *length*, and k the *rank*.

The defining feature of a linear code is that every linear combination of codewords is also a codeword. This idea only makes sense because we are working in a finite field; otherwise, this requirement would imply infinitely many codewords! We can quickly verify that the triple repetition code is in fact linear. The code is spanned by $(0;0;0)$ and $(1;1;1)$; because the only coefficients available are 0 and 1, no other vectors can be obtained by multiplication. The last possibility is adding these vectors, which of course yields the second again.

We described `Hamming(7,4)` in terms of generator and check matrices. These are general objects that can be constructed for any linear code. Remember that every linear combination of codewords is also a codeword; in other words, the set of codewords forms a

[3]This means that there are 2^k codewords. In the general case of $\mathbb{F}_q$, there are of course q^k codewords.

(k-dimensional) linear subspace of $\mathbb{F}_q^n$. The generator matrix G is nothing but a list of k basis vectors. The bits (or q-dits in general) serve as coefficients of these k vectors; the vector $G^T m$ is the encoding of a message m (itself a vector with k components).

The check matrix H is a list of $n-k$ basis vectors for the orthogonal subspace $\mathbb{F}_q^{n-k}$. The inner product between a valid codeword and any vector in this orthogonal subspace must be 0; thus, computing Hv tells us immediately whether v is a valid codeword.

The Hamming distance between two vectors is the number of elements that are different; equivalently, the distance $d(u,v)$ between vectors u and v is the number of non-zero elements in $u-v$. In a typical linear code, many Hamming distances are possible. For instance, in the `Hamming(7,4)` code described above, three valid codewords are

$$\begin{pmatrix}0\\0\\0\\0\\0\\0\\0\end{pmatrix}, \begin{pmatrix}1\\1\\1\\0\\0\\0\\0\end{pmatrix}, \text{and} \begin{pmatrix}1\\1\\1\\1\\1\\1\\1\end{pmatrix}. \tag{1}$$

The first two have a Hamming distance of 3, and the first and third have a Hamming distance of 7.

Given a codeword v, we can ask what the minimum Hamming distance is to a distinct codeword. For a linear code, this does not depend on choice of v! To see this, first select two codewords u_0, v_0 that have a globally minimal Hamming distance; i.e. are as close as any two codewords in the code can be. Now let v be some other codeword. Since the code is linear, $v - v_0 + u_0 \equiv u$ is a valid codeword. The distance between this new pair is $d(u,v) = d(u_0,v_0)$, demonstrating that v (an arbitrary codeword) has the same minimum Hamming distance as any other codeword. As a result of the codeword-independence of the minimum Hamming distance, the quality of a linear code does not depend on what data is being encoded.

With a minimum Hamming distance of d, how many bit-flips can be corrected? We can of course *detect* up to $d-1$ flips, but this is not the same as being able to correct them. For instance, in the case of the triple repetition code, $d-1=2$ flips will result in the error being ‘corrected’ to the wrong value. The number of correctable errors is half the minimum distance, $\lfloor \frac{n-1}{2} \rfloor$.

Early on in this section, we noted that with $n=k$, no error correction — or even error detection — can be performed. Since the Hamming distance quantifies the level of protection received from a code, it comes as no surprise that there is a general bound relating n, k, and d:

$$k+d \leq n+1 \tag{2}$$

A code that saturates this bound is in some sense optimal, at least among codes with that value of k and n. The triple repetition code has $n=3$, $k=1$, and $d=3$, and therefore saturates this bound. This notion of optimiality does not account for efficiency, that is, the ratio $\frac{n}{k}$.

2. Quantum Errors

'To err is human,' but a human error is nothing to what a computer can do if it tries.

(Agatha Christie)

It's tempting to try to apply the classical codes of the previous section directly to quantum computers. Unfortunately, even the simple triple repetition code fails in the face of reasonable quantum errors. Suppose we wish to transmit the state[4] $|\psi\rangle = |0\rangle + |1\rangle$. This state is often refered to as $|+\rangle$, and is an eigenstate of σ_x with eigenvalue 1. It is the equivalent of the state $|0\rangle$, but in the X-basis. Under the triple repetition code[5], this is mapped to

$$|\Psi\rangle = |000\rangle + |111\rangle.$$

A bitflip, which the triple repetition code is designed to be able to handle, is implemented by the operator σ_x acting on a single qubit. In this case, we can see that

$$\sigma_x(2)|\Psi\rangle = |010\rangle + |101\rangle.$$

When all qubits are measured in Z-basis, it will not be difficult to automatically repair this error, as expected. However, this is not the only sort of error that could occur. Consider for instance the action of σ_z on a single qubit. Now we have

$$\sigma_z(2)|\Psi\rangle = |000\rangle - |111\rangle.$$

Decoded, this yields $|0\rangle - |1\rangle = |-\rangle$ — the triple repetition code does nothing to protect us from this sort of error.

In this section we will define in detail what sorts of errors we can hope to protect against. Error correcting codes that are effective against these sorts of errors are discussed in the final two sections.

2.1. Examples of Errors

We have already seen one example of a quantum error that can occur: the bitflip. When a bitflip is performed, the state transforms as $\Psi \mapsto \sigma_x|\Psi\rangle$, and so the density matrix transforms as $\rho \mapsto \sigma_x\rho\sigma_x$. Therefore, in the language of density matrices, the bitflip error is

[4] Here and throughout, I will work with non-normalized states.

[5] You may reasonably object that this is not the most natural generalization of the triple repetition code. One could instead map $|\Psi\rangle \mapsto |\Psi\rangle \otimes |\Psi\rangle \otimes |\Psi\rangle$. Unfortunately, by the no-cloning theorem, no unitary operation can implement the encoding!

superoperator defined as

$$\mathscr{E}_{\mathrm{bf},p}(\rho) = (1-p)\rho + p\sigma_x\rho\sigma_x.$$

This denotes exactly the event "the bitflip operator σ_x is applied to the qubit with probability p".

Another option is the phase flip, again seen above. The only change is that σ_z is applied instead of σ_x:

$$\mathscr{E}_{\mathrm{pf},p}(\rho) = (1-p)\rho + p\sigma_z\rho\sigma_z.$$

The same idea extends to σ_y. In fact, we can be slightly more general. The Pauli matrices $\sigma_x, \sigma_y, \sigma_z$ are all elements of the group $SU(2)$, and any element $U \in SU(2)$ can be used in their place to obtain a sensible error operation (although note that if U is the identity element, the error is not concerning!).

$$\mathscr{E}_{U,p}(\rho) = (1-p)\rho + pU\rho U.$$

It's reasonable to imagine that designing a code that protects against all possible errors $\mathscr{E}_U$ will be difficult. However, note that any element of $SU(2)$ — that is, any 2×2 unitary matrix of unit determinant— can be written as a linear combination of the Pauli matrices and the identity matrix[6]:

$$U = \alpha + \vec{\beta}\cdot\vec{\sigma}.$$

Then the most general possible error operation along these lines is

$$\mathscr{E}_U(\rho) = c_{00}\rho + c_{0x}\rho\sigma_x + \ldots + c_{xx}\sigma_x\rho\sigma_x + c_{xy}\sigma_x\rho\sigma_y + \ldots = \sum_{i,j} c_{ij}\sigma_i\rho\sigma_j$$

with some coefficients c_{ij} determined from $\alpha, \vec{\beta}$. (For notational convenience, let σ_0 be the identity matrix.)

This is remarkably convenient! It means that in order to verify that an error code is effective against a continuum of possible errors (in this case $SU(2)$), we only need to consider a finite number of discrete errors.

One special case is worth particular mention. Arguably the most natural error operation of the form $\mathscr{E}_U$ is the one that is symmetric with respect to exchange of the axes (and in fact, with respect to any $SU(2)$ rotation). This is called the *depolarizing channel*, and has the form

$$\mathscr{E}_{\mathrm{dp},p} = (1-p)\rho + \frac{p}{3}\left[\sigma_x\rho\sigma_x + \sigma_y\rho\sigma_y + \sigma_z\rho\sigma_z\right].$$

[6]Unitarity imposes an additional constraint between α and $\vec{\beta}$, which we won't need to use.

The error operation $\mathscr{E}_U$ is not the most general one-qubit error possible. For instance, it is not necessary for $\mathscr{E}$ to be trace-preserving (you can check for yourself that $\mathscr{E}_U$ always is). For the rest of these notes we will ignore such processes and focus on the case here of errors obtained from the Pauli matrices. However, you should be aware that these other sorts of errors exist, although the analysis methods described in these notes carry over with little or no change.

2.2. Independent Errors

So far, all the errors described affect a single qubit at a time. It's perfectly possible to construct and consider forms of noise that affect two qubits together. However, when the probability of an error on a single qubit is small, it's often reasonable to assume that the probability of a correlated error on two qubits is smaller still, and so this form of noise can be neglected. There are some places when this assumption is definitely invalid. Most obviously, when we apply a two-qubit gate (like the controlled-not gate), the implementation of the gate is likely to be slightly faulty, introducing an error channel that certainly acts in a correlated way on the two qubits. For simplicity, we'll ignore such situations.

This sort of assumption — that noise affects different (qu)bits independently — was critical to the error analysis of classical codes in the previous section. In the case of quantum errors, the precise assumption we will make is that the full error operation $\mathscr{E}(\rho)$ factorizes into a bunch of one-qubit errors. That is, we can write

$$\mathscr{E}(\rho) = [\mathscr{E}_1 \circ \mathscr{E}_2 \circ \cdots \circ \mathscr{E}_N](\rho),$$

where $\mathscr{E}_n$ acts only on the nth qubit.

3. Quantum Error Correction

Now that we have a sensible model for errors on a quantum computer, we can construct reasonable schemes for quantum error correction, and eventually, in the final section, fault-tolerant circuits.

3.1. Three-Qubit Codes

The three-qubit code — the naive generalization of the classical triple repetition code — was introduced in the previous section, and we know it doesn't correct for some errors (like bitflips in X-basis). Nevertheless, it's worth examining in detail how it might be implemented. The ideas behind the implementation generalize well to other quantum codes, and are simplest to see in this low-dimensional case.

In some sense the defining operation of the three-qubit code is the encoding circuit. We cannot simply duplicate the state of the qubit due to the no-cloning theorem (otherwise quantum error correction would be easy!). However, we can define a unitary operation that acts on the basis states $|0\rangle$ and $|1\rangle$ in the expected way, and extends to all others by linearity. This is shown on the left of Figure 3. We have a single qubit, which must be encoded. We bring in two extra qubits, initialized to $|0\rangle$. Two `CNOT` gates are sufficient to rotate these to match (in Z-basis) the original qubit, and the encoding is complete.

Decoding introduces a caveat: we must be careful not to accidentally measure the state of the logical qubit early. This would happen, for instance, if we simply threw out two of the qubits. Instead, we must separate out a single qubit that contains the logical state, and two extra qubits that only have information about what errors occured. Such a procedure is shown on the right of Figure 3. The top qubit contains the decoded message.

It turns out we can also perform error correction without ever decoding. This is very useful, since immediately after decoding, the logical qubit is not error-corrected, and is

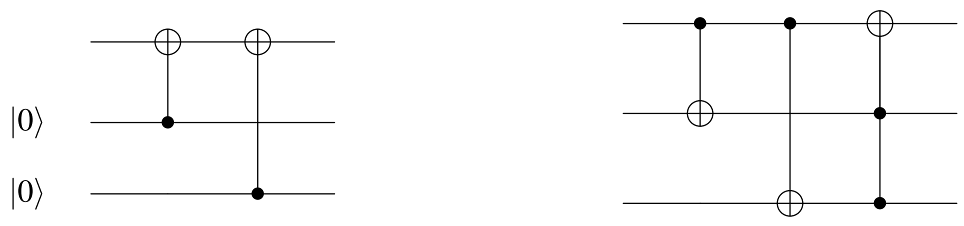

Figure 3: Encoding (left) and decoding (right) circuits for the three-qubit (bit flip) code.

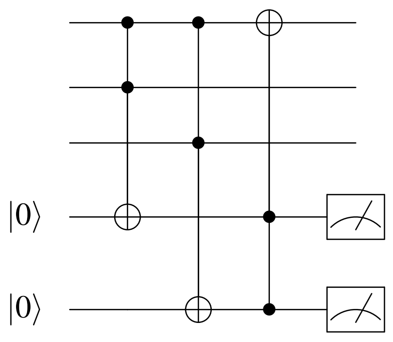

Figure 4: An error correction circuit for the three-qubit (bit flip) code. Note that the entire process is unitary until the two ancilla are discarded.

vulnerable. The core idea comes from looking at the decoding procedure more carefully: we were able to separate our from the three qubits, two qubits that contained only information about what error had occured. If instead of performing a unitary transformation on the original three qubits, we bring in two (or, for convenience, possibly more) ancilla, we can put the information about what error occured (called the *syndrome*) in those ancilla. The ancilla are then used to control a circuit that corrects the error in the original three qubits. After this is complete, the ancilla can be safely measured or discarded. This procedure is shown in Figure 4.

This completes our implementation of the three-qubit code. Before moving on, note that there are actually many possible three-qubit codes. The example above is often called the *three-qubit bit flip code*, as it is designed to prevent bit flips, that is, the operation of σ_x. However, we could just as well design a *three-qubit phase flip code*. This is obtained simply by using the Hadamard gate to swap the X- and Z-axes before encoding. It has the property that it is immune to the operation of σ_z.

3.2. Shor Code

The Shor code extends the three-qubit code to address all errors affecting a single qubit. The encoding procedure is simple. First, a qubit is encoded using the phase-flip code. (To be concrete, encode $|0\rangle \mapsto |+++\rangle$, and $|1\rangle \mapsto |---\rangle$.) This results in three qubits. Each of those three qubits is now encoded with the bit-flip code. This results in nine physical qubits to encode the single original logical qubit. This sort of construct, where two error correcting codes are applied in sequence to achieve an improved code, is known as *concatenation*, and will feature prominently in the next section.

The resulting codewords are:

$$|0\rangle \mapsto (|000\rangle + |111\rangle)^{\otimes 3} \text{ and } |1\rangle \mapsto (|000\rangle - |111\rangle)^{\otimes 3}.$$

The decoding procedure mirrors the encoding procedure. First, each of the three intermediate qubits is decoded using the bit-flip decoding procedure discussed above. This results in three qubits which represent a single logical qubit via the phase-flip code, and that decoding procedure is used.

Now we must show that the Shor code actually works — that it protects against arbitrary one-qubit errors. The easiest one-qubit error to consider is a bit-flip. Because the most recent QEC in the concatenation was the three qubit bit-flip code, we immediately know that we are protected from single bit-flips.

Now we'll check phase-flips. Suppose $|0\rangle$ is encoded, and the first qubit is hit by σ_z. The state is now $(|000\rangle - |111\rangle) \otimes (|000\rangle + |111\rangle)^{\otimes 2}$. The decoding takes place in two steps. After the first step, we will have $|-++\rangle$, which the second step will correctly transform to $|0\rangle$. Similarly, you can readily check that the action of σ_y does not change the result after decoding, as long as only one qubit is hit.

3.3. Steane Code

The Shor code uses nine physical qubits to encode a single logical qubit; this section describes a slightly less unwieldy code that uses only seven qubits for the same purpose.

The Steane code can be defined by the encodings

$$\begin{aligned}|0\rangle \to &|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle\\ &+|0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle, \text{ and}\\ |1\rangle \to &|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle\\ &+|1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle.\end{aligned}$$

That this code is robust against both bit-flip and phase-flip errors is a matter that can be verified through straightforward, but tedious, algebra. Rather than trace that calculation, let us see how the Steane code is derived.

The Steane code is derived from `Hamming(7,4)`. Recall the generating matrix for that code:

$$G^T = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The $2^4 = 16$ valid codewords are obtained by multiplying $G^T m$ for any $\{0,1\}$-vector m. We could, of course, construct a quantum code according to the rule $|m\rangle \to |G^T m\rangle$; however, as with the three-qubit bit-flip code, we would be protected from bit flips but not phase flips.

Instead, let's introduce one more classical code, the *dual code* of `Hamming(7,4)`. This code has 8 codewords and is defined by having a generating matrix equal to the parity check matrix of the original code:

$$G_D = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

This code is termed the dual code because each codeword in G_D is orthogonal to each codeword in G: $G_D G^T = 0$. Moreover, G_D is the largest such code with this property. Every codeword which is orthogonal to G is a valid codeword generated by G_D.

In this case, every codeword of the dual code is also a valid codeword of the original code. You can prove this easily by verifying that every row of G_D can be obtained as a linear combination of rows of G.

For any valid codeword x in `Hamming(7,4)`, we can define a quantum state

$$|\bar{x}\rangle \equiv \sum_y |x+y\rangle.$$

where the sum runs over all codewords y in the dual code defined by G_D, and the addition $x+y$ is performed modulo 2.

From this definition, it looks like there are 16 different quantum states $|\bar{x}\rangle$ that have been defined; however, the sum over codewords generated by G_D means that they are not all distinct. If $x_1 - x_2$ is a valid codeword generated by G_D, then $|\bar{x}_1\rangle = |\bar{x}_2\rangle$. There are

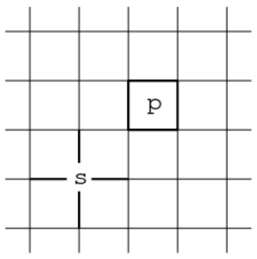

Figure 5: The lattice used for the toric code. Figure taken from Kitaev's original paper.

only two distinct quantum states constructed this way, and they are the two states used in the Steane code above.

Now that we know how the Steane code is constructed, it's possible to see without long and tedious computation that both bit-flip and phase-flip errors are protected against. A bit-flip error in the Z-basis yields

$$\sum_y |x+y+e\rangle = |\overline{x+e}\rangle,$$

which will be a valid state only if e is itself a valid codeword generated by G — that would require at least three bit-flips to have occurred! A phase-flip error is only slightly harder to analyze: the phase-flipped state

$$\sum_y (-1)^{e\cdot(x+y)}|x+y\rangle$$

will be a valid codeword only if e is orthogonal to $x+y$ for all y generated by G_D. This requires that e have at least three bits on.

The Steane code is a specific example of a Calderbank-Shor-Steane code, a class of codes derived in this manner from classical linear codes. The CSS codes, in turn, are a subclass of the stabilizer codes discussed in 3.B below.

3.A. Toric Code

For some integer L, picture $2L^2$ qubits layed out in an $L \times L$ grid, such that each qubit is associated to an edge of the grid. The grid is taken to be periodic, so that the top edge is identified with the bottom, and the left edge with the right. (In other words, the grid is placed on a two-dimensional torus — hence the name.) This situation is shown in Figure 5. There are a total of $3L^2$ Pauli operators associated with this lattice, and many more composite operators that can be constructed; for our purposes, there are three families of composite operators of particular interest.

First, we have the "plaquette" operators $P(r)$. These are associated to a small square on the lattice; for convenience, we'll label them by the name of the site (r) in the lower-right corner of that square. The operator P is defined as the product of the four σ_z operators going around that square:

$$P(r) = \sigma_z(r, r+\hat{x})\sigma_z(r, r+\hat{y})\sigma_z(r+\hat{x}, r+\hat{x}+\hat{y})\sigma_z(r+\hat{y}, r+\hat{x}+\hat{y}).$$

Note that flipping a qubit along a link — that is, the action of σ_x — flips two neighboring plaquettes, while leaving all other unaffected. It turns out that we can flip four qubits in such a way as to leave all plaquettes unaffected. This is the action of the "star" family of operators, $S(r)$. The operator $S(r)$ is associated to a site r, and has the effect of flipping all edges that touch that site:

$$S(r) = \sigma_x(r, r+\hat{x})\sigma_x(r, r+\hat{y})\sigma_x(r, r-\hat{x})\sigma_x(r, r-\hat{y}).$$

Because every plaquette that borders the site r gets two edges flipped, there is no overall effect. The action of the star operator does not affect any plaquettes:

$$S(r)P(r')S(r) = P(r').$$

One last family of operators is of great interest. First, note that from the small loops of the plaquette operator, we can construct operators corresponding to almost all larger loops. For instance, the product of two adjacent plaquette operators $P(r)P(r+\hat{x})$ yields an operator corresponding to the product of the 6 σ_zs going around a 2×1 loop. However, there is one sort of loop that cannot be obtained this way: a loop that winds itself all the way around the lattice, coming back around to the original position. Two such loops are shown in the figure. The associated operators, Z_v and Z_h, are products of σ_z operators along the loops; one can also define $X_\bullet$, for instance, as the product of σ_x operators around a loop. Note that these operators, too, are unaffected by the action of $S(\cdot)$, as each star operator must flip two links in the loop.

We have selected only one vertical loop and one horizontal loop, out of many (far more than $L^2!$) possibilities. The choice of specific loops is arbitrary, but the decision to pick only one in each direction is not. Any other horizontal loop, for instance, can be obtained as a product of our chosen Z_h and some plaquette operators.

The set of operators we've chosen is not "complete" — many operators, like $\sigma_z(\ell)$ for any edge ℓ, cannot be constructed from products of operators chosen from these families. However, we are particularly interested in operators that commute with the $S(\cdot)$ family, and in fact the plaquettes and loop operators (once the σ_x and σ_y versions are included) exhaust that list of operators. Every operator that commutes with $S(\cdot)$ can be obtained as some product of $P(\cdot)$ and $Z_\bullet$, and their X- and Y-basis equivalents.

To summarize, we now have three families of operators ($S(r)$, $P(r)$, and $Z_\bullet$) acting on a Hilbert space of $2L^2$ qubits. Both the $P(\cdot)$ and the $Z_\bullet$ family commute with the star operators, and in fact these are the only operators that do so. From this large set of physical qubits, we will encode only two logical qubits. They will be encoded via the operators Z_v and Z_h.

At the moment, this does not seem like a very useful idea. A single bit-flip error on any link lying along Z_v can change the value of Z_v, so there is no protection from quantum noise. To gain meaningful protection, let's define a restricted subspace $\mathscr{H}_0 \subset \mathscr{H}$ of the original Hilbert space. This subspace will be the set of states $|\Psi\rangle$ obeying two laws:

$$S(r)|\Psi\rangle = |\Psi\rangle \text{ and } P(r)|\Psi\rangle = |\Psi\rangle.$$

Thinking in Z-basis, the first law can be interpreted as limiting us to the symmetric subspace of $\mathscr{H}$, of states unaffected by $S(r)$. The second law dictates that every plaquette have an even number of flipped bits.

Remember that every operator that commutes with $S(\cdot)$ is obtainable as some combination of $P(\cdot)$ and $Z_\bullet$. That means that every operator that does not remove us from the protected subspace $\mathscr{H}_0$ must be constructed as some combination of these operators. However, on this protected subspace, we have also required that $P(\cdot)$ leave all states unchanged. As a result, the only non-trivial operators on this subspace are the loop operators $X_\bullet$, $Y_\bullet$, and $Z_\bullet$. This is exactly enough to encode two qubits, and any other operation removes us from the protected subspace and indicates an error.

An equivalent way to phrase this: within the protected subspace $\mathscr{H}_0$, there are exactly four orthogonal states. One of these states corresponds to $Z_v = Z_h = 0$, one corresponds to $Z_v = Z_h = 1$, and the other two correspond to one flipped qubit each.

How many bit-flips does it take to change the value of V, without removing us from the protected subspace $\mathscr{H}_0$? The best we can do is to draw a horizontal line through the lattice, and apply σ_x at every vertical edge that crosses this line. Thus, in the toric code, L bit-flip errors are required to cause a logical flip.

The toric code's performance is nothing remarkable, but its simple structure and physical interpretation make it a popular target for physical implementation.

The toric code is an example of a physical model called a *gauge theory*. Consider the Hamiltonian

$$H = \sum_r P(r) + \sum_\ell \sigma_x(\ell),$$

where the first sum ranges over all squares (taking the product of σ_z around the square) and the second ranges over all edges. This Hamiltonian is unchanged under the action of the star operator S defined above. Ordinarily, symmetries of physical systems are global: we rotate the entire system by some amount, or flip all spins, for example. The symmetry under S is a local symmetry, as there is one for each lattice site, and the operator does not affect sites that are far away. This is termed a *gauge symmetry*, and the theory posessing it a *gauge theory*. The local symmetry group here is $\mathbb{Z}_2$, other gauge theories with different groups appear in the standard model[7].

The $\mathbb{Z}_2$ gauge theory, in addition to being used in the toric code, has a very interesting property. All lattice models have a notion of a *correlation length*: a distance scale at which "typical" fluctuations occur. If you view the lattice at scales much larger than this length, you will perceive only uncorrelated noise — each region is doing something random, with no relation to each other. At scales much shorter than the correlation length, all spins are aligned and nothing interesting is happening.

In the $\mathbb{Z}_2$ gauge theory, if the coupling constant g is tuned to just the right value, the correlation length ζ blows up, becoming arbitrarily large. This means that if you try to look at the lattice at the scale of a typical fluctuation, you have to "zoom out" very far. This is tremendously valuable, because it suggests that when g is tuned in this way, the geometry of the lattice is no longer relevant to the behavior of these fluctuations. In this way, a discrete lattice can serve as a good approximation to a continuous space.

3.B. Stabilizer Codes

A key aspect of the discussion of the toric code above was the idea that we could specify certain operators under which a state was unchanged (like $S(\cdot)|\Psi\rangle = |\Psi\rangle$), and thereby dramatically restrict or even uniquely identify the state. A couple additional examples are in order. If we have one qubit, and the state in question obeys $X\langle\Psi| = \langle\Psi|$, then the state must be $|\Psi\rangle = |0\rangle + |1\rangle$. We say that the operator X *stabilizes* the state $|\Psi\rangle$. Similarly,

[7] The gauge theories that appear in the standard model have continuous gauge groups: electromagnetism is described by a $U(1)$ gauge field, the weak force by $SU(2)$, and the strong nuclear force by $SU(3)$.

with two qubits, if a state is stabilized by X_1X_2 and Z_1Z_2, then the state must be the EPR (Einstein-Podolsky-Rosen) state $|00\rangle + |11\rangle$.

In general, given a set S of operators, the set of states V_S stabilized by S must form a linear subspace of the original vector space V. Note also that if $S_1, S_2 \in S$ both stabilize V_S, it follows that S_1S_2 and S_2S_1 stabilize V_S. The set of stabilizers, in other words, is closed under multiplication (and therefore forms a group).

Because we're working with qubits, it is sensible to require the elements of S to be selected from the n-qubit *Pauli group*. For one qubit, the Pauli group has 16 operators

$$G = \{\pm 1, \pm i, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}$$

For n qubits, the Pauli group has 2^{4n} operators.

Every pair of operators $g, h \in G$ either commute, obeying $gh = hg$, or anticommute, obeying $gh = -hg$. If we allow two anticommuting operators g, h into the stabilizer S, then V_S will be trivial, since the only state obeying $v = -v$ is $v = 0$. Therefore, the in order for V_S to be interesting, S must consist only of some commuting subset of G. (In other words, S is an abelian subgroup $S < G$.)

We say that the group S has m generators if m elements $g_1, \ldots, g_m \in S$ can be found such that any $h \in S$ can be expressed as a product of the gs, and if this cannot be done with any fewer than m elements. The one-qubit Pauli group itself has three generators, which can be chosen to be X, Y, Z.

Every commuting generator that we add to S cuts the dimension of the stabilized subspace by a factor of two. Starting from n physical qubits, we can obtain a 2^k-dimensional V_S by selecting $n-k$ commuting generators from the Pauli group. This defines a code $C(S)$: states in $V(S)$ will be the valid codewords, and any state outside that subspace will indicate an error. Both the toric code and the nine-qubit code above are special cases of such stabilizer codes.

3.C. Gottesman-Knill Theorem

The motivation for quantum computers is the promise that they can perform certain computations more efficiently than classical computers. In fact, for certain problems (notably factoring products of large primes), quantum computers are believed to provide an exponential speedup over classical computers, meaning that where a classical computer might take time $O(e^n)$ where n is a measure of the size of the problem, a quantum computer requires only $O(n^a)$ steps for some $a > 0$. (Often, but not always, a is a small integer.)

If we believe that quantum computers do have such an exponential speed-up, it follows immediately that simulating a quantum computer on a classical computer must be hard. Indeed, no efficient algorithms for this task are known. However, if the class of permitted quantum circuits is restricted, then some efficient algorithms *are* known. For example, if we allow only controlled-not and Toffoli (controlled-controlled-not) gates, then the quantum circuit is just a classical circuit in disguise, and efficient simulation becomes trivial.

A broader class of efficiently simulable circuits is given by the *Gottesman-Knill theorem*: “Clifford circuits”. A Clifford circuit consists only of the following elements:

- Preparation of states in the Z-basis
- The Hadamard gate H
- The phase gate $S = e^{iZ\frac{\pi}{4}}$
- Controlled-not gates
- Measurement of Pauli operators

Note that the Pauli gates X, Y, Z are implicitly included, as they can be obtained by combinations of H and S.

A naive attempt at simulating a quantum computer with n qubits would proceed by storing a 2^n component complex vector, and updating it for every gate that is applied. Because each of the 2^n complex numbers will need to be updated for every gate that is applied, this requires exponential time in the number of qubits. For general circuits, this is close to optimal[8].

In the case of Clifford circuits, we can do better by keeping track only of the group S that stabilizes the current state. Although this group has exponentially many elements, it only has polynomially many generators. The task of updating these generators can also be done in polynomial time. The Gottesman-Knill theorem presents such an algorithm.

[8]There is a tradeoff between space and time — this algorithm is as fast as possible, but uses much more space than some other algorithms, which use more time.

4. Fault-Tolerance and Thresholds

Be conservative in what you send; be liberal in what you accept.

(Jon Postel)

So far we have only considered the problem of *storage* (or equivalently, transmission) of qubits. Ultimately, the goal is to perform quantum computations in a way that ensures errors are not introduced during the computation. As we will see, this requires actually operating on encoded qubits without decoding them (which would allow errors to leak in while they are unprotected).

As circuits get longer, there are more opportunities for errors to occur. For a fixed error-correction scheme and quality of physical qubits, there is an upper bound on the circuit length, above which errors become impractically frequent. At first glance, this suggests that truly "universal" quantum computers, able to run any circuit (given sufficient time), might be forbidden by the laws of physics. Fortunately, there is an escape hatch. Rather than asking for a single error correction scheme to solve them all, we can have a sequence of error correction schemes, each appropriate for larger families of circuits. Thus, given a circuit, we could pull up an error correction scheme that allows us to run circuits of that size.

It turns out that this is possible. A central point of this section is to convince you of the following theorem: if the probability of errors on physical gates is sufficiently small ($p < p_{\text{th}}$ for some threshold value p_{th} which is independent of the circuit), then for any desired logical circuit and desired error rate ε, there is an fault-tolerant implementation of that logical circuit with logical error rate below ε. Moreover, this can be done with overhead which is only polylogarithmic in the size of the original circuit. Results of this form are called "threshold theorems", and exist for both classical and quantum computation.

Some aspects of this theorem are underspecified. What class of gates? What sorts of errors are considered? In fact a cluster of closely related theorems along this line have been proven. The general idea is fairly robust to details of how the terms are defined, although Section 4.4 below will show how sufficiently large changes to the noise model can break the threshold theorem altogether.

Perhaps the most interesting part of this theorem is that, for any given noise model, the best value of p_{th} is not known. Each proof of a threshold theorem provides some *lower* bound for p_{th}, at least under some specific noise model.

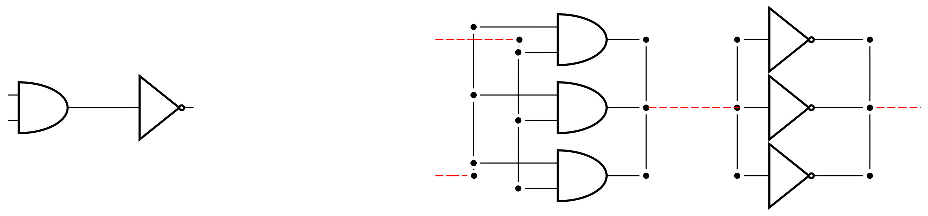

Figure 6: A failed attempt at classical fault-tolerance. On the left is the circuit we seek to make fault-tolerant, consisting of an `AND` gate followed by a `NOT` gate. On the right, the circuit has been transformed using the triple repetition code. Errors can still leak in, with probability proportional to the original error probability, along any wire marked with a dashed red line.

4.1. Classical Fault-Tolerance

The general idea of fault-tolerance is visible already in classical computing[9]. We have some circuit, perhaps the one shown in the left panel of Figure 6, and we would like to reduce the errors in this circuit using an error-correcting code.

For concreteness, we'll use the triple repetition code in this section. The most obvious guess for an error-corrected circuit is shown on the right side of Figure 6. Each gate is sandwiched by a decoding/encoding pair, so that gates act on logical bits, but bits are protected by error-correction when not being used.

For any reasonable noise model, this method is not effective. The gates being used at the core of the original circuit are certain to introduce errors themselves, and nothing has been done to reduce those errors. In fact, if we neglect errors that occur when bits are being 'stored', then this error correction scheme has made no improvement. On the contrary, errors are much worse now, because we've added many more gates in order to do the encoding/decoding!

From this example, it is apparent that new gates must be constructed which act directly on encoded bits; otherwise, the processes of encoding and recoding introduces more errors than it removes. Many classical gates are used in practice; however, we will demonstrate

[9]In general, this method is not actually used in classical computing. It's possible to get errors in classical circuits down to very low levels just by careful hardware design, so fault-tolerant circuits of the sort we discuss here don't appear.

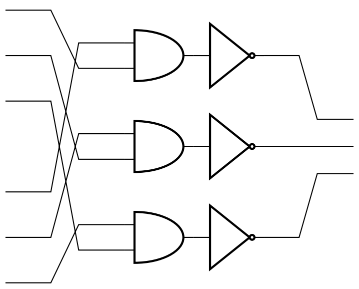

Figure 7: A fault-tolerant implementation of the `NAND` gate. Critically, no decoding is ever performed, so there are no vulnerable wires.

the idea of a fault-tolerant gate implementation just on the `NAND` gate:

a	b	`NAND`(a,b)
0	0	1
0	1	1
1	0	1
1	1	0

It turns out the `NAND` gate is universal, meaning that every other circuit can be built just from `NAND` gates. As a result, this demonstration is sufficient to show that arbitrary classical circuits can be made fault-tolerant.

A fault-tolerant implementation of the `NAND` gate is shown in Figure 7. The construction is as simple as possible: the bits of the two inputs are paired off, and the `NAND` gate is applied to each pair independently. A bit flip, anywhere in the circuit, can only affect one bit of the output. Thus, if the original inputs were each either `000` or `111`, then after only a single bitflip, the output will decode to the correct answer (although it might be any bitstring).

4.2. Classical Thresholds

The construction of the previous subsection reduces the probability of an uncorrected error from $\sim p$ to $\sim p^2$. For very large circuits, this may still be unacceptably large. In fact, if we're limited to $\sim p^2$, this places an upper-bound on the size of a circuit that we can ever run. Aside from being a practical pain point, this is of great theoretical concern, as it means that we haven't constructed a true "universal Turing machine" — there are some programs that we can never run, no matter how long we're willing to take.

Fortunately, it is possible to get the probability of an uncorrected error to be arbitrarily small, provided that the physical error probability p is not too large. The core trick is *concatenation*, which we saw in the previous section. One iteration of error correction transforms a circuit with error probability $\sim p$ to one with error probability $\sim p^2$. If we then repeat the error-correction transformation on this (already once-error-corrected) circuit, the result has a failure probability $\sim p^4$. We can continue the concatenation to get this error arbitrarily small. The main cost is that we require ever-more physical qubits, which may be expensive to construct.

That quick analysis is a bit heuristic, and we never quantified the condition that "p is not too large". For concreteness and simplicity, assume that errors are only ever introduced when gates act on bits: there is no corruption of bits that are just sitting in storage. Furthermore, our fault-tolerant circuit transformation takes a gate with error probability p and transforms it to a small circuit with error probability ap^2. Concatenating the fault-tolerant construction n times, then, yields a circuit with error probability $\frac{L}{a}(ap)^{2^n}$, where L is the number of gates in the original circuit.

Now we must distinguish two cases. If $ap < 1$ (by any amount[10]), then making n larger lowers the error probability. If, however, $ap > 1$, making n larger actually makes things work. We call $p_{\text{th}} \equiv a$ the *threshold*.

Another way to phrase this result is as follows. Suppose we have a circuit of L gates, and we want to obtain a fault-tolerant version with an error probability below ε. In order to do this, we must concatenate n times, with

$$n \geq \log_2 \left(\frac{\log \frac{\varepsilon a}{L}}{\log ap} \right).$$

Of particular interest, if we're interested in ensuring that we can implement arbitrarily large circuits with a minimum of overhead, is how slowly n grows as a function of L, for fixed p, ε, and a. The above equation indicates that $n \sim \log_2 \log L$.

How large will this make our error corrected circuits? The size of a fault-tolerant circuit grows exponentially — but *singly* exponentially — with n. As a result we find that the size of the fault-tolerant circuit, with n concatenations (enough to get the error probability below ε), grows as $L\,\mathrm{poly}(\log L)$. Thus the overhead from this sort of error correction is only polylogarithmic in the size of the circuit. Asymptotically, at least, this scheme is everything we could ever hope for.

[10]Remember that our model of noise was incredibly simplistic, so this sort of crude estimate of the threshold is likely to be optimistic.

In practice, a lot of hard work is hiding in the word "asymptotically". Present-day quantum processors do not obtain $p < p_{th}$. Until the threshold is achieved, this construct is of no practical use.

4.3. Quantum Fault-Tolerance

The central principle of quantum fault-tolerance is the same as that of classical fault-tolerance: we must apply gates to encoded qubits without ever performing a decoding. However, in the quantum case, several complications arise. Most obviously, interference can occur between different qubits in a code. Consider, for instance, the $\frac{\pi}{8}$ gate T:

$$T = \begin{pmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{pmatrix}.$$

We will attempt to implement this in a fault-tolerant way on the three-qubit bit-flip code[11]. An obvious first guess is to apply T to each qubit separately. However, assuming no errors occur, this results in a total phase of $e^{i\frac{3\pi}{8}}$, thus implementing the wrong gate. A similar issue occurs if we attempt to implement the Hadamard gate with the same strategy:

$$\begin{aligned} |000\rangle \xrightarrow{H^{\otimes 3}} &(|0\rangle + |1\rangle)(|0\rangle + |1\rangle)(|0\rangle + |1\rangle) \\ =&|000\rangle + |001\rangle + |010\rangle + |011\rangle + \cdots \\ \neq&|000\rangle + |111\rangle. \end{aligned}$$

Of course, this issue does not occur with all one-qubit gates. In this case, a fault-tolerant X gate really can be constructed as $X^{\otimes 3}$.

A less obvious, but more severe, issue is that a poorly designed fault-tolerant gate may make pre-existing errors worse. If a one-qubit error occurs early in a circuit, then some later gate may entangle this qubit with another, resulting in a two-qubit gate. Importantly, this does not require the later gate to have any errors at all. The one-qubit error is expanded into a two-qubit error despite the fact that only one (one-qubit) error ever occured.

This second issue can be mitigated, in some cases, by *transversal gates*. A transversal gate is a fault-tolerant gate implemented by performing an operation on each qubit independently. For instance, the obvious fault-tolerant implementation of X, in which X is

[11] This is only for demonstration purposes. Because this code does not protect us from phase errors, we should not expect a phase gate to be implementable in a fault-tolerant way.

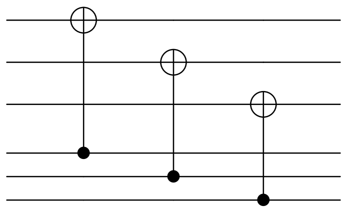

Figure 8: The controlled-NOT gate, implemented on the three-qubit bit-flip code. Because the circuit is transversal, a single error going in results in only a single error going out (assuming no additional errors occur in the circuit itself).

applied independently to each qubit in the three-qubit bit-flip code, is transversal. Because this gate never operates on two qubits at once, errors have no opportunity to expand (just as in the classical case).

Another case of a transversal gate is a fault-tolerant CNOT gate, again implemented on the three-qubit bit-flip code. This is shown in Figure 8, and follows exactly the same scheme one would use in the classical case. This idea generalizes to the Toffoli gate, and any gate which is a permutation matrix in the Z-basis. This is true not only for the three-qubit bit-flip code, but for a large class of QECs called Calderbank-Shor-Steane codes, which are derived from the clasical linear codes.

Once a method for constructing fault-tolerant circuits is obtained, it can be concatenated with itself to yield further improvements. The logic of Section 4.2 needs no modification for the quantum case.

4.4. Model-Dependence

In the analysis of Section 4.2, we assumed that errors were only introduced at gates, and that there was only one type of gate. This serves to illustrate the idea behind the threshold theorems, but is not a very realistic error model. Performing the same task with realistic models is tedious and yields the same qualitative result (polylogarithmic overhead), so we won't perform such an analysis here. However, it is possible to construct a (non-realistic) noise model under which this sort of threshold theorem does *not* hold.

First, instead of having errors occur only when gates are applied, let's have errors occur also when (qu)bits are sitting in storage. This is in fact the case[12]! In addition, and much

[12]Even classical bits are likely to encounter errors this way. Although it sounds like a joke, the phenomenon

less realistically, let us forbid ourselves from running gates in parallel. The gates must operate in sequence, and so the amount of time for errors to be introduced is proportional to the number of gates[13].

Remember that with n concatenations, we have a circuit of length b^n for some constant b. If all operations are performed in sequence, this means that the probablity of error in a single physical gate is not p, but $\sim pb^n$. There is now no p which is "sufficiently small": for any p, there's an L such that the required concatenation length n implies that $pb^n > 1$, and error correction is now useless.

Happily, this is an absurd model of noise, since in the real world we allow gates to execute in parallel all the time.

of cosmic rays flipping bits in RAM is considered by some to be a real security concern, exploited in part by "bitsquatting". Other sources of noise may be more realistic, of course.

[13]This may sound reasonable by analogy with the many classical computers which have only a single CPU. You must remember that even a classical computer with a single CPU is running many gates at a time (some in the CPU, some in RAM, some elsewhere). Requiring that the gates execute in sequence is much stronger than requiring that large-scale logical operations, like addition, execute in sequence.

A. Bits and Qubits

A.1. A Bit of Linear Algebra

There is hardly any theory which is more elementary [than linear algebra], in spite of the fact that generations of professors and textbook writers have obscured its simplicity by preposterous calculations with matrices.

(Jean Dieudonné)

A *vector space* is a set (of vectors) equipped with two operations: addition of two vectors, and scalar multiplication by a complex number[14]. Addition is commutative ($u+v=v+u$) and associative ($u+(v+w)=(u+v)+w$), as usual. Scalar multiplication is associative ($a(bu)=(ab)u$) and distributes over both vector addition ($a(u+v)=au+av$) and scalar addition ($(a+b)u=au+bu$). Finally, scalar multiplication by 1 does not change the vector ($1v=v$) and multiplication by 0 yields the unique zero-vector.

Just as the concept of a set is not very useful without the idea of a function, a vector space is fairly pointless without linear operators. A linear operator M on a vector space V is a function that takes a vector $v \in V$ and returns another, Mv. This function must be linear, meaning that $M(au+bv)=aMu+bMv$, for all vectors u,v and scalars a,b.

For the purposes of quantum mechanics, we need to add a little bit of extra structure to our vector space. An *inner product space* is a vector space that comes equipped with an inner product (often introduced as a "dot product"). The inner product between two vectors u,v returns a scalar (a complex number), and is denoted[15] $\langle u,v\rangle$. The inner product is linear in both arguments, is Hermitian ($\langle u|v\rangle = \langle v|u\rangle^*$), and is positive-definite ($\langle v|v\rangle \geq 0$ for any non-zero v).

The existence of an inner product allows us to define a norm — a notion of the "length" of a vector. The norm of a vector v is defined to be $|v|=\sqrt{\langle v,v\rangle}$.

A *Hilbert space* $\mathscr{H}$ is an inner product space with the extra restriction that that the space is *complete*, meaning that if a sequence of vectors $v_1,v_2,\ldots$ get arbitrarily close to each other, then the limit $\lim_{k\to\infty} v_k$ exists. This is automatically true for finite-dimensional inner product spaces. This requirement is not often used in physics; nevertheless, it is conventional to refer to "the Hilbert space" of states even when the completeness axiom is irrelevant.

[14] In fact vector spaces can be defined over any field, as noted in Section 1.A, but the interesting ones in quantum mechanics are all over $\mathbb{C}$.

[15] In the context of quantum mechanics $\langle u|v\rangle$ is more traditional. There, a vector is written $|v\rangle$ or $\langle v|$.

In the case of a finite-dimensional inner product space, the "usual" formulation in terms of a list of numbers can be recovered by selecting a maximal set of orthonormal vectors $e_1, \ldots, e_N$. Any other vector v is uniquely determined by its inner products $v_i = \langle e_i, v\rangle$. Those inner products can be treated as the N components of v.

An important piece of intuition about many vector spaces, including all those used in these notes, is the idea of the dimension of a vector space. A *basis* of a vector space is a list of vectors $e_1, e_2, \ldots$, such that every vector v can be written in a unique way as a linear combination $a_1 e_1 + a_2 e_2 + \cdots = v$. If there are a finite number d of basis vectors, the vector space is said to be d-dimensional. If the vector space is equipped with an inner product, the basis vectors can always be chosen to be orthogonal, so that $\langle e_i, e_j\rangle = 0$ when $i \neq j$.

Two special classes of operators are particularly relevant to physics. First, the *Hermitian* operators are those operators H obeying $\langle u, Hv\rangle = \langle Hu, v\rangle$. This implies, for instance, that $\langle v, Hv\rangle$ is real. Written as a matrix, taking the transpose of H followed by the complex conjugate of all elements yields the same matrix back again: $H^\dagger = H$.

A *unitary* operator U is one whose adjoint is its own inverse, $U^\dagger U = 1$. It follows that U preserves the inner product, obeying $\langle Uv_1, Uv_2\rangle = \langle v_1, v_2\rangle$.

The Hermitian operators and the unitary operators are closely related. Define the matrix exponential of an operator M as

$$\exp M \equiv \sum_k \frac{1}{k!} M^k.$$

For any unitary U, the exponential $\exp iU$ is always Hermitian.

This mathematical correspondence has a simple physical interpretation. Hermitian operators correspond to "observables" — physical quantities that can be measured to describe the state of a system. Because the energy of a system can be measured, the energy of a system must be described by a Hermitian operator, termed the Hamiltonian. Unitary operators, meanwhile, correspond to legal sorts of time-evolution (like the application of a quantum circuit). Time-evolution operators are obtained by exponentiating the Hamiltonian. The requirement that time-evolution be unitary is equivalent to the requirement that the Hamiltonian be Hermitian.

A nice exercise in linear algebra is to prove the *no-cloning theorem*:

No-Cloning Theorem. *Let $\mathscr{H}$ be a Hilbert space of dimension at least 2, and $|0\rangle$ some fiducial state in that space. No unitary operator on $\mathscr{H} \otimes \mathscr{H}$ obeys, for all $|\Psi\rangle$, $U|\Psi\rangle|0\rangle \propto |\Psi\rangle|\Psi\rangle$.*

This theorem states that there's no quantum operation we can perform to take the state of one qubit and duplicate it in another qubit. This is an unfortunate but fundamental property of quantum computation.

Proof. Let $|\Psi\rangle, |\Phi\rangle \in \mathscr{H}$ be two orthogonal states, and suppose we have some unitary U that obeys the cloning property on these two states:

$$U|\Psi\rangle|0\rangle \propto |\Psi\rangle|\Psi\rangle \text{ and } U|\Phi\rangle|0\rangle \propto |\Phi\rangle|\Phi\rangle.$$

Since U is linear, we know how U acts on linear combinations. For example, we have

$$U(|\Psi\rangle + |\Phi\rangle)|0\rangle \in \operatorname{span}\{|\Psi\rangle|\Psi\rangle, |\Phi\rangle|\Phi\rangle\}.$$

But, the desired result $(|\Psi\rangle + |\Phi\rangle)^{\otimes 2}$ does not lie in this two-dimensional space! The difference between the two is $|\Psi\rangle|\Phi\rangle + |\Phi\rangle|\Psi\rangle$, which is in fact orthogonal to that space since Ψ and Φ were chosen to be orthogonal at the beginning. $\square$

A.2. The Density Matrix

If I tell you that a qubit is in the state $|+\rangle$, then there's some uncertainty about what will happen when the qubit is measured in the Z-basis. The result will be either 0 or 1, and with equal probability. Importantly, this uncertainty does not correspond to any uncertainty about what state the qubit is in. The qubit is definitely in the state $|+\rangle = |0\rangle + |1\rangle$. Only the result of the measurement is uncertain. (Furthermore, not all measurements have uncertain results. A measurement in the X-basis is sure to yield 0.)

A different source of uncertainty commonly appears in classical physics. I might have a bit, and tell you that the state is either 0 or 1, but I don't know which. We represent this situation with a probability distribution, $p(b)$, with $p(0) = p(1) = \frac{1}{2}$. This is superficially similar to the quantum uncertainty above, but is really fundamentally different. This uncertainty represents information about the world that we don't have: the bit definitely is either 0 or 1, we just don't know which.

What happens if these two types of uncertainty exist simultaneously? For instance, I could have a qubit that I know is either in the state $|0\rangle$ or the state $|+\rangle$, and with equal probability. Now there is some information about the state that we don't have, but also, even if we did have that information, measurements would still not have deterministic results! In principle, we should represent this situation by a probability distribution on the space of 'pure' (i.e. classically certain) states. It turns out that there's a much simpler way to represent these 'mixed' (classically uncertain) states, termed a *density matrix*.

Classical uncertainty is, in some sense, fundamentally *linear*. The expectation value of any observable $\mathscr{O}$ is given by

$$\langle \mathscr{O} \rangle = \sum_s p(s)\mathscr{O}(s),$$

where the sum is taken over all states s, $p(s)$ is the probability of that state, and $\mathscr{O}(s)$ is the value of that observable in the state s.

This is a nice property that, at first glance, quantum mechanics does not share. An expectation value of $\mathscr{O}$ in the state $|\psi\rangle$ is

$$\langle \psi|\mathscr{O}|\psi\rangle = \sum_{v,w} \psi_w^\dagger \psi_v \mathscr{O}_{wv}.$$

This is quadratic in the wavefunction, not linear. Linearity is never conceptually necessary, but is always convenient. We can improve the situation by thinking, instead of the wavefunction, about the operator $|\psi\rangle\langle\psi|$. Now the expectation value can be written

$$\langle \psi|\mathscr{O}|\psi\rangle = \operatorname{Tr}|\psi\rangle\langle\psi|\mathscr{O}.$$

The magic here is that we're now going to treat the operator $|\psi\rangle\langle\psi|$ (which is itself non-linear in the wavefunction) as the fundamental object, rather that the wavefunction.

Now we're able to easily accomodate classical uncertainty. If the system is in either state $|\psi_1\rangle$ or $|\psi_2\rangle$ with equal probability, then the expectation value of any operator $\mathscr{O}$ is given by

$$\langle \mathscr{O} \rangle = \frac{1}{2}\Big[\operatorname{Tr}|\psi_1\rangle\langle\psi_1|\mathscr{O} + \operatorname{Tr}|\psi_2\rangle\langle\psi_2|\mathscr{O}\Big] = \operatorname{Tr}\underbrace{\left(\frac{|\psi_1\rangle\langle\psi_1| + |\psi_2\rangle\langle\psi_2|}{2}\right)}_{\rho}\mathscr{O}.$$

The object ρ is the density matrix; it is formed simply by taking a linear combination of the two matrices coming from pure states.

The density matrix ρ contains enough information to compute the expectation value of any observable. Usefully, though, it has far less information than a probablity distribution on the space of states would. For example, a system which is in state $|0\rangle$ or $|1\rangle$ with equal probability is described by the density matrix

$$\rho = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}.$$

A short computation will reveal that the same matrix describes a system which is in state $|+\rangle$ or $|-\rangle$ with equal probability. There is no measurable difference between these scenarios!

B. Further Reading

This is a great book. Out of respect, I've never opened it.

(Paulo Bedaque)

B.1. Pedagogical

The canonical text for all things quantum computing is Nielsen and Chuang's textbook, "Quantum Computation and Quantum Information". Many copies are available online, such as this one. Chapter 10 of that book is most relevant to these lectures.

Classical Error Correction

I like this introduction by Matej Boguszak. Those notes are based on Baylis's book "Error Correcting Codes: A Mathematical Introduction".

Less formal is 3Blue1Brown's video.

Most introductions to quantum error correction (like this one!) begin with a brief overview of relevant aspects of classical error correction. Thus, any article on quantum error correction in the section below is also as an introduction to classical error correcting codes.

Quantum Error Correction

Steane's tutorial discusses quantum error correction but not fault-tolerance. This article by Devitt et al., and Gottesman's introduction, both discuss fault-tolerance as well. Finally, Preskill has an article focusing on fault-tolerance.

B.2. Classic Papers

Classical Error Correction

Classical error correcting codes existed before 1950, but Hamming's paper in that year layed more careful foundations for the field.

A classical threshold theorem (probably the first of its type) was proven around 1956 by von Neumann. I cannot find a canonical reference, but these notes were made from a series of lectures he gave.

Quantum Error Correction

The toric code was invented by Kitaev in 1997.

The threshold theorem began in a paper of Peter Shor, and was independently developed by three groups: Knill et al, Aharanov and Ben-Or, and Kitaev.