

Topological Codes and Computation

A lecture course given at the University of Innsbruck
May - June 2014

Dan Browne

June 6, 2014

Contents

Contents	1
1 Quantum Error Correction and The Stabilizer Formalism (Revision)	7
1.1 Classical three-bit repetition code	7
1.2 Quantum three-qubit repetition code	9
1.3 Stabilizer formalism	14
2 The Toric Code	23
2.1 Defining the code	23
2.2 Errors on the toric code	33
3 Elements of Topology and Homology	47
3.1 Topological equivalence and topological invariants	48
3.2 \mathbb{Z}_2 Homology	58
3.3 Cohomology	72

4	Topological codes from homology	83
4.1	Topological codes on surfaces with no boundary - a homological definition	83
4.2	Surfaces with boundary – Planar codes	91
5	Elements of fault-tolerant quantum computation with planar surface codes	111
5.1	Overview	111
5.2	Achieving universal quantum computation	112
5.3	Fault tolerance	114
5.4	Code deformation	117
5.5	Completing the set of gates	127
5.6	Summary and outlook	128

Recommended Further Reading

This is not an exhaustive list of references, rather a starting point for further reading.

Revision of Quantum Computation and Quantum Error Correction

- Michael A. Nielsen and Isaac Chuang - *Quantum computation and quantum information* (Cambridge University Press: 2000)

The classic (and still the best) introductory textbook on quantum computation theory. The chapter on error correction (which covers the stabilizer formalism) is chapter 10.

- Daniel Gottesman, *An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation*, <http://arxiv.org/abs/0904.2557>

A detailed introduction to the stabilizer formalism, by the person who invented it, which covers similar ground, but more deeply and more rigourously, than the notes for part 1 of this course.

Toric Code

The following papers are the research papers in which many of the key results of topological codes were established. They are densely written research papers and are not an easy read, but are sparkling with ideas and well worth working through.

- Alexei Y. Kitaev, *Fault-tolerant quantum computation by anyons*, <http://arXiv.org/abs/quant-ph/9707021>

The paper which introduced both the toric code (and surface codes in general) and topological quantum computation with anyons.

- Eric Dennis, Alexei Kitaev, Andrew Landahl, John Preskill *Topological quantum memory*, <http://arXiv.org/abs/quant-ph/0110143>

This paper established many of the key results in the study of toric codes.

Planar codes

- Sergei B. Bravyi and Alexei Y. Kitaev, *Quantum codes on a lattice with boundary*, <http://arxiv.org/abs/quant-ph/9811052>

One of the two papers (see also Meyer and Freedman, <http://arxiv.org/abs/quant-ph/9810055>) which generalised the toric code construction to surface codes with boundary, including planar codes.

Fault-tolerant quantum computation with planar codes

- Robert Raussendorf, Jim Harrington, Kovid Goyal, *A fault-tolerant one-way quantum computer*, <http://arxiv.org/abs/quant-ph/0510135>

The first of a series of papers by Robert Raussendorf and co-authors which introduce many of the key ideas of fault-tolerant surface code computation. Raussendorf and co-authors develop these ideas in the measurement-based quantum computation framework.

- Austin G. Fowler, Matteo Mariantoni, John M. Martinis, Andrew N. Cleland *Surface codes: Towards practical large-scale quantum computation* <http://arxiv.org/abs/1208.0928>

A clearly written and self-contained introduction to fault tolerant quantum computation via code-deformation in the surface code.

- Barbara Terhal, *Surface Code Architecture* http://profs.if.uff.br/paraty07/paraty11/en/notes_2011/lecture1_surface_code.pdf

Slides from a lecture course on fault-tolerant computation with surface codes by Barbara Terhal. Covers many of the same topics as this course, and provides an excellent overview of the subject.

Homology

- P. Giblin, *Graphs, Surfaces and Homology*, Cambridge University Press (2010).

An intuitive introduction to topology and homology with many explicit examples.

- M. Henle, *A Combinatorial Introduction to Topology*, Dover Notes (1994).

Another clear introduction to topology, clearly illustrated with many examples.

1 Quantum Error Correction and The Stabilizer Formalism (Revision)

We begin with a brief revision of quantum error correcting codes, focussing on the stabilizer formalism, looking at how codes are defined, and how errors, their detection and correction are described. To refresh many of the key concepts of error correction, we start with the simplest classical error correcting code – the repetition code.

1.1 Classical three-bit repetition code

The classical repetition code encodes bits by repeating them. The smallest repetition code which can correct errors is the three-bit code:

1.1.1 Codewords

$$0_L = 000 \quad 1_L = 111 \quad (1.1)$$

1.1.2 Bit-flip Error

A *bitflip error* flips bit-value 0 to 1 and 1 to 0.

1.1.3 Codewords after a single bit-flip error

Consider a bit-flip on the second bit. This transforms the logical codewords as follows

$$0_L \rightarrow 010 \quad 1_L \rightarrow 101 \quad (1.2)$$

1.1.4 Error detection

The signature that an error has occurred is that the bits in the string are not identical.

1.1.5 Error correction

To correct errors, we can compute the majority-function of the bitstring. This returns the bitvalue that occurs the most often, we then reset all bits to that majority value.

$$010 \rightarrow 000 \quad 101 \rightarrow 111 \quad (1.3)$$

Provided fewer than half of the bits were flipped, this error correction process will return the bitstring to the correct codeword.

1.1.6 Code Distance

The *distance* of a classical code is the fewest number of bit flips needed to transform any two code-words into one another. The distance of the 3-bit repetition code is 3, and more generally, the n -qubit repetition code has distance n .

The minimum number of bit flips to transform bitstring a to bitstring b is called the Hamming distance between a and b .

The code-distance determines the maximum number of errors which can be detected, $(d-1)$, and the maximum number that can be corrected, $\lfloor (d-1)/2 \rfloor$.

1.1.7 $[n,k,d]$ Notation

Three important parameters describing a code are n , the number of bits in the codewords, k , the number of encoded bits, and d the code distance, are often written as a triple $[n,k,d]$. The n -bit repetition code is an $[n,1,n]$ code.

1.1.8 Logical operations on the code

We may wish to perform computations on encoded information. The simplest logic gate is the NOT gate, which flips the logical bit-value. A logical NOT gate on a repetition code qubit is to flip all the qubits:

$$000 \leftrightarrow 111 \quad (1.4)$$

1.2 Quantum three-qubit repetition code

We shall now consider the quantum code derived from this classical error correcting code.

1.2.1 Codewords

We encode a quantum bit using the same three-bit repetition encoding:

$$|0\rangle_L = |0\rangle \otimes |0\rangle \otimes |0\rangle \equiv |000\rangle \quad |1\rangle_L = |1\rangle \otimes |1\rangle \otimes |1\rangle \equiv |111\rangle \quad (1.5)$$

A general pure qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is thus encoded:

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle \quad (1.6)$$

We call states $|0\rangle_L$ and $|1\rangle_L$ *codeword basis states*, and we call the state space spanned by these states, the *codespace*. Any state representing an error-free encoding of a quantum state lies in the codespace.

1.2.2 Pauli operators

Pauli operators will play a central role in this course. In the computational basis, they are defined via the following matrices. It is convenient to include the identity operator as a Pauli operator. (Following standard practise in quantum information, we write X, Y, Z, I instead of $\sigma_x, \sigma_y, \sigma_z, \mathbb{1}$).

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (1.7)$$

Pauli operators are Hermitian (so define measurements) and also unitary (so define quantum logic gates). We shall use them here in both of these (very different!) roles.

1.2.3 Properties of Pauli operators

- They are unitary *and* Hermitian.
- Any two Pauli operators either *commute* or *anticommute*.
- They have eigenvalues $+1, -1$.
- They are self-inverse $X^2 = Y^2 = Z^2 = I$.
- They form a basis for 2×2 matrices, so any such matrix can be written $M = \alpha X + \beta Y + \gamma Z + \delta I$.
- $Y = iXZ$, so, if we allow the Pauli operators to have prefactors $\pm 1, \pm i$, the operators form a group. This group can be generated by iI, X and Z . NB This full group contains operators such as iY and iI which are not Hermitian. In this course we reserve the term “Pauli operator” for Hermitian members of this group (and its n -bit generalisation).

1.2.4 Bit-flip Error

A classical *bitflip error* flips bit-value 0 to 1 and 1 to 0. A quantum bit-flip error does the same. It is represented by the Pauli operator X

X acts as a bit-flip on qubit computational basis states:

$$X|0\rangle = |1\rangle \quad X|1\rangle = |0\rangle \quad (1.8)$$

1.2.5 Phase-flip Error

In addition to bit-flip errors, quantum bits can undergo phase errors. A phase-flip is represented by the Pauli operator Z .

$$Z|0\rangle = |0\rangle \quad Z|1\rangle = -|1\rangle \quad (1.9)$$

Although there can be much wider variety of quantum errors than simply a X and Z operator, due to the way quantum error correcting codes work, if both X and Z errors can be corrected, more general errors can be corrected as well.

1.2.6 Encoded state after a single bit-flip error

After a bit-flip error on the second qubit, our encoded qubit is transformed to:

$$|\psi\rangle_L \rightarrow (I \otimes X \otimes I)|\psi\rangle_L = \alpha|010\rangle + \beta|101\rangle \quad (1.10)$$

where $I = \mathbb{1}$ is the identity operator.

1.2.7 Error Detection

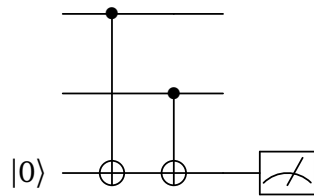
Unlike the classical code, we do not want to measure individual qubits in computational basis. This would detect the error but measure (collapse) the encoded state, destroying the encoded qubit. Instead, we can detect errors by measuring *symmetries* of the state rather than individual bits.

The symmetry of the repetition code we exploit is that the *parity* of any pair of bits is *even*.

- Even parity: 00, 11
- Odd parity: 01, 10

1.2.8 Quantum circuit to measure parity

We can measure the parity of two qubits (whether state is in subspace spanned by $|00\rangle, |11\rangle$ or the subspace spanned by $|01\rangle, |10\rangle$) by adding an ancillary qubit prepared in $|0\rangle$ and using CNOT gates as follows:



Here the meter indicates a measurement of the ancilla qubit in the computational basis (i.e. a measurement of Z).

Measuring the parity of two-qubits with respect to the computational basis is equivalent to measuring the observable $Z \otimes Z$ on the first two qubits, since

$$\begin{aligned} (Z \otimes Z)|00\rangle &= |00\rangle & (Z \otimes Z)|01\rangle &= -|01\rangle \\ (Z \otimes Z)|10\rangle &= -|10\rangle & (Z \otimes Z)|11\rangle &= |11\rangle \end{aligned} \quad (1.11)$$

Note that even parity is associated with the +1 eigenvalue, and odd parity with the -1 eigenvalue.

1.2.9 Error Detection with parity measurements

If we make a parity measurement between *any* pair of qubits on the encoded state

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle \quad (1.12)$$

the measurement will detect even parity. In other words, the measurements $Z \otimes Z \otimes I$, $Z \otimes I \otimes Z$ and $I \otimes Z \otimes Z$, will all return +1 eigenvalues.

If there has been a bit-flip error, e.g.

$$|\psi\rangle_L \rightarrow \alpha|010\rangle + \beta|101\rangle \quad (1.13)$$

two of the measurements (here $Z \otimes Z \otimes I$ and $I \otimes Z \otimes Z$) will return -1 outcomes.

Unlike individual qubit measurements, these parity measurements do not reveal any information about the logical encoded qubit, and making these measurements does not collapse or otherwise disrupt the encoded qubit – encoded states are left invariant.

Different errors on the qubits correspond to different combinations of outcomes for pairwise parity measurements:

Error	$Z \otimes Z \otimes I$	$Z \otimes I \otimes Z$	$I \otimes Z \otimes Z$
$X \otimes I \otimes I$	-1	-1	+1
$I \otimes X \otimes I$	-1	+1	-1
$I \otimes I \otimes X$	+1	-1	-1

Note that the outcomes in the table coincide with whether the error commutes (+) or anticommutes (−) with the measured operator.

1.2.10 Error Correction

Once the error is identified, we may apply the inverse of the error operator to correct it. E.g. if $X \otimes I \otimes I$ is the error, the operator $X \otimes I \otimes I$ (since Pauli operators are self-inverse) will correct it.

1.2.11 Logical operations on encoded states

We wish to perform computation on our encoded qubit. Identifying encoded logical X and Z operators allows us to derive any single qubit unitary on the encoded qubit (since $Y = iXZ$, and is thus specified once X and Z are defined, and since X , Y , Z and I form a basis to express any single qubit operator).

The X operator flips the bit value: $X|0\rangle = |1\rangle$, $X|1\rangle = |0\rangle$. By inspection we see that the operator $X \otimes X \otimes X$ transforms the logical states as a logical bit flip.

$$(X \otimes X \otimes X)|000\rangle = |111\rangle \quad (X \otimes X \otimes X)|111\rangle = |000\rangle \quad (1.14)$$

Hence, this operator is an encoded X . We write encoded logical operators \bar{X} , hence for this code,

$$\bar{X} = X \otimes X \otimes X. \quad (1.15)$$

The encoded logical Z must map $|0\rangle_L$ to $|0\rangle_L$ and $|1\rangle_L$ to $-|1\rangle_L$. Again, by inspection, we see that the operator $Z \otimes I \otimes I$ will achieve this, hence

$$\bar{Z} = Z \otimes I \otimes I. \quad (1.16)$$

It is clear from the symmetry of the code that we could have chosen *other* operators with the same effect – $I \otimes Z \otimes I$ and $I \otimes I \otimes Z$, for example. There are *multiple* equivalent operators which realise the *same* encoded unitary. This is best understood in the stabilizer formalism (see below).

1.2.12 Distance of a quantum code

The *distance* of a quantum code is the minimal weight of any (non-identity) encoded logical operator on the code. This coincides with the minimum weight of logical \bar{X} and \bar{Z} operators for the code. The *weight* of an operator is the number of qubits it acts non-trivially on. E.g. $Z \otimes Z \otimes I$ has weight 2, $X \otimes X \otimes X$ has weight 3, and $I \otimes I \otimes I$ has weight 0.

We can consider the minimum weight of X and Z operators separately. For the 3-qubit repetition code, the minimum weight of encoded X is 3 ($X \otimes X \otimes X$). Hence the code can detect up to 2 X errors. The minimum weight of encoded Z , on the other hand is 1. This means that the repetition code cannot detect Z errors at all. This is to be expected, since the repetition

code is a classical code, and not designed to correct quantum errors, like the phase-flip. Fortunately, quantum codes exist which can detect and correct both types of error.

1.3 Stabilizer formalism

The stabilizer formalism is a powerful set of techniques to define and study quantum error correcting codes in terms of Pauli operators, rather than directly with the state vector kets of the code-words. It has many advantages, and is the most widely used formalism used to describe topological codes.

Its advantages include:

- An efficient description of many-qubit states (who could have exponentially many terms in their state vector).
- A straightforward analysis of symmetries of codes, error detection measurements and corrections.
- Clear rules to derive encoded logical operators.

1.3.1 Multi-qubit Pauli operators and the Pauli group

An n -qubit Pauli operator is a tensor product of n Pauli operators (X, Y, Z, I) with pre-factor $+1$ or -1 , e.g. containing $(X \otimes X)$. It shares many of the properties of a single qubit Pauli operators.

As a notational shorthand, when we write down n -qubit Pauli operators, we will often suppress the tensor product symbol, so instead of $X \otimes X \otimes Y \otimes I$ we'll write $XXYI$. When we multiply such operators together we'll enclose them in brackets, e.g. for $(X \otimes X \otimes Y \otimes I) \cdot (Y \otimes Z \otimes I \otimes Y)$ we'll write $(XXYI)(YZIY)$.

1.3.2 Properties of n -qubit Pauli operators

- They are both unitary and Hermitian.
- Any two operators either *commute* or *anticommute*.
- They have eigenvalues $+1, -1$.
- They are self-inverse.

- They form a basis for $2^n \times 2^n$ matrices, or equivalently, for the space of n -qubit operators.
- In this course, the term *Pauli operator* will be used to mean these general n -qubit Pauli operators.

1.3.3 n -qubit Pauli-group

Again, because $Y = iXZ$, if we allow our Pauli operators to take prefactors $\pm i$ as well as ± 1 , then the operators form a group. This group can be generated by iI , X_j and Z_j for each qubit j , where notation X_j refers to the operator which acts as X on the j th qubit, and as the identity elsewhere, e.g. $Z_2 = IZI$. This full group contains operators such as $iYZI$ which are not Hermitian. We will reserve the term *Pauli operators* for the Hermitian elements of this group.

1.3.4 Stabilizer formalism for Quantum Error Correction

A quantum error correcting code that can be defined in the stabilizer formalism is called a *stabilizer code*. Stabilizer codes are defined by specifying two sets of operators, a set of *stabilizer generators*, and a set of *encoded logical operators*.

1.3.5 The stabilizer (group)

The stabilizer group for a quantum code (often simply called the *stabilizer* of the code) is the set of n -qubit Pauli operators which leaves the code invariant.

Consider a quantum error correcting code with code-word basis states $|\psi_j\rangle$. The stabilizer group is the set of Pauli operators P (stabilizer operators) which leave all codeword basis states $|\psi_j\rangle$ invariant. I.e. for all $P_k \in P$ and all j

$$P_k |\psi_j\rangle = |\psi_j\rangle. \quad (1.17)$$

By the linearity of quantum mechanics, these operators must leave any linear superposition of codeword states, and the entire codespace, invariant.

We shall call a member of the stabilizer a stabilizer operator (or stabilizer element). The set of stabilizer operators must commute (the proof is left as an exercise) and form a group, since if P_1 and P_2 satisfy equation (1.17),

$$P_1 P_2 |\psi_j\rangle = P_1 |\psi_j\rangle = |\psi_j\rangle. \quad (1.18)$$

The set of stabilizer operators therefore forms an Abelian group, and is thus an Abelian sub-group of the n -qubit Pauli group.

1.3.6 Stabilizer generators

Any group G can be specified by a set of generators g_j , where j ranges from 1 to m . For an Abelian group of self-inverse operators, any element $g \in G$ can be written:

$$g = \prod_j g_j^{a_j} \quad (1.19)$$

where $a_j \in \{0, 1\}$. Each element can therefore be expressed in terms of the bitstring $a_1 a_2 \dots a_m$. We say a set of generators is *independent* if the only solution of

$$\prod_j g_j^{a_j} = I \quad (1.20)$$

is $a_j = 0$ for all j , where I is the identity operator. This independence condition (strongly analogous to linear independence) implies that no generator can be written as a product of other generators, and also implies that any operator g is described according to equation (1.19) by a unique bit-string. This further implies that the order of G is 2^m .

The order of a stabilizer group must therefore be 2^m , where m is the number of independent generators of the group.

1.3.7 The number of stabilizer generators for a stabilizer code

For a stabilizer code, the order of its stabilizer group 2^m , and hence the number of stabilizer generators m , the number of encoded logical qubits k and the number of physical qubits n are related via a simple formula

$$m = n - k \quad (1.21)$$

When $m = n$, $k = 0$, the dimension of the codespace is $2^0 = 1$. There is then just a single state that is the $+1$ eigenstate of all stabilizer operators. We call such states *stabilizer states*. Famous stabilizer states include the Bell states, the GHZ states and the cluster states.

1.3.8 Example: Repetition code

The codeword basis states for the three-qubit repetition code are:

$$|0\rangle_L = |000\rangle \quad |1\rangle_L = |111\rangle \quad (1.22)$$

For this code, $n = 3$, $k = 1$, and thus the order of the stabilizer group must be $2^2 = 4$ with 2 generators. By inspection, the following 4 operators can be seen to leave both codeword basis states (and hence the entire codespace) invariant:

$$ZZI \quad ZIZ \quad IZZ \quad III \quad (1.23)$$

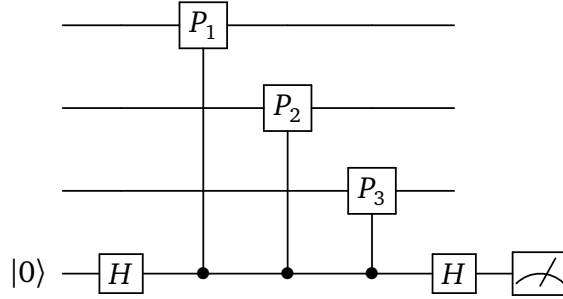
There are 4 operators, as required, the operators all commute, and any two operators chosen from ZZI , ZIZ and IZZ would make an independent generating set for the group.

1.3.9 Error detection in the stabilizer formalism

The operators in the stabilizer of the three-qubit code (except for the identity operator) are the *same* operators that are measured to detect bit-flip errors in the code. This is not a coincidence. Pauli operators are Hermitian operators and thus define measurable observables. We can thus reinterpret the definition of the stabilizer - it is the set of measurements which, error-free states will return eigenvalues $+1$. If a state has a detectable error, one or more -1 eigenvalues will be returned.

We therefore detect errors on stabilizer codes by measuring the stabilizer operators. However, we do not need to measure every operator in this 2^m order group. The group property ensures that it suffices to measure an independent set of stabilizer generators. For example, in the repetition code, it was not necessary to measure all three operators to detect and identify the three different bit-flips, measuring any two would suffice. Since the independent generating set has m elements, the number of measurements needed is modest ($n - k$) and scales linearly with the number of physical qubits.

There is a simple quantum circuit to measure arbitrary n -qubit Pauli operator $P_1 \otimes P_2 \otimes \cdots \otimes P_n$, where P_j is the Pauli operator acting on the j th qubit. It has the following form (in this example, with three qubits, and operator $P_1 \otimes P_2 \otimes P_3$):



Here the controlled gates are controlled P_j gates and the measurement is in the computational basis. The proof that a circuit of this form does indeed measure observable $P_1 \otimes P_2 \otimes \cdots \otimes P_n$ is left as an exercise. The set of outcomes of the measurement of the stabilizer generators is sometimes called the *syndrome*.

1.3.10 Constructing projectors for Pauli measurements

The eigenprojectors for n -qubit Pauli measurements have a particularly simple form. Recall that these operators have eigenvalues $+1$ and -1 . The projectors onto the $+1$ and -1 eigenspaces for the operator P , which we will denote Π_+ and Π_- have the form:

$$P_{\pm} = \frac{1}{2}(I \pm P) \quad (1.24)$$

1.3.11 Encoded logical operators in the stabilizer formalism

Above we derived logical operators for the three-qubit repetition code.

$$\bar{X} = X \otimes X \otimes X \quad \bar{Z} = Z \otimes I \otimes I. \quad (1.25)$$

and we noted that there were many equivalent alternatives, e.g. IZI , IIZ , operators.

The stabilizer formalism provides a simple characterisation of this equivalence. Consider state $|\psi\rangle$ in the codespace a stabilizer code, the encoded logical operator \bar{L} and a member of the stabilizer group S_j . Since $S_j|\psi\rangle = |\psi\rangle$ we can show immediately that

$$LS_j|\psi\rangle = L|\psi\rangle \quad (1.26)$$

and thus the operator LS_j acts identically to L on states in the codespace. This means, given an encoded operator L , there exist a family of 2^m operators, which act equivalently on the codespace. We see that IZI , IIZ can be generated from $\bar{Z} = ZII$ by multiplying it by elements of the stabilizer.

Encoded logical operators must satisfy certain properties to ensure that they leave map every state in the codespace to a state in the codespace.

By convention (and for convenience), one always constructs logical encoded Pauli operators \bar{X} , \bar{Z} out of physical Pauli operators. Recall that any pair of Pauli operators must commute or anticommute. Any Pauli operator which anticommutes with a stabilizer element cannot leave the codespace invariant (proof will follow below). Therefore, encoded logical Pauli operators must *commute* with every element of the stabilizer.

Formally, the set of Pauli operators which commute with every element of the stabilizer is called the *centraliser* of the code. In group theory, the set of elements which commutes with a sub-group is called the centralizer of this subgroup.

The commutation properties of the stabilizer and logical encoded Pauli operators provide a method for deriving logical encoded operators. The encoded logical Pauli operators must commute with all elements of the stabilizer and must satisfy the commutation / anticommutation relations of the Pauli operators which they are representing.

1.3.12 Distance of a quantum code

The distance of a quantum code is given by the minimum weight of any (non-identity) logical operator. It suffices to consider the minimal weight of any logical Pauli operator, e.g. any operator in the centralizer of the code.

For the three qubit repetition code, the centralizer is as follows

$$\begin{array}{cccc}
III & ZZI & ZIZ & IZZ \\
XXX & -YYX & -YXY & -XYY \\
YXX & XYX & XXY & -YYY \\
ZII & IZI & IIZ & ZZZ
\end{array} \tag{1.27}$$

where the four rows correspond to the logical I (which is nothing other than the stabilizer group itself), and equivalent encoded logical X , Y and Z operators. We confirm that the weight of the code is 1. This confirms that there are certain errors (Z phase-flip errors) which this code can not correct.

The smallest code which can correct both single bitflip and phaseflip errors is the 5-qubit code, which has stabilizer generators:

$$\begin{aligned} XZZXI \\ IXZZX \\ XIXZZ \\ ZXIXZ \end{aligned} \tag{1.28}$$

Its properties are explored in problem sheet 1.

1.3.13 Relationship between errors and syndromes

On page 12, we saw a table relating single qubit errors on the code to the outcome of the stabilizer measurements. We saw that the outcome of the stabilizer measurement was +1 whenever the error commuted with the measurement operator, and -1 whenever is anti-commuted. Again, this reflects a more general relationship.

Let E be a Pauli error operator which commutes with the stabilizer generator S , and let $|\psi\rangle$ be the codestate prior to the error. Since,

$$E|\psi\rangle = ES|\psi\rangle = SE|\psi\rangle \tag{1.29}$$

the post-error state $E|\psi\rangle$ is a +1 eigenstate of S .

Similarly, if E anti-commutes with S ,

$$E|\psi\rangle = ES|\psi\rangle = -SE|\psi\rangle \tag{1.30}$$

so

$$-E|\psi\rangle = SE|\psi\rangle \tag{1.31}$$

and the post-error state $E|\psi\rangle$ is a -1 eigenstate of S .

Since any two n -qubit Pauli operators must either commute or anticommute with each, and since we will only consider errors which are Pauli operators (because the measurement of the stabilizer operators projects converts more general errors into Pauli errors), the above analysis holds for all errors we will consider.

NB This analysis also explains why encoded logical Pauli operators must commute with stabilizer generators.

1.3.14 Decoders – Computing the error from the stabilizer measurement outcomes

There is a simple relationship between errors and stabilizer measurements, but it is not one-to-one. Given any error E and any element of the stabilizer S , errors E and SE will lead to identical measurement outcomes. This phenomenon is known as code degeneracy. It means the choice of the most appropriate error correction operator requires some judgement. Algorithms which automate the choice of error correction operator from a set of stabilizer measurement outcomes are called *decoders*.

This terminology is unfortunate (since they do not “decode” the encoded information) but is inherited from classical error correction theory. We will study several decoders for topological codes.

1.3.15 *Finding codeword basis states in the stabilizer formalism

While all error correcting properties of a code can be studied solely in terms of the Pauli operators in the stabilizer formalism, sometimes it can be useful to return to a state vector or density matrix description of codeword basis states. There is a straightforward way to do so.

For example, in a code-encoding a single qubit, the logical basis state $|0\rangle_L$ is the only $+1$ eigenstate of the stabilizer generators, and additionally the \bar{Z} logical operator.

The density matrix for this state (a rank-one projector) then satisfies the equation

$$|0\rangle_L\langle 0|_L = \frac{1}{2^n}(1 + \bar{Z}_L) \prod_{S_j} (I + S_j) \quad (1.32)$$

In practise, one rarely wants to write down the density matrix for a codeword, but it is reassuring that there is a simple formula to do so.

This formula makes explicit the fact that codewords are only fully defined by both the stabilizer generators and the appropriate encoded logical operator.

1.3.16 Repetition code as a stabilizer code

To define a stabilizer code, we write down a set of independent stabilizer generators plus encoded logical Pauli operators for each encoded qubit.

The three-qubit repetition code is therefore defined by stabilizer generators ZZI, IZZ , plus encode logical operators $\bar{X} = XXX$, and $\bar{Z} = ZII$.

Most quantum error correcting codes studied can be represented in the stabilizer formalism, including the Shor code, the Steane code, the 5-qubit code, all CSS codes, and the toric and planar surface codes.

1.3.17 Summary of the stabilizer formalism for quantum codes

- An $[n, k, d]$ stabilizer code is defined by $m = n - k$ independent stabilizer generators S_j , which are mutually commuting Pauli operators that generate an order 2^m Abelian group,
- together with a set of encoded logical Pauli operators.
- The encoded logical operators commute with all elements of the stabilizer.
- The encoded logical operators must also satisfy the commutation relations of the (unencoded) logical operators they represent.
- Every logical operator is a member of a set of 2^m alternative operators which act identically on the code. These can be obtained by multiplying any member of the set with the elements of the stabilizer.
- Errors are detected by measuring an independent set of stabilizer generators.
- Errors are corrected by applying a correction operator. A suitable correction operator is calculated from stabilizer measurement outcomes by an algorithm called a decoder.
- The distance d of the code is given by the minimal weight of all (non-identity) encoded logical Pauli operators.

2 The Toric Code

The *toric code* is the simplest example of a *topological code*, or *topological surface code*. It was introduced by Alexei Kitaev¹ in 1996.

2.1 Defining the code

2.1.1 Qubits on a toric lattice

The toric code is defined in terms of a square lattice with periodic boundary conditions. Consider an $L \times L$ square lattice, wrapped around a torus, such that the right-most edge is identified with the left most edge and the upper edge identified with the lower edge (see figure 2.2). The lattice consists of

¹A. Y. Kitaev, *Fault-tolerant quantum computation by anyons*, arXiv:quant-ph/9707021.

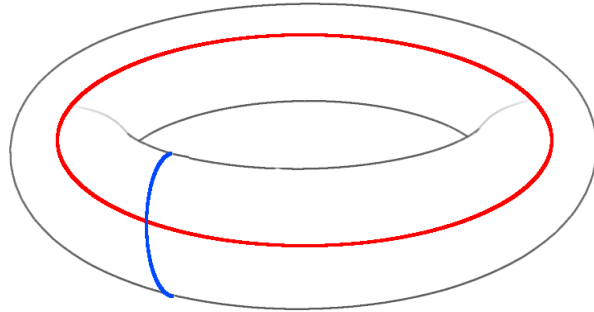


Figure 2.1: A torus. The two loops indicated in red and blue cannot be continuously deformed to a point or to each other, a topological feature reflected in the encoded logical operators of the toric code.

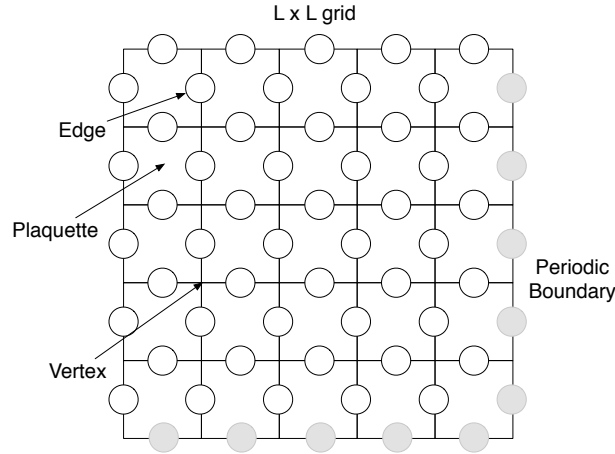


Figure 2.2: The toric code is defined in terms of an $L \times L$ lattice (here $L = 5$) with periodic boundary conditions. Edges, plaquettes and vertices are all important concepts. Edges have circles embedded on them, which will represent qubits.

edges, *vertices* (points where edges meet) and *plaquettes* (individual tiles enclosed by a set of edges - here squares). Each of these will play an important role in the description of the toric code.

We associate a qubit with every *edge* on the lattice (these are indicated by circles on figure 2.2). On an $L \times L$ grid with periodic boundary conditions there are $2L^2$ edges (verify this), where we have taken care that the edges at the boundary are not counted twice (in the figure, the second representation of an edge is coloured grey, so we can count the white circles). A toric code has therefore $n = 2L^2$ physical qubits.

2.1.2 Stabilizer generators

An n -qubit stabilizer code encoding k qubits is defined by specifying $m = n - k$ independent stabilizer generators, together with encoded logical \bar{X} and \bar{Z} qubits.

On the toric code, there are two types of stabilizer generators, associated with the *plaquettes* and *vertices* of the lattice.

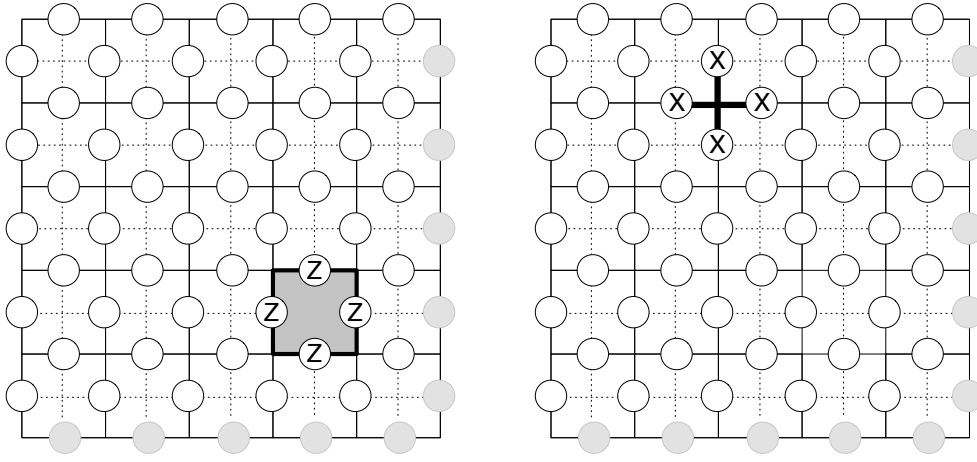


Figure 2.3: Examples of a plaquette and a vertex operator. In each case the respective vertex and plaquette is shaded, and the non-identity factors of each operator is marked on the qubits on which they act. The set of these operators for all (but one) plaquettes and vertices on the lattice are the stabilizer generators for the toric code. The dashed line indicated the dual lattice (see below).

2.1.3 Plaquette operators

For every plaquette, we define a plaquette operator, consisting of a tensor product of Pauli Z operators acting on the 4 qubits which lie on the boundary of the plaquette (see figure 2.3), with identity operators acting on all other qubits. On an $L \times L$ grid there are L^2 plaquettes.

2.1.4 Vertex operators

For every vertex, we define a vertex operator, consisting of a tensor product of Pauli X operators acting on the 4 qubits adjacent to a vertex (see figure 2.3), with identity operators acting on all other qubits. On an $L \times L$ grid there are L^2 vertices.

NB A useful mnemonic to help you remember that the vertex operators consist of X Pauli's is that vertex contains the letter x .

2.1.5 Commutation of plaquette and vertex operators

Stabilizer generators must all mutually commute. The plaquette operators trivially commute with other plaquette operators, and the vertex operators must with other vertex operators, since all are built of Z and X respectively.

Operators associated with plaquettes and vertices which are not neighbouring commute, since they act non-trivially on different qubits. To see why neighbouring plaquette and vertex operators commute, we recognise that a vertex at a corner of a plaquette is adjacent to two edges of the plaquette. Thus the plaquette and vertex operators will overlap on two qubits. Although X and Z anticommute, the products $X \otimes X$ and $Z \otimes Z$ commute. Therefore neighbouring plaquette and vertex operators commute, and thus all plaquette and vertex operators commute with one another.

It is worth noting that this reasoning holds for plaquettes of any shape, and one can construct toric codes on non-square lattices where the same commutation argument will hold.

2.1.6 Multiplication and independence of plaquette operators

We generate the full stabilizer by multiplying together generator operators. Let us consider the effect of multiplying together plaquette operators. This multiplication can be understood via a simple rule involving the joint boundary of the multiplied plaquettes.

Consider two adjacent plaquette operators. There is one qubit in common between them, where both operators coincide with a Z . If we multiply these operators together, the two Z 's on the shared boundary qubit will multiply together to the identity ($Z^2 = I$). Hence the result operator will be 6 Z 's around the joint boundary of the two plaquettes.

This reflects a more general property of plaquette operator multiplication. Any two plaquettes will have a single common boundary edge or no common boundary edge. Thus multiplying together any set of plaquettes will lead to cancellations of Z s at the shared boundaries. The result will be that the product of the operators will consist of the overall boundary of the plaquettes in the product. For an illustration of this, see figure (2.4).

There is a deep relationship between the concept of *boundary* and the multiplication properties of the operators. *Homology* is the name for the branch of topology which is concerned with the boundaries of objects. We

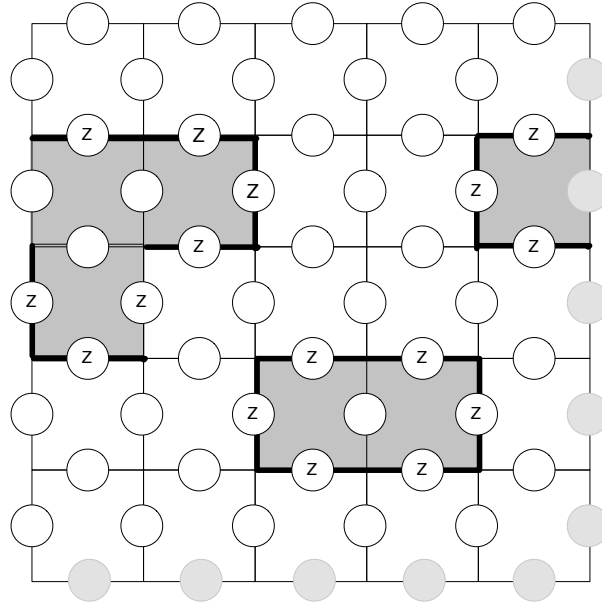


Figure 2.4: When we multiply a set of plaquette operators, the resultant operator will consist of Z operators acting on the boundary of the combined plaquettes.

will see that homological ideas provide a clean and concise language for understanding the properties of the toric code.

Consider now the product of *all* plaquettes in the entire lattice. What is its boundary? The plaquettes now cover the entire surface of the torus and have no boundary (see figure 2.5). Thus, the product of all plaquette operators is the identity. This means that the full set of plaquettes is not a set of independent operators.

Recall, we would like our generating set to be independent, as defined above in equation (1.20). This means it should be impossible to construct the identity from any product of the generators. In order to create an independent set of plaquettes, the resolution is simple. We remove any single plaquette from the generator set. There is then no longer a product of such plaquettes with zero boundary, and the operators are thus independent. We therefore count $L^2 - 1$ independent plaquette operators.

Note that boundaries of sets of plaquettes always form closed loops of Z

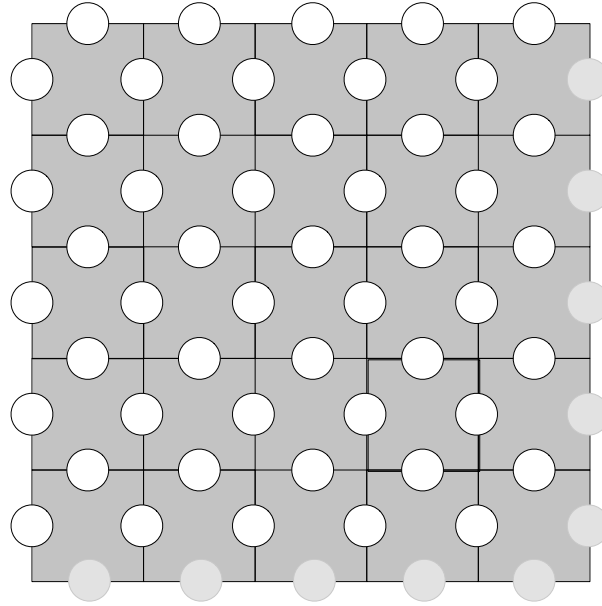


Figure 2.5: If we multiply all plaquettes together, the resultant area covers the entire torus and has no boundary. All of the Z operators in the generators have cancelled and the resultant operator is the identity operator. Note that this argument, in terms of boundaries, is valid for any sized lattice on the torus (and also holds true for other boundaryless surfaces such as the double torus).

operators.

2.1.7 Dual lattice

Let us call our initial lattice, the *primal lattice*. It is possible to create a *dual lattice* by shifting our lattice by half a cell, down and to the right. The dual lattice has the same size and same boundary conditions as the original lattice, but every plaquette in the primal lattice has become a vertex in the dual, and vice versa. The edges of the dual lattice are printed with dashed lines in figure 2.3. Notice that edges centred in the same location as before (and thus still refer to the same qubits), but their orientation in the dual lattice is perpendicular to the primal lattice.

Lattice duality is a powerful tool in the study of the toric code. Since all

vertex operators are plaquette operators in the dual lattice, and vice-versa we can choose to describe them in whichever lattice is best suited for our calculation. For example, multiplication of operators is best understood in the plaquette picture.

In general, the dual of any lattice is found by interchanging plaquettes with vertices, and reorienting edges accordingly. This will usually change the underlying lattice structure (e.g. the dual of the hexagonal lattice is the triangular lattice). It is an unusual property of the square lattice that its dual is also a square lattice. We say that it is *self-dual*, a property which makes the square lattice well-suited for defining topological codes where X and Z errors are equally protected.

The dual lattice will be indicated on figures with a dashed line (see, e.g. figure 2.2).

2.1.8 Multiplication and independence of vertex operators

Using lattice duality, the vertex operators can be defined as the boundary of the corresponding plaquette in the dual lattice. Multiplication of vertex operators can be described in terms of the common boundary of plaquettes in the dual lattice, exactly analogously to the plaquette operators above.

Via the same arguments as for the plaquette operators there are therefore the product of all vertex operators is the identity, but by removing one vertex operator we achieve an independent set of generators with $L^2 - 1$ elements.

2.1.9 Encoded qubits

There are $2L^2$ physical qubits in the code, $L^2 - 1$ plaquette and $L^2 - 1$ vertex operators. There are therefore $2L^2 - 2(L^2) = 2$ encoded qubits. To complete the code description, we must identify the encoded logical operators $\bar{Z}_1, \bar{X}_1, \bar{Z}_2, \bar{X}_2$.

Recall that these operators must commute with all elements of the stabilizer, must not be elements of the stabilizer themselves, and must satisfy the commutation and anti-commutation properties of X and Z .

First we seek a product of Z operators will commute with the stabilizer, but is not a stabilizer operator itself. Consider a single Z operator acting on a single qubit in the code. This operator will commute with all plaquette operators (since they are formed from Z operators) so we should consider the vertex operators. All vertex operators except those immediately neighbouring

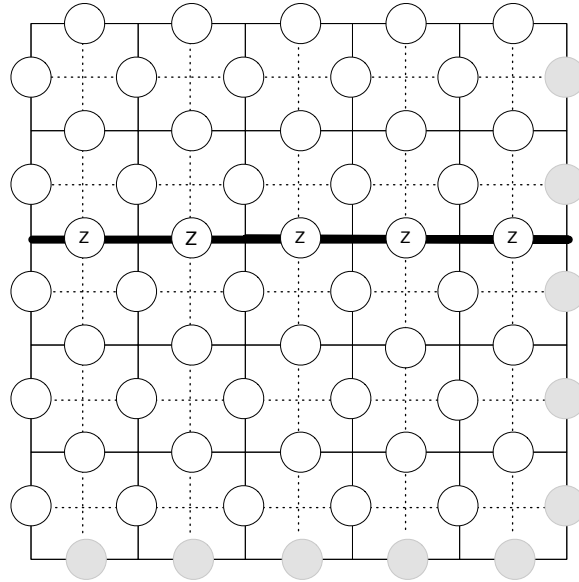


Figure 2.6: An encoded logical Z operator on the toric code. The operator forms a closed loop of Z operators along a sequence of edges on the primal lattice.

the qubit must commute with the Z , since they act on different qubits. This leaves only the vertex operators immediately adjacent to the qubit. These two operators anticommute with the single Z . Thus this single Z cannot be the logical operator we seek.

If we consider a pair of adjacent Z 's, they will now commute with one of these two vertex stabilizers, but there will now be an extra vertex operator which anticommutes. Indeed if we form a string of Z operators, regardless of its shape, it will always anticommute with the vertex operators at the ends of the string.

The only solution is to find a string of operators which has no end - a loop! We have already seen that the stabilizer contains closed loops of Z s, products of plaquette operators generate closed loops around their boundary. Are there any other possible closed loops on the toric lattice?

Yes, there are. Consider a closed loop which winds around the torus, such as the red and blue loops in figure 2.1. Such loops are closed, but are of a

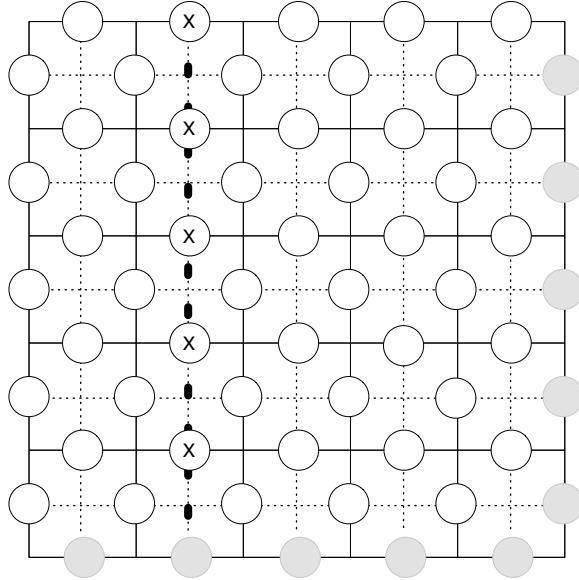


Figure 2.7: An encoded logical X operator on the toric code. The operator forms a closed loop of X operators along a sequence of edges on the dual lattice.

different type to those we have considered so far. Convince yourself that a loop of Z operators, which winds around the torus, as illustrated in figure 2.6 cannot be generated as a product of plaquette operators. Let us label this loop of operators \bar{Z}_1 , and identify it with the encoded Z for the first encoded qubit.

We must now identify \bar{X}_1 . This operator must anti-commute with \bar{Z}_1 , yet commute with all elements of the stabilizer. Consider a loop of X operators on the *dual lattice*, perpendicular to the loop defining Z_1 and illustrated in figure 2.7. From lattice duality, this operator must commute with all stabilizer elements via an analogous argument to the logical Z. The operator anticommutes with \bar{Z}_1 since they share only one qubit on which they act non-trivially.

We now turn to the second encoded qubit. If we rotate the loops corresponding to \bar{Z}_1 and \bar{X}_1 90 degrees, we obtain two new operators which commute with \bar{Z}_1 and \bar{X}_1 , commute with the stabilizer, and anticommute with

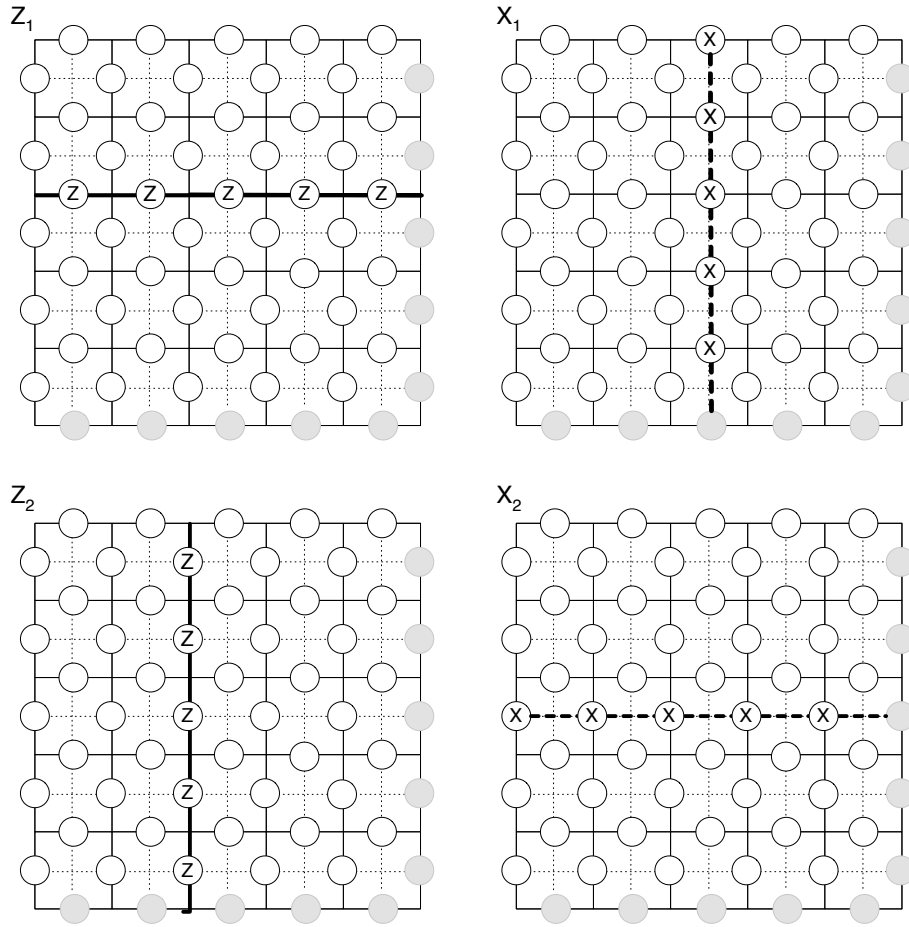


Figure 2.8: Encoded logical operators $\bar{Z}_1, \bar{X}_1, \bar{Z}_2$ and \bar{X}_2 for the two encoded qubits in the toric code.

each other. These form the logical operators \bar{Z}_2 and \bar{X}_2 . The four operators are illustrated in figure 2.8.

2.1.10 Equivalence of logical operators under stabilizer multiplication

As we saw in part 1, encoded logical operators are not unique, and one can generate an equivalence class of operators which act identically on the code space, by multiplying a logical operator with elements of the stabilizer.

Notice that the logical operators and stabilizer generators all consist solely of Z operators or solely of X operators. Here we shall consider only effect of multiplying a logical Z operator with a plaquette stabilizer.

Let us multiply the \tilde{Z}_1 operator with a plaquette operator adjacent to it. We see that the resultant operator is still a loop around the torus, and has been diverted around the plaquette from its original path (see figure 2.9).

This property holds generically. Multiplication by any stabilizer operator will divert the original course of the operator, but does not change the fact that it loops around the torus (see further examples figure 2.9). Precisely this sort of equivalence is studied in homology theory, which we will return to later.

The same behaviour (on the dual lattice) is seen when we multiply logical X operators with vertex operators (which are plaquettes on the dual lattice).

2.1.11 Code distance

The code distance is the minimum weight of a logical operator in the code. It corresponds to the lowest weight of any undetectable error. The arguments above show us that since a logical operator must wind around the torus, the lowest weight logical operator must be L , the dimension of the torus. Hence the code distance for a toric code on an $L \times L$ lattice is L , and its code parameters are $(n = 2L^2, k = 2, d = L)$.

This implies that the code might be more robust against errors if we increase the size of the lattice. We will consider this in more detail (and show the type and regime of errors for which such reasoning is valid) below when we consider error correction thresholds for the toric code.

2.2 Errors on the toric code

2.2.1 Error detection

Error detection in the stabilizer formalism is discussed in part 1 above. To recap, it suffices to consider error operators E , that are n -qubit Pauli operators. In the stabilizer formalism, errors are detected by measuring the stabilizer generators. If no error has occurred, these measurements will all output the $+1$ eigenvalue. If error E has occurred, the stabilizer generators which anti-commute with E will output -1 . We call the output of the stabilizer measurements the *error syndrome*.

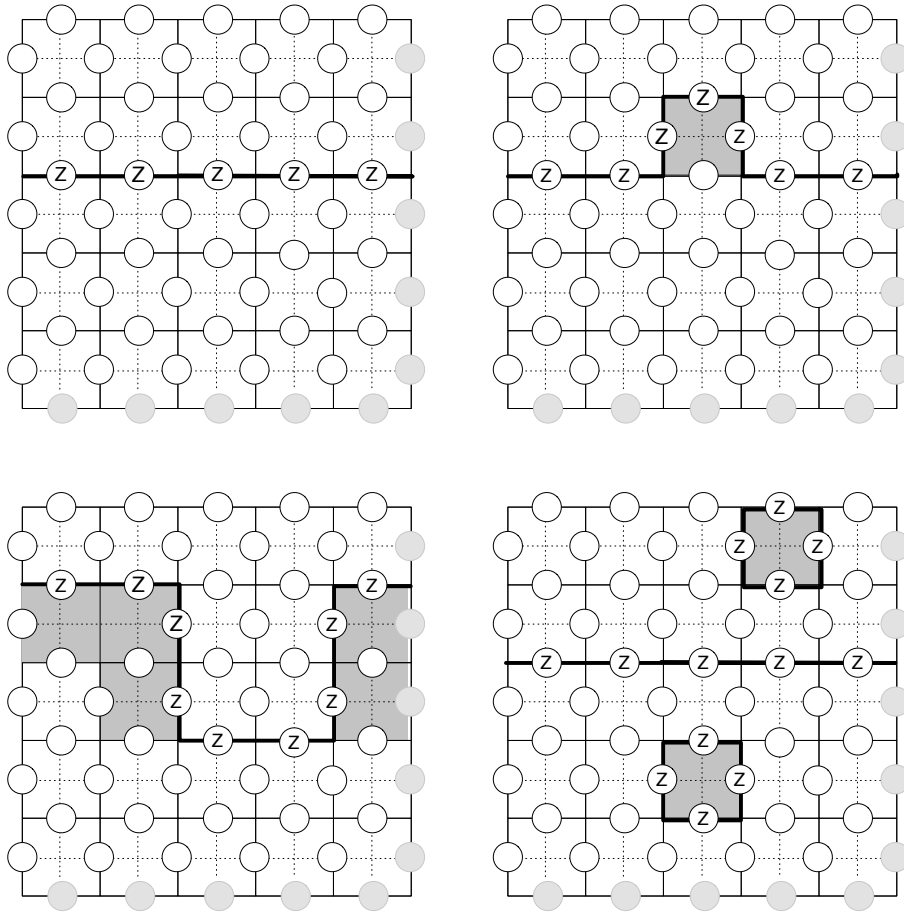


Figure 2.9: An encoded logical Z and the equivalent operators obtained by multiplication with plaquette operators (shaded grey).

We first consider a Z error on a single qubit. Which stabilizer generators anticommute with it? All plaquette operators commute with it, as do all vertex operators which do not act on that qubit. The only stabilizer generators which anticommute with the error operator are the vertex operators immediately adjacent to it (see figure 2.10). Hence this single error has flipped the output value of 2 stabilizer measurements.

Now consider two Z errors on adjacent qubits. The vertex operator between the two errors now *commutes* with the error operator, since $X \otimes X$ commutes with $Z \otimes Z$. The only vertex operators whose outcomes are flipped to -1 are those at the two ends of the string of errors.

This holds in general. Given any string of errors on the primal lattice, the only stabilizer generators have their outcomes flipped are the vertices at the two ends of the string. (The ends of a string can be considered its “boundary”, so yet again boundaries are playing a role. This will be formalised when we discuss homology later).

Detection of X errors occurs in an entirely analogous way. Strings of X errors on the *dual* lattice trigger -1 measurement outcomes on the plaquette operators corresponding to the vertices in the dual lattice at the ends of the strings.

Since every Pauli operator can be written as a product of Z and X operators, and since the two classes of errors independently affect different types of stabilizer measurement (vertices and plaquettes independently), we can consider them as two separate error correction processes. The two processes are analogous, up to lattice duality, so for the remainder of this section we shall largely focus on Z errors and vertex stabilizer measurements.

2.2.2 Error correction

To correct errors, we apply the inverse operator of the error. In the case of self-inverse Pauli errors we may apply the same operator. Therefore the main task in error correction is identification of the correction operator to apply given the syndrome - the set of measurement outcomes from the stabilizer generators.

The relationship between syndrome and error is never one-to-one² - a

²Although all stabilizer codes exhibit this degeneracy, there are a family of codes that are such called *non-degenerate codes*. We call a code non-degenerate, when for the subset of errors of weight strictly smaller than half the code distance $\leq \lfloor (d-1)/2 \rfloor$, the mapping from error to syndrome is a one-to-one. For example, the classical repetition code is non-

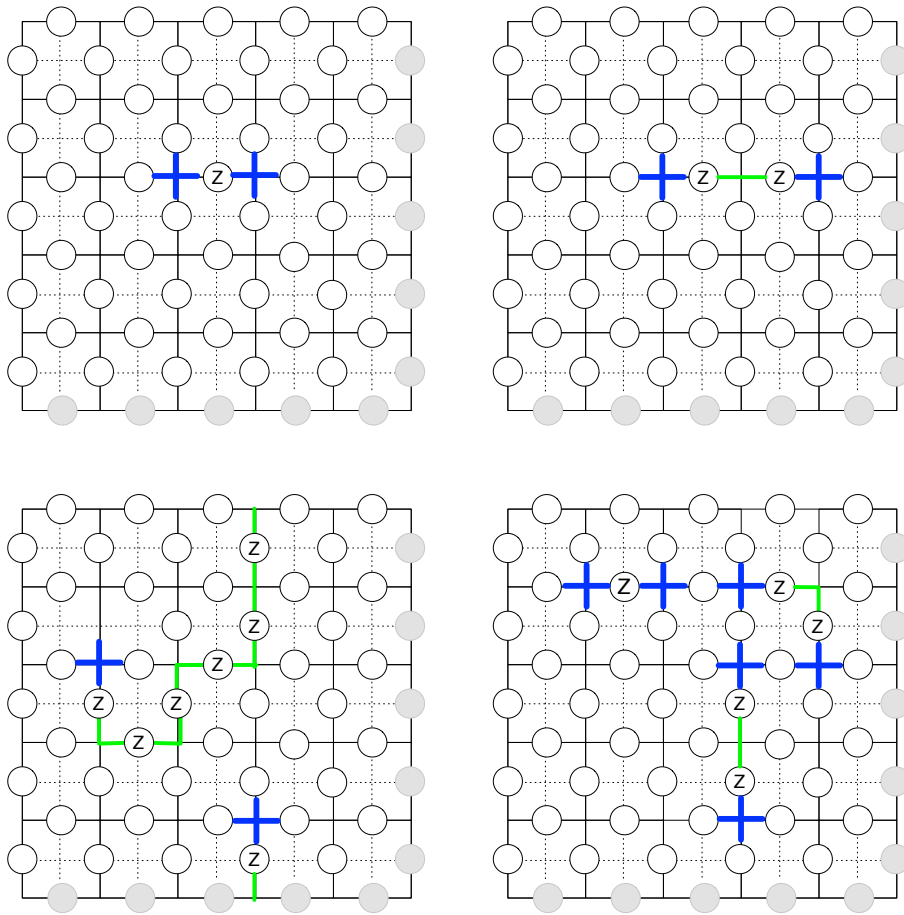


Figure 2.10: Strings (green) of Z -errors on the primal lattice create -1 measurement outcomes at the vertices (blue) at the end of each string.

property we call degeneracy. Two Pauli error operators E and E' will lead to the same syndrome if they satisfy $E = E'L$ where L is any operator which commutes with the stabilizer. In other words, where L is a member of the equivalence class of logical operators.

For example, in the first two panes of figure 2.11 you see two strings of errors which, since they have the same end-points, correspond to the same error syndrome (the same stabilizer measurement outcomes).

If L is in the stabilizer of the code, then the product, $E'E = L$ (remember that E' is self-inverse) is in the stabilizer. Therefore an error E followed by correction E' will combine to generate a stabilizer operator, leaving the code invariant. In other words, if $E'E$ is in the stabilizer, E' will correct error E . This is illustrated in the third pane of figure 2.11.

However, the encoded logical operators also commute with the stabilizer. Therefore, (considering here only Z errors) given error E , $E\bar{Z}_1$, $E\bar{Z}_2$ and $E\bar{Z}_1\bar{Z}_2$ would all lead to identical error syndromes, as illustrated in figure 2.12.

For a given syndrome, then, the error operators which could have generated it fall into four equivalence classes. Given single error operator E , the four equivalence classes can be obtained by multiplying E with operators I , \bar{Z}_1 , \bar{Z}_2 and $\bar{Z}_1\bar{Z}_2$ and then by any element in the stabilizer consisting solely of products of plaquette operators.

It is therefore a non-trivial task to choose the most appropriate correction chain. Algorithms which perform this task are called *decoders*. This choice, however, will depend strongly on the particular error model, so it is this consider next.

2.2.3 Quasi-particle picture

The syndrome consists of a set of measurement outcomes $+1$ and -1 . It can be helpful to think of syndrome outcomes as a set of *quasi-particles* associated with the plaquettes and vertices of the lattice. When the outcome is $+1$ we say that the quasi-particle has charge 0 (and we usually think of a charge 0 particle as being no particle at all), when the outcome is -1 , the associated quasi-particle has charge 1. This may seem a strange convention, at first sight, but there are a number of ways in which the syndrome outcomes do behave like in a particle-like way.

degenerate in this sense.

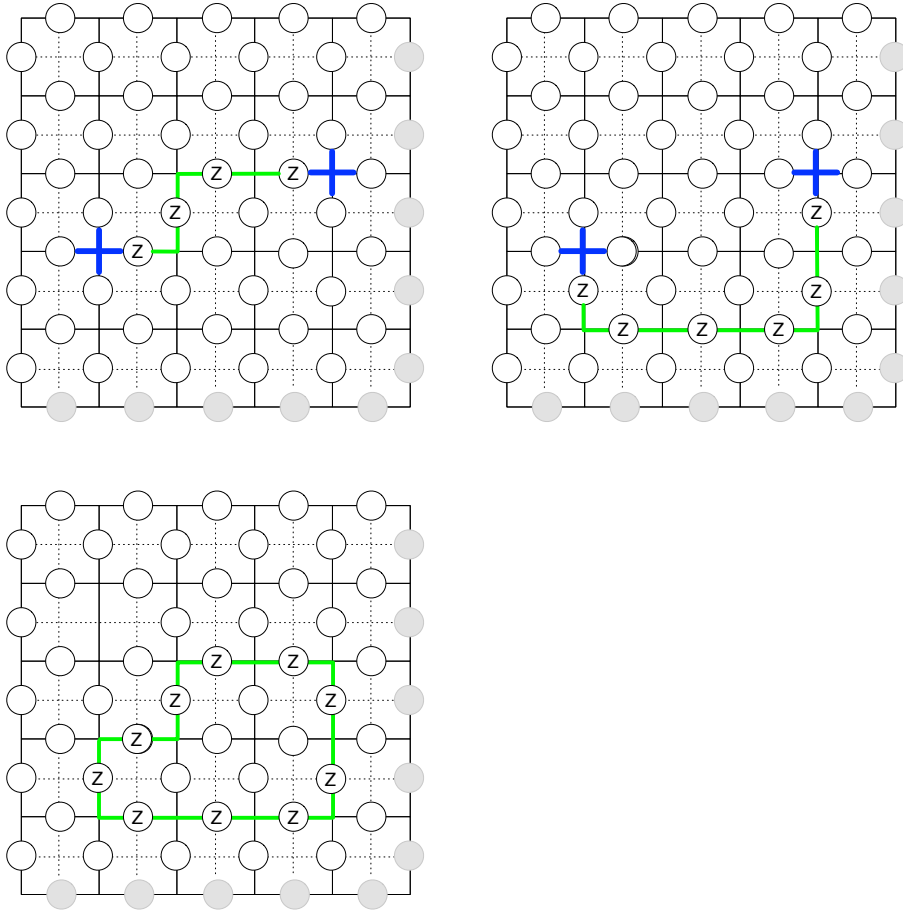


Figure 2.11: Two strings (green) of Z-errors on the primal lattice, which correspond to the same syndrome. Their product (third pane) is in the stabilizer (it is a closed loop which encloses a set of plaquettes), thus applying the second operator would correct the first error and vice-versa.

- **Pair creation:** A Z error on an error-free code state creates a pair of $+1$ charged quasi-particles on the adjacent vertices.
- **Quasi-particle movement:** A Z error adjacent to a $+1$ quasi-particle will move the quasi-particle to the other vertex adjacent to the error.
- **Pair annihilation:** If two quasi-particles are moved onto the same lattice vertex (or plaquette), they will annihilate. The lattice vertex (plaquette) will be left in a charge 0 state.

The quasiparticle picture gives a convenient language for understanding the error creation and correction process. Errors create quasi-particles, and are corrected by annihilating these quasi-particles along the correct paths.

The existence of this quasi-particle picture is not a coincidence. The construction of the toric code was developed by Kitaev out of quasi-particle models (called quantum double models) which had been studied in theoretical physics since the 1980s. In the toric code literature, following the conventions of quantum double models, vertex quasi-particles are sometimes called electric charges while plaquette particles are called magnetic charges.

2.2.4 Error models

The error correction capabilities of a quantum error correcting code depend upon the kind and strength of errors that the qubits are undergoing. The simplest noise models assume that every qubit is undergoing the same noise process (sometimes called a “noise channel”), and that this noise process is independent. Such noise processes are called independent and identically distributed (i.i.d.).

In particular, two errors models are most widely studied in the context of topological codes.

In the **depolarising noise model**, each qubit undergoes an error according to the following probabilities:

- $(1 - p)$: I (i.e. no error)
- $p/3$: X
- $p/3$: Y
- $p/3$: Z

where p is a parameter between 0 and 1. This model is symmetric between X , Y and Z . However, from the point of view of the toric code, a Y error is a combination of both an X and a Z error, so the depolarising model looks like:

- $(1 - p)$: I (i.e. no error)
- $p/3$: X
- $p/3$: both X and a Z error on the same qubit
- $p/3$: Z

To optimally decode therefore, one would need to take into account correlations between X and Z errors, and analyse both vertex and plaquette syndromes together.

From the perspective of the toric code, an easier to analyse, and therefore more widely studied, model is the **uncorrelated noise model**, otherwise known as the **independent noise model**. In this model, single qubit errors occur independently according to the distribution:

- $(1 - p)^2$: I (i.e. no error)
- $p(1 - p)$: X
- p^2 : Y
- $p(1 - p)$: Z

This process is equivalent to a combination of two separate and independent processes:

- $(1 - p)$: I (i.e. no error)
- p : X

and

- $(1 - p)$: I (i.e. no error)
- p : Z

In the uncorrelated noise model, the two types of errors are independent. Therefore we gain no advantage from studying the two types of syndrome (vertex and plaquette) together, and can study them independently. Since it is the simplest model for analysis, we shall largely consider the uncorrelated noise model in this course.

2.2.5 Code thresholds

We have seen that the code distance of the toric code on an $L \times L$ lattice is L . Therefore you'd expect that increasing L might increase the robustness of the code. However, for an independent noise-model increasing L also increases the number of errors, and the added complexity may make it harder to choose a suitable correction operator.

In practise, there is a trade-off between these two effects. When the error rate is low enough, increasing L will increase the probability of successful error correction. When the error rate is high, increasing L will decrease the probability of successful error correction. The point of transition between these behaviours is called the *code threshold*.

The probability of successful error correction will depend upon the decoder (the algorithm which computes the correction operator to apply) used. Different decoders can therefore have different thresholds. We call the threshold for the best decoder possible the *optimal threshold*.

For the toric code under the uncorrelated noise error model, the optimal error threshold has been estimated to be approximately 11%.

2.2.6 Optimal Decoder

To correct the error E , we can apply any correction operator C which is in the same equivalence class as E , i.e. $C = ES$ for some element of the stabilizer S . If we apply a correction operator C in a different equivalence class, i.e. such that $C = E\bar{U}S$ where $U \in \{\bar{Z}_1, \bar{Z}_2, \bar{Z}_1\bar{Z}_2\}$, while the system will be returned to the code-space, the data will have been corrupted with an undetectable logical error, \bar{Z}_1 , \bar{Z}_2 or $\bar{Z}_1\bar{Z}_2$. If a decoder outputs a correction operator in a different equivalence class we consider it a failure.

An optimal decoder is one which, upon analysing the probabilities of all possible errors consistent with the observed syndrome, given the error model, computes the equivalence class of each operator, and the outputs a correction operator in the most likely equivalence class.

The optimal decoder is not efficient, the number of possible error operators scales exponentially in L (it is equal to the number of operators in the centralizer of the stabilizer group). It is therefore not a decoder one would use in practise. Nevertheless, it is important since its threshold sets an upper-bound on, and a benchmark for comparison with the thresholds of efficient decoders.

The threshold for the optimal decoder for the toric code, under the uncorrelated noise error model, is approximately $p = 0.11$. This value was obtained using a very elegant method, by mapping the error model and optimal decoder properties onto a statistical mechanics model that had already studied by Physicists - the “random bond Ising model”. The threshold value corresponds to a phase transition point in this model - a value that only already been computed when the toric code optimal decoder was first proposed by Dennis et al (see citation in the footnote below). We will not cover this model and its derivation in this course³.

For values of p greater than the error threshold, as we make the lattice size bigger, the overall probabilities of the 4 equivalence classes or errors become more equal, so the optimal decoder performs less well. Below the threshold, as L becomes larger a single equivalence class becomes dominant, and the probability of the other equivalence classes falls to zero. Hence, in this regime, by making L larger, the optimal decoder probability will tend to unity.

The optimal decoder threshold for the toric code has also been estimated (also by mapping to a statistical mechanics model ⁴) to be at $p = 18.9\%$. We must be cautious when comparing the thresholds between these two models, since in the depolarising model, the probability of no error on a qubit is $(1 - p)$, whereas in the uncorrelated model it is $(1 - p)^2$. It is thus incorrect to interpret these threshold values as telling us the toric code is more robust to depolarising noise than uncorrelated errors.

³Those interested will find details of the model in Eric Dennis, et al, *Topological Quantum Memory*, arXiv:quant-ph/0110143, part IV and in Chenyang Wang, Jim Harrington and John Preskill, *Confinement-Higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory*, arXiv:quant-ph/0207088 (Don't be put off by the jargon-heavy title! It reflects the interdisciplinary nature of the methods that arise in this research area, with lots of interplay between different branches of theoretical physics.).

⁴H. Bombin, et al, Strong Resilience of Topological Codes to Depolarization arXiv:1202.1852 [quant-ph]

2.2.7 Minimum Weight Perfect Matching

The most widely used efficient decoder algorithm for the toric code is based on **Minimum Weight Perfect Matching**. Minimum weight perfect matching (MWPM) is a problem from graph theory. It was first studied in the 1950s, and an efficient algorithm was proposed for its solution by Jack Edmonds in the 1960s⁵.

For the uncorrelated noise model, the Edmond's MWPM algorithm allows us to compute, given the error syndrome (i.e. the set of outputs of the stabilizer generator measurements) the most likely error compatible with the measured syndrome.

Recall that in the uncorrelated noise model, we can treat and decode X and Z type errors separately. Consider n qubits, undergoing Z errors independently with probability p . The probability of obtaining a given weight m operator is simply $p^m(1-p)^{m-n}$. Hence the most likely error is the error with the lowest weight.

Edmund's algorithm for solving MWPM (known as the Blossom algorithm) can be used to calculate the lowest weight error consistent with an error syndrome.

Recall that the syndrome consists of a set of measurement outcomes $+1$ and -1 . It is helpful to work in the quasi-particle picture (see above). In that picture, the error syndrome consists of pairs of quasi-particles. To correct the errors, we pair the syndromes up and annihilate them. To find the lowest weight correction operator we cast the problem as a minimum weight perfect matching problem on a graph.

We create a graph whose vertexes correspond to the set of $+1$ charge quasi-particles in the error syndrome (recall we are only considering Z errors and vertex stabilizers here). We then create edges between each vertex and every other vertex, i.e. we create the complete graph on the vertices. We now assign a weight to every edge of the graph. The weight assigned is the minimum taxi-cab distance (i.e. the number of edges in the path) between the corresponding vertex operators on the lattice.

The lowest weight error correction operator for the syndrome now corresponds to the Minimum Weight Perfect Matching on the graph. In graph theory, a **matching** on a graph is a sub-graph where every vertex is connected to at most one edge. In other words we delete edges until no more than one

⁵Jack Edmonds, *Paths, Trees and Flowers*, *Canadian Journal of mathematics*, **17.3** 449 (1965)

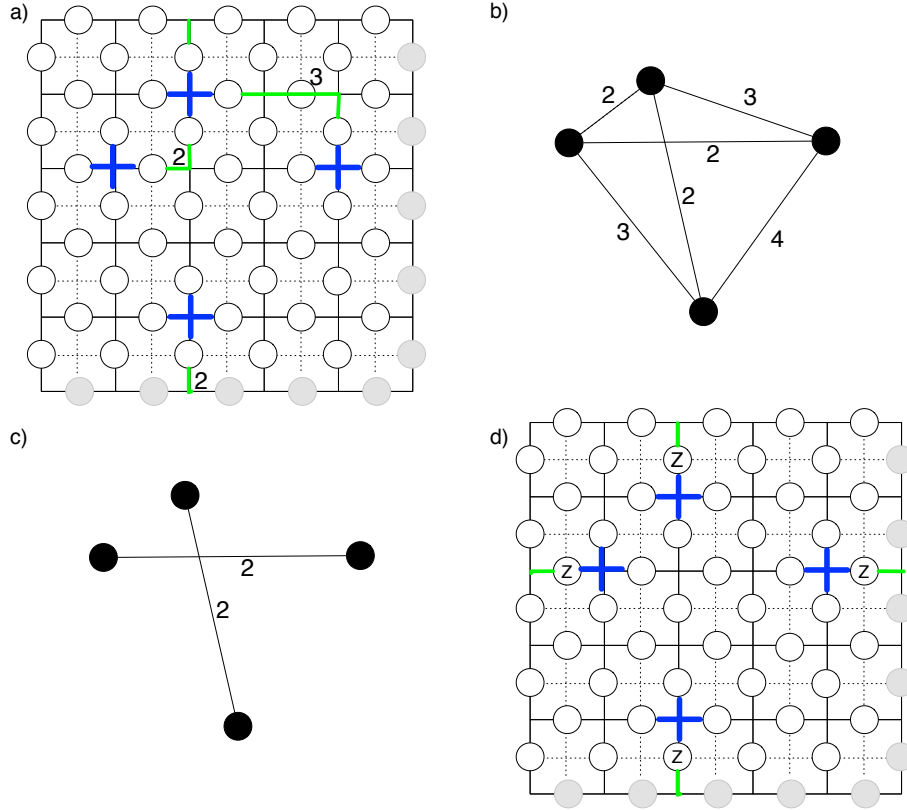


Figure 2.13: An illustration of using a MWPM algorithm to find the minimal weight error operator consistent with a syndrome. In a) shortest paths, together with the taxi-cab distance between the top quasiparticle and the other three are highlighted (the other three paths are omitted, for clarity). b) The graph of all taxi-cab distances between quasi-particles. c) The minimum weight perfect matching for the graph. d) The corresponding correction operator.

edge is attached to each vertex. This has the effect of leaving vertices either paired or isolated. A **perfect matching** is a matching where every vertex is paired. A **minimal weight perfect matching** is a perfect matching where the sum of the weights of the edges on the matching is minimized. Implementations of Edmonds' algorithm can be readily found online in c++, Fortran and Python.

Given the graph of the syndrome quasi-particles weighted by the taxi-cab length of the minimum path between them, the MWPM algorithm will return a matching which corresponds to the minimal weight correction operator corresponding to the syndrome. This process is illustrated in figure 2.13.

The MWPM algorithm returns the most likely error as a correction chain. It is therefore a very successful decoder. It is not, however, optimal, since it can occur that the most likely error operator is not within the most likely equivalence class of errors. For this reason the threshold of the MWPM algorithm is slightly less than the optimal threshold, and has been numerically estimated to be 10.3%.

Unlike the optimal decoder, however, the MWPM algorithm is efficient. For a complete graph, the runtime of Edmonds algorithm scale with V^3 , where V is the number of vertices in the graph, e.g. the number of syndrome quasi-particles, giving a worst-case scaling of L^6 in the size of the code.

Several decoder algorithms have recently been proposed which run more efficiently than MWPM, but with a reduced code threshold.

3 Elements of Topology and Homology

Topology is the branch of mathematics which – loosely speaking – studies *global structural* properties of objects that are independent of *local details* – and thus are preserved under local deformations such as stretching, bending and shrinking. It thus studies certain structural properties of objects rather than the specific details of their shape.

The field has a long history – Euler is credited as founding the field with his solution to the *Bridges of Königsberg* problem, which abstracted the road and bridge layout of 18th century Königsberg to a graph – and has become a cornerstone of modern mathematics, and important in a number of areas of theoretical physics.

In the context of this course, topological ideas formed the basis of the toric code construction, and it is through *topology* (in particular, a set of topological techniques called *homology*) that the toric code can be most compactly described and most readily generalised. In particular, in topological



Figure 3.1: To solve the *Seven Bridges of Königsberg* problem, Euler abstracted the topological layout of the bridges to a graph. Images from http://en.wikipedia.org/wiki/Seven_Bridges_of_Koenigsberg.

codes we encode data in *global* degrees of freedom which are robust against *local* errors. In this part of the course, there will be a brief introduction to topology and homology, focussing, in particular, on those elements most important to topological codes.

The “Seven Bridges of Königsberg” problem was the question of whether or not it was possible to cross all seven bridges in the medieval city (see figure 3.1) on a single continuous walk without crossing any of the bridges twice. Euler proved the impossibility of this task by abstracting the network of bridges to a graph, and proving that such for such a walk to be possible, a maximum of two vertices of the graph should be of odd degree (the only nodes allowed to be odd degree would represent the start and end points of the walk). In Euler’s graph all four vertices are of odd degree, proving the impossibility of the walk.

Euler’s proof succeeds because the proof does not rely on the precise positions and geometry of the bridges. The graph captures the essential topological features of the network of bridges needed to solve the problem.

3.1 Topological equivalence and topological invariants

3.1.1 Homeomorphism

A central idea of topology is that certain objects, although appearing superficially different, can share the same essential topological features. Such *topological equivalence* can be made precise in several ways. One of the most important kinds of topological equivalence is equivalence under *homeomorphism*.

Homeomorphism (the first of three similar sounding – but crucially different – topological terms beginning with “h”, and, as Wikipedia says, “not to be confused with homomorphism”) is the continuous distortion of an object via an invertible map. Loosely speaking, homeomorphism of an object can include stretching, bending, but not (in general) tearing or glueing.

Homeomorphism: A continuous function between two objects or topological spaces which

- is continuous
- is a bijection (i.e. one-to-one and onto, and hence invertible)
- has a continuous inverse function.

Homeomorphic objects can therefore be mapped to one another by continuous stretching and bending, but also by cutting an object, deforming it and reattaching it precisely where it was cut (since such a cut and glue preserves the continuity and reversibility of the mapping). Apart from this special case of cutting and precise reattachment, tearing and glueing are not allowed since they break the continuity of the map (or its inverse). Note that homeomorphism also does not include continuous deformation of a finite region to a point, since this mapping is not a bijection.

Some examples of homeomorphic equivalence (and inequivalence) are illustrated in figure 3.2. When we say two objects are “topologically equivalent”, we usually mean that they are equivalent *up to homeomorphism*.

3.1.2 Topological invariants

Mathematicians often study highly baroque structures and it can be challenging to identify whether or not two objects are homeomorphic. *Topological invariants* are properties or numerical quantities which remain unchanged under homeomorphism. Therefore such invariants can provide a simple way to demonstrate homeomorphic inequivalence. A topological invariant of particular importance to topological codes is the *Euler characteristic*.

3.1.3 Euler characteristic

In his study of polyhedra, Euler realised (and proved) that all convex polyhedra objects satisfy the following equation:

$$F - E + V = 2 \quad (3.1)$$

where F is the number of faces, E , the number of edges and V the number of vertices.

For example, a cube has 6 faces, 12 edges and 8 vertices ($6 - 12 + 8 = 2$). See figure 3.3 for more examples.

In general, we call the quantity

$$\chi = F - E + V \quad (3.2)$$

the *Euler characteristic*. For all convex polyhedra $\chi = 2$.

The Euler characteristic can be generalised to parameterise general surfaces via the process of *cellulation* (or *triangulation*). The surface of any

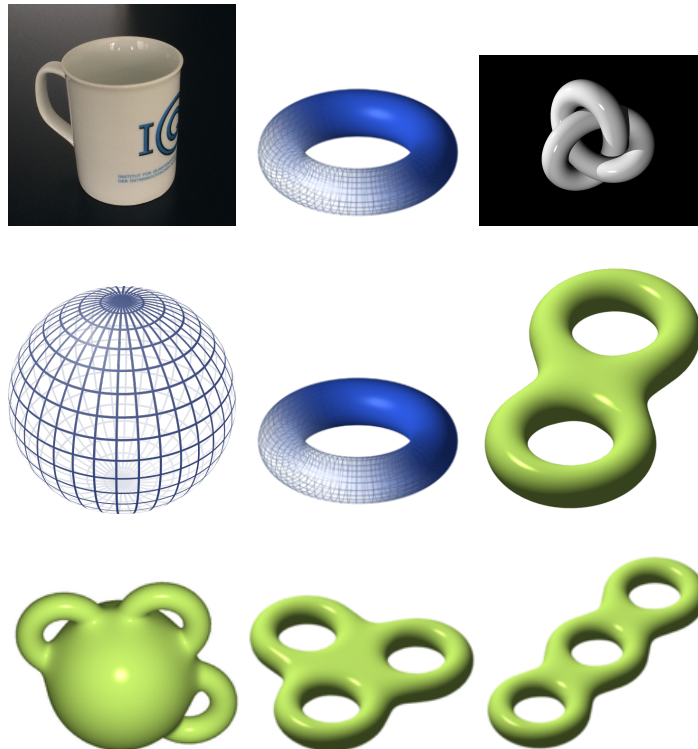


Figure 3.2: (The surface of) a coffee mug is homeomorphic to a torus, and also to a knotted torus (we can cut-and-reglue to unknot). A sphere, a torus and a double torus are *not* equivalent under homeomorphism. A sphere with two “handles” and the two triple tori illustrated are all homeomorphic. All images (except for photograph) from <http://wikipedia.com>.

3.1. Topological equivalence and topological invariants






Name	Image	Vertices V	Edges E	Faces F	Euler characteristic: $V - E + F$
Tetrahedron		4	6	4	2
Hexahedron or cube		8	12	6	2
Octahedron		6	12	8	2
Dodecahedron		20	30	12	2
Icosahedron		12	30	20	2

Figure 3.3: The Euler characteristic for all convex polyhedra is 2. Table reproduced from http://en.wikipedia.org/wiki/Euler_characteristic.

convex polyhedron is homeomorphic to the sphere (imagine “inflating” the polyhedron so that its surface becomes spherical). Left in place, the edges, vertices, and faces will now be inscribed on the surface of the sphere.

In general, the division of a surface up into a set of faces (also known as *cells*, also known as *plaquettes*) is known as *cellulation*. *Triangulation* is a special case, where a 2-d surface is divided into triangular faces. (Triangulation can also be performed on higher (n) dimensional topological spaces, where the space is divided into n -simplices).

One can prove (left as an exercise) that the Euler characteristic of *any* cellulation of the sphere is 2, and that more generally, for any surface, χ is independent of the cellulation chosen and is a topological invariant.

3.1.4 Some terminology for surfaces

Here I have paraphrased standard (and more precise) definitions to avoid topological concepts we have not introduced. These suffice for our purposes. Standard definitions can be found in most textbooks on topology.

- **Cellulation:** The division of a surface into polygonal cells (which need not be triangles) such that all cells meet edge-to-edge and corner-to-corner. There must be a finite number of triangles in any finite region of the surface.
- **Triangulation:** A cellulation where every cell is a triangle. (This definition generalises to higher dimensional spaces, which are divided into simplices – a simplex is a generalisation of the triangle to dimensions other than 2).
- **Triangulatable:** A surface is triangulatable if it can be triangulated. Non-triangulatable surfaces are exotic and we shall not consider them here.
- **Compact:** A triangulatable surface is compact if it can be triangulated with a finite number of finite cells. For example, the sphere and torus is compact, whereas the plane is not.
- **Connected:** We say a surface is connected if there is a continuous path between any two points on the surface. E.g. A sphere is connected, but a set of two spheres is not connected (since there is no continuous path from a point on one sphere to a point on the other sphere).
- **Orientable:** A triangulatable surface is orientable if we can assign an orientation to every cell in the triangulations, such that the orientations of edges of adjacent cells agree. E.g. A sphere is orientable, a mobius strip is not. We do not need to consider orientations in the study of qubit topological codes (but they are important in the definition of topological codes for qudits with $d > 2$.)
- **Closed:** A surface is closed if it has no *boundary*. E.g. A sphere and a torus are closed. A finite planar surface or a surface with holes is not closed.

In this course, we shall only consider surfaces which are orientable (and hence triangulatable), compact and connected, and will consider surfaces which are closed and also those with boundary.

3.1.5 Torus, n -torus and Genus

Let us now consider the Euler characteristic for the torus. We already computed the number of faces (L^2), edges ($2L^2$) and vertices (L^2) for a torus cellulated via a $L \times L$ square lattice. Hence the Euler characteristic for a torus is $L^2 - 2L^2 + L^2 = 0$.

A torus is a compact, closed orientable surface, similar to a sphere, but with a different Euler characteristic. Since the Euler characteristic is invariant under homeomorphism, this proves that sphere and torus are topologically inequivalent.

A torus is homeomorphic, however, to a sphere with a “handle”. Consider now a triple torus (bottom row of figure 3.2). The triple torus has Euler characteristic $\chi = -4$. It is not hard to see that it is homeomorphic to a sphere with three handles, the handles corresponding to the “holes” in the double torus. A word of caution regarding terminology. In topology, the word hole is always reserved for a cut in a surface which creates a boundary. To avoid confusion, do not call the hole-in-the-doughnut type of “hole” in the middle of a torus a hole, but rather a “handle” too.

The *genus* is the formal term we use to count the number of handles in a surface. The genus of a sphere is 0, the genus of a torus is 1, and the genus of a sphere with n handles is n . The genus is formally defined as follows:

- **Genus:** The genus of a connected surface is the number of closed cuts (here “closed”, means along a path which starts and ends at the same point) you can make on the surface before it is no longer connected.

In practise, however, you can think of genus as just another way of saying “number of handles”. One can prove that the Euler characteristic of any closed, compact, orientable surface of genus g is $2 - 2g$.

It was proved by Klein in 1882¹ that all closed compact surfaces with genus g are homeomorphic to a sphere with g handles. Thus the genus, and the Euler characteristic, fully characterise the topological equivalence for this subclass of surfaces.

¹F. Klein, *Über Riemann's theorie der algebraischen Funktionen und ihrer Integrale*, Teubner Verlag, Leipzig, 1882.

3.1.6 Euler characteristic and higher genus toric codes

We can make an immediate application of the Euler characteristic to toric codes. Recall that number of independent stabilizer generators for the toric code was equal to $(F - 1) + (V - 1)$ and the number of physical qubits is E . Hence the number of encoded qubits is:

$$k = n - m = E - ((F - 1) + (V - 1)) = 2 - \chi \quad (3.3)$$

We can immediately generalise this to the genus g g -torus. We can cellulate a general torus with a square lattice (exercise - try to construct such a cellulation) and define stabilizer generators in the same way as for the toric code.

The Euler characteristic for such a code is $2 - 2g$, and thus the number of encoded qubits is simply $k = 2g$. Thus adding handles into the surface allows us to increase the number of qubits in the code - each extra handle gives us 2 extra encoded qubits.

We will show precisely, that such a construction *does* define a valid topological code encoding $2 - 2g$ qubits, (and, later, planar codes with punctures and boundaries) in later sections, after we have introduced homology.

3.1.7 Surfaces with a boundary

Consider a sphere with a circular hole cut into it. We can flatten out the sphere continuously to a circular disk, so disk and sphere-with-hole are homeomorphic. These are examples of surfaces with a *boundary*.

Any polygon is equivalent to a disk up to homeomorphism, so we can use the polygon to cellulate the disk and compute its Euler characteristic. Recall that an n -sided polygon has n vertices and 1 face, so the Euler characteristic $\chi = 1$.

Imagine we cellulate our disk with many cells. We can convince ourselves that every extra hole reduces the Euler characteristic by 1. Removing a single face from the disk is equivalent to punching a new hole in the surface. Every face removed adds one extra boundary and an extra hole in the surface.

In general for a compact oriented surface with genus g and b boundaries (where the boundaries can be due to the “edge” of the surface and to holes),

$$\chi = 2 - 2g - b \quad (3.4)$$

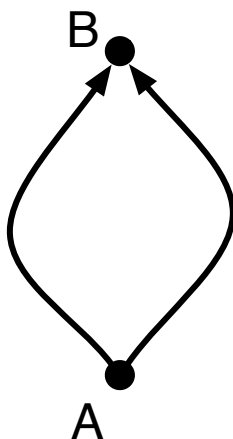


Figure 3.4: Two points A and B lie on a sphere. The two indicated paths are homotopic, since there is a continuous function on the surface (continuous deformation) which maps one to the other.

Surfaces with boundary will be important in our studies of the planar code, and we shall return to them, and study their topological properties in more depth in part 5 of this course.

3.1.8 Homotopy

Homotopy is the second topological term beginning with ‘h’ that we encounter. Homotopy is not, in itself, an important concept for topological codes and we shall not cover it in depth. However, due to its importance for defining topological invariants, its relationship with *homology* (which is of central importance for topological codes), and since the similarity between these two terms creates the danger of confusing them, we shall briefly describe it.

Consider two points on a surface A and B and two paths from A to B . We say that there is a homotopy the two path, or that the paths are *homotopic*, when there exists a continuous deformation which transforms one path into the other. See figure 3.4 for an example.

Holes and handles can form barriers to homotopy. In figure 3.5 we see three paths which are not homotopic.

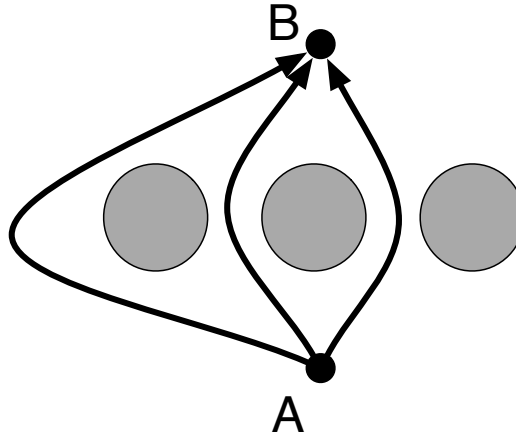


Figure 3.5: Two points A and B lie on a sphere with three punctured holes. The three indicated paths are not homotopic, since there is not continuous function on the surface (no continuous deformation) which maps one to the other. The holes prevent this.

3.1.9 Fundamental group

Homotopy is used to define an important topological invariant, the *fundamental group*. The fundamental group of a connected surface is defined as follows.

- Choose a point A on the surface and consider all directed paths on the surface which both start and end at A .
- Group the paths into equivalence classes under homotopy, i.e. two paths belong to the same class if they are homotopic.
- The *fundamental group* of the surface is the group isomorphic to this set of equivalence classes.

Although point A appears in its definition, one can show that the fundamental group is independent of the point A chosen. How do these paths represent a group? We define the composition of two paths as taking one path after the other. We can do this, since the paths we consider here all start and end at A . We define the inverse of the path to be the path with its

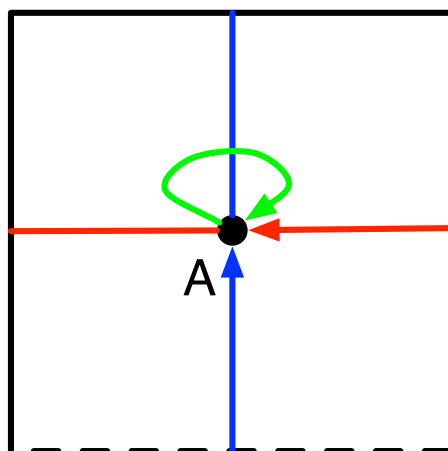


Figure 3.6: Three non-homotopically equivalent paths on a torus from point A to point A (left hand side is identified with right hand side and top with bottom). The green path is homotopically equivalent to the “identity” path – the path which never leaves point A.

direction reversed. The composition of paths is clearly associative. Thus the paths satisfy all the axioms for a group. The point-loop and any loop which can be deformed to a point are equivalent, and these will represent the identity operator in this group. All loops which cannot be deformed to a point represent non-trivial elements of the group.

An immediate consequence is that the fundamental group of a sphere is trivial, since all loops can be deformed to a point.

The fundamental loop of a torus is non-trivial. Consider a simple closed torus with 1 handle (see figure 3.6). We can classify three separate types of loops starting and ending at a particular point A. Loops which do not pass around the handle, and loops which circuit the handle in two inequivalent ways. Loops which do not pass around the handle can be deformed to a point - they thus represent the identity operator. However the loops which pass around the handle cannot be deformed to a point. They form two inequivalent classes of loop.

Let us first consider just one of these equivalence classes of loops. We can compose paths and invert them, so the only distinguishing feature (up to homotopy) of these paths is the number of times they pass through A

and their direction. This group structure is isomorphic to the integers under addition, \mathbb{Z} .

Now consider the other equivalence class of loops, those passing the other way around the handle of the torus. On their own, these loops are isomorphic to \mathbb{Z} as well. However, we can create paths which combine the two types of loop, first going one way round the torus handle, then going the other way. The fundamental group of the torus is Abelian, since one can continuously deform a path which first goes one way round the handle, and then the other way round, into a path which reverses this order. The video at the following URL: <https://www.youtube.com/watch?v=nLcr-DWVEto> gives a very clear illustration of this. Putting this together, we find that the fundamental group of the torus is isomorphic to the Abelian group $\mathbb{Z} \times \mathbb{Z}$, i.e. regular addition over pairs of integers.

The fundamental group is useful for characterising topological spaces since it (loosely speaking) “sees” their holes and handles. However, for surfaces more complicated than the torus, constructing the fundamental group can be quite involved. For example, the fundamental group for an n torus for $n > 1$ is not Abelian.

The fundamental group is therefore something of a *false friend* for our studies of topological codes - while the structure of the fundamental group of the torus is reminiscent of the structure of the logical operators of the toric code, this is not true in general. We shall *not* be using the fundamental group in our studies of topological codes, but it does provide a simple example of how a group structure can emerge from topology.

Homology, which we study next, illustrates a further marriage of topology and group theory. Homology is the branch of topology most relevant for topological codes. Indeed, we will see that topological codes can be formulated and understood almost entirely in homological terms.

3.2 \mathbb{Z}_2 Homology

Homology is the mathematical field which abstracts and generalises the notion of *boundary*. From this simple idea, a versatile and powerful set of mathematical tools and structures emerge. There are a number of ways to introduce homology. The approach taken here is slightly unconventional, and focuses on the concrete aspects of homology relevant to qubit topological codes. In particular, we will limit our study to homology on the group \mathbb{Z}_2 ,

the simplest form of homology, and will largely study square lattice cellulations. This is the most relevant form of homology for the study of topological qubit codes. However, everything stated here holds for more general cellulations. We remark on how homology can be constructed from other groups than \mathbb{Z}_2 at the end of this section.

We shall start by recalling the square lattice with periodic boundary conditions, from which we defined the toric code. The lattice is an example of a *cellulation* of the torus. It consists of 0-dimensional objects, the vertices, 1-dimensional objects, the edges and 2-dimensional objects, the plaquettes. We call an n -dimensional object in a cellulation an n -cell - i.e. the vertices 0-cells, the edges 1-cells and the plaquettes 2-cells.

In cellular homology, one associates an element from a *chosen group*, here \mathbb{Z}_2 , with each n -cell, e.g. each vertex, each edge and each plaquette forming an object called an n -chain.

The group \mathbb{Z}_2 is the simplest non-trivial group. It has two elements, and is isomorphic to the set of integers $\{0,1\}$ with group composition given by addition modulo 2. The identity element is identified with 0. \mathbb{Z}_2 is an Abelian group, and all of its elements are self-inverse. These special features make \mathbb{Z}_2 homology simpler, in some aspects, than homologies constructed from other groups. Nevertheless, \mathbb{Z}_2 homology illustrates all the key features of homology in general, and is \mathbb{Z}_2 homology which can be used to describe qubit topological codes.

3.2.1 1-Chains

Chains are the basic objects in homology. Let us introduce them with some examples. Consider the cellulation of the torus we studied before.

Let's assign to each edge on the lattice an integer 0 or 1. We call this collection of integers a 1-chain (see figure 3.7) since edges are 1-dimensional objects.

Recall that the set 0, 1 together with addition mod 2 is isomorphic to the cyclic group \mathbb{Z}_2). Our 1-chain is therefore, more abstractly, an assignment of elements from \mathbb{Z}_2 to every 1-dimensional object on our lattice.

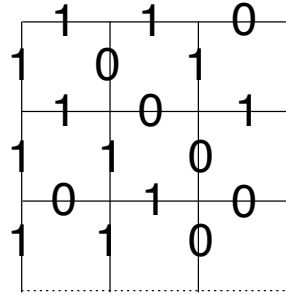


Figure 3.7: In \mathbb{Z}_2 homology on a 1-chain is an assignment of an element of \mathbb{Z}_2 (or equivalently a 0 or 1) to every edge (1-dimensional object).

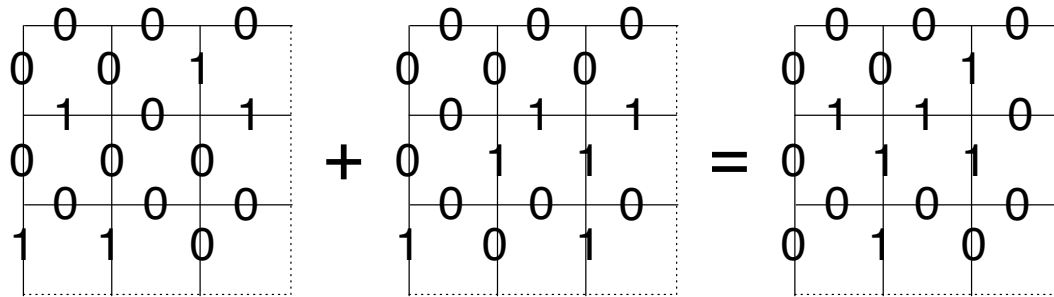


Figure 3.8: Group composition of 1-chains is defined by edge-wise binary addition.

We can add two 1-chains together via bit-wise modulo 2 (see figure 3.8) addition to produce another 1-chain. The 1-chains form a group, via the following observations:

- Edge-wise (bit-wise) addition, modulo 2, represents group composition.
- This composition is associative, due to the associativity of addition.
- The “all-zeros” one-chain – the 1-chain with 0 assigned to every edge – represents the identity operator.

- An inverse exists for every 1-chain (by inverting the group element for every edge). In this special case of \mathbb{Z}_2 every 1-chain is self-inverse.

1-chains therefore satisfy all the properties of a group. Furthermore, the group is Abelian (inherited from the group \mathbb{Z}_2). Let E be the number of edges, the order of this group is simply the number of E -bit strings 2^E . Furthermore, we can identify E independent generators of the group. The most natural set of generators to choose is to associate a generator with each edge, defining the 1-chain where we place a 1 on the specified edge, and a 0 on every other edge. Creating a similar element for every edge, we recover an independent generating set for the full 1-chain group. We call such a generator an edge generator.

There is one further nice feature of \mathbb{Z}_2 homology. Since each edge takes values 0 and 1 only, each 1-chain can be identified with a subset of the edges (see, e.g. figure 3.9), i.e. the 1-values represent the edges which are “there”. One important warning: Chains, in this context of homology, have no connection with the usual meaning of the English word “chain”. In particular 1-chains are not limited to “chain-like” sets of connected edges - any subset of edges, connected or not, represents a valid 1-chain. We shall call this view of chains the “subset picture”.

I should make an important comment regarding terminology. Since the group structure is so important in homology, we shall often talk about combining 1-chains (or other objects) in group theoretical terms, and speak about “group products” and “group composition”. In the representation we are using, however, the group product is represented by bit-wise addition modulo 2. Therefore, chain “addition” and (group) multiplication mean the same thing! As an example, the group of 1-chains can be thought of as a vector space, and the set of independent generators could be equally described as a basis for that space. In general, the language we choose will be the one most suitable for explaining the phenomenon in hand, so while we will typically speak in group theoretical terms, we will use the language of vector spaces when it is more natural (for example, when we speak of a “linear functional” rather than a “group homomorphism to \mathbb{Z}_2 ” below).

3.2.2 0-chains, 2-chains and n -chains

0-chains and 2-chains are constructed in a similar manner to 1-chains. To construct a 0-chain, we assign an element of \mathbb{Z}_2 to every vertex (0-dimensional

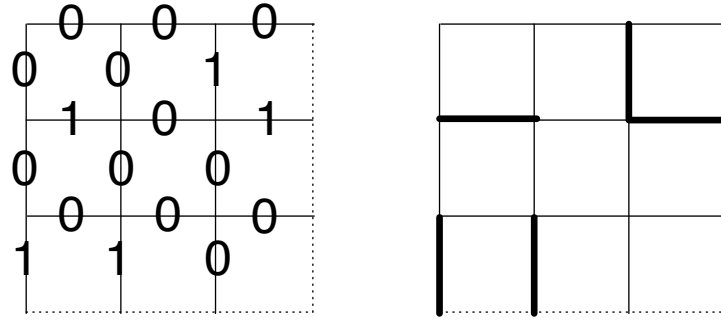


Figure 3.9: One can identify every 1-chain in \mathbb{Z}_2 homology with a subset of edges.

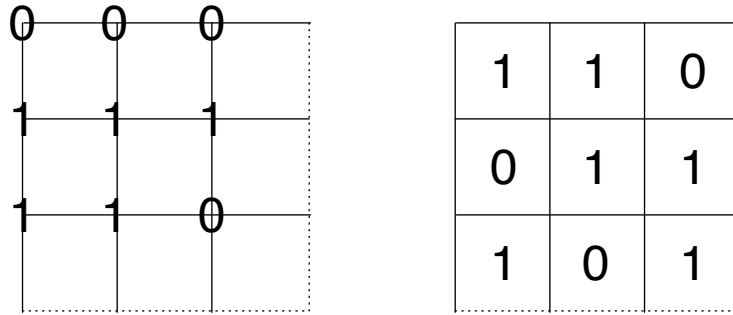


Figure 3.10: An example of a 0-chain and a 2-chain.

object). To construct a 2-chain we assign 0 or 1 to every plaquette (1-dimensional object).

The sets of 0-chains and 2-chains form groups and we define independent sets of generators associated with single vertices and single plaquettes analogously to the way single edges define generators for the 1-chain.

Homology is not restricted to 2-dimensional surfaces. We can cellulate an n dimensional space, into n -dimensional cells, whose faces will be $n - 1$ dimensional, which in turn will have $n - 2$ -dimensional faces / edges and so on. One then can construct n -chains, $n - 1$ chains, etc. Here we will focus on 2-dimensional surfaces for all our examples, but everything which follows generalises naturally to the n -dimensional case.

$$\partial_2 \left(\begin{array}{|c|c|c|} \hline & & \\ \hline & 1 & \\ \hline & & \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline & & \\ \hline & 1 & 1 \\ \hline & 1 & \\ \hline \end{array}$$

Figure 3.11: The boundary map acting on a plaquette generator 2-chain. In this and the following figures, 0 valued plaquettes and edges have been left unlabelled for clarity.

A note on notation. I will label the group of n -chains C_n , and use lower case roman letters to label chains, e.g. 1-chain $c \in C_1$.

3.2.3 The boundary map

The *boundary map* plays a central role in homology theory. Consider a single plaquette on the square lattice (a 2-dimensional object). We'd say that its *boundary* comprises the 4 edges – i.e. 1-dimensional objects – which enclose it. The boundary map formalises and generalises this.

In general, the n -boundary map ∂_n is a group-structure preserving map (a group homomorphism) from the set of n -chains c to the $(n - 1)$ -chains d

$$\partial_n(c) = d \quad (3.5)$$

Before studying its general properties, we will study some concrete examples.

3.2.4 The 2-boundary map

Consider a 2-chain plaquette generator, i.e. the 2-chain consisting of a single plaquette alone, in the subset picture. The boundary map, maps this 2-chain to the 1-chain representing its boundary - i.e. the 1-chain with 1's around the edges of that plaquette and 0's everywhere else (see figure 3.11).

A boundary map is a group homomorphism – a group composition preserving map (not to be confused with a homeomorphism, a homotopy or homology!). Since the group composition in \mathbb{Z}_2 is addition modulo 2, another

$$\partial_2 \left(\begin{array}{|c|c|c|} \hline & 1 & \\ \hline & 1 & \\ \hline 1 & & \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline 1 & & 1 \\ \hline & 1 & 1 \\ \hline & 1 & 1 \\ \hline 1 & 1 & \\ \hline \end{array}$$

Figure 3.12: An example of the boundary map acting on a 2-chain. In \mathbb{Z}_2 homology, the boundary maps have an intuitive meaning, the 1-chain represents the joint “boundary” of the subset of plaquettes in the 2-chain.

way of saying this is that ∂_n is a *linear map* modulo 2. More specifically, for the case of 2-chains, let c_j be the set of 2-chains generators for and let $c = \sum_j a_j c_j$ be a general 2-chain, where a_j are bit-valued coefficients. We see that the boundary of the 2-chain is the sum (mod 2) of the boundaries of the non-zero 2-cells of the 2-chain.

$$\partial_2(c) = \partial_2\left(\sum_j a_j c_j\right) = \sum_j a_j \partial_2 c_j. \quad (3.6)$$

With the definition of boundary map for plaquette generators defined as in figure 3.11, the boundary map for a general 2-chain is very intuitive. It maps the 2-chain to the 1-chain which “surrounds” the non-zero cells of the 2-chain. See for example figure 3.12. (You might notice that this is identical to the way multiplication for plaquette operators was represented – this is not a coincidence.)

3.2.5 The 1-boundary map

In homology, we define boundary maps for all n -chains. The 1-boundary map therefore maps 1-chains (subsets of edges) to 0-chains (subsets of vertices).

It’s definition is very natural. For a generator identified with an edge (a 1-chain with 1 on that edge and 0s on all other edges) the 1-boundary corresponds to the set of vertices adjacent to that edge. This is illustrated in figure 3.13.

$$\partial_2 \left(\begin{array}{|c|c|c|} \hline & & \\ \hline & 1 & \\ \hline & & \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline & & \\ \hline & 1 & \\ \hline & 1 & \\ \hline \end{array}$$

Figure 3.13: The 1-boundary map acting on an edge generator 1-chain.

$$\partial_1 \left(\begin{array}{|c|c|c|} \hline & & 1 \\ \hline & 1 & \\ \hline & & 1 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline 1 & & 1 \\ \hline 1 & & \\ \hline & & 1 \\ \hline \end{array}$$

 Figure 3.14: An example of the boundary map acting on a 1-chain. In \mathbb{Z}_2 homology, the boundary maps have an intuitive meaning, the 0-chain represents the ends of each disjoint “string” in the 1-chain.

The boundary map for general 1-chains, and, indeed for n -chains of any dimension, by group homomorphism, so for general n -chain $c = \sum_j a_j c_j$, where a_j are bit-valued coefficients,

$$\partial_n(c) = \partial_n\left(\sum_j a_j c_j\right) = \sum_j a_j \partial_n c_j. \quad (3.7)$$

The interpretation of the 1-chain boundary map is again intuitive. Given a 1-chain consisting of connected edges (e.g. a “string”), the boundary consists of the vertices at the two ends of those strings. For one-chains consisting of multiple disjoint “strings” the boundary consists of 2 vertices at the ends of each disjoint string.

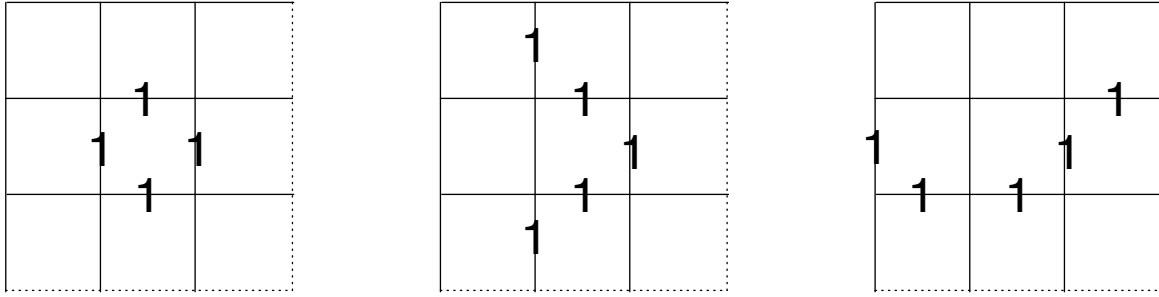


Figure 3.15: Three examples of 1-cycles on the torus. Only the first example is a 1-boundary.

3.2.6 The 0-boundary map

What is the boundary map for 0-cells? We do not have any -1 dimensional objects, it is useful to define a ∂_0 map in any case. For every group of n -chains we identify the identity operator with the “null chain”, the chain with 0s on all n -cells. We define the group of -1 chains as containing the trivial group containing simply this identity element. We shall refer to the identity element in the n -chain group as the *null n -chain*, and write it as 0_n . The 0-boundary map is then defined as mapping every 0-chain to the null (-1) -chain. E.g. for any 0-chain c , $\partial_0(c) = 0_{n-1}$.

A remark on notation. When it is obvious from context what the value of n is, it is common to omit the n in ∂_n and write simply ∂ .

3.2.7 Cycles

A n -cycle is an n -chain c satisfying:

$$\partial_n c = 0_{n-1} \quad (3.8)$$

In other words, a cycle is a chain with null boundary or, equivalently “without a boundary”. Cycles are a central concept in homology. Note that cycles form a group, since, given n -cycles c and d , then the group product $c \cdot d \equiv c + d$ is also a cycle.

$$\partial_n(c + d) = \partial_n(c) + \partial_n(d) = 0_{n-1} + 0_{n-1} = 0_{n-1} \quad (3.9)$$

We label the group of n -cycles Z_n (from the German).

Some examples of 1-cycles on a torus are illustrated in figure 3.15. On the torus there are just two 2-cycles, the null 2-chain and the “all-ones” 2-chain. From the definition of the ∂_0 map, every 0-chain is a 0-cycle.

3.2.8 n -boundary chains

Any n -chain which is the boundary of an $(n+1)$ -chain we call an n -boundary chain, or simply n -boundary. The set of n -boundaries forms a group (by a similar reasoning to Z_n) which we label B_n .

A very important observation is the following:

- Every n -boundary is a n -cycle, but not every n -cycle is a boundary.

The property that every n -boundary is a n -cycle is one of the defining features of homology. This has an immediate consequence for the boundary maps, applying a boundary map twice in succession, always produces a null chain, symbolically

$$\partial_{n-1}\partial_n c = 0 \quad (3.10)$$

where c is any n -chain and 0 is the null $(n-2)$ -chain. This equation can be used to define boundary maps in a more abstract setting. This equation is sometimes called the *fundamental lemma of homology*.

You can verify that the boundary maps that we defined earlier in this chapter are consistent with this rule, e.g. $\partial_1\partial_0 c = 0$, which is guaranteed since the 0th boundary map maps every 0-chain to the null chain.

We have defined, for each dimension n , three different groups of chains, C_n (the full group of n -chains) which contains Z_n (the subgroup of n -cycles) which contains B_n (the subgroup of n -boundaries). Homology studies the properties and interplay of these groups to characterise the topology of the surfaces or spaces.

3.2.9 Homological Equivalence

The group of n -boundaries allow us to define an importance equivalence relation, *homological equivalence*. We say two n -chains c and d are homologically equivalent if they are equal up to composition with an n -boundary b ,

$$c = d + b \quad (3.11)$$

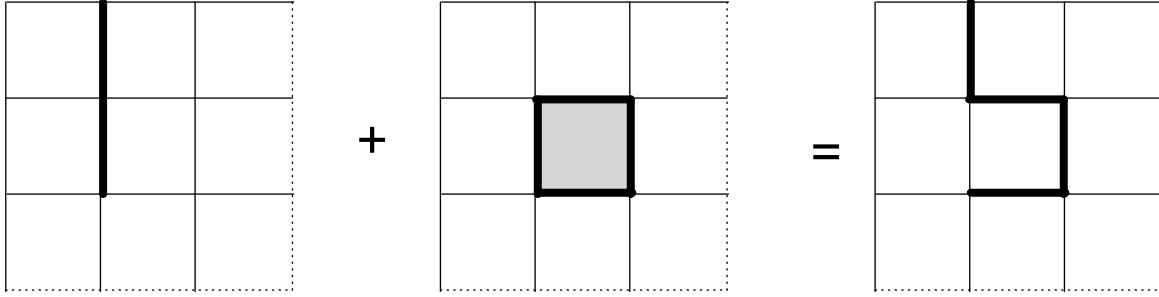


Figure 3.16: The 1-chain on the left is homologically equivalent to the 1-chain on the right. The 1-chain on the right is equal to the 1-chain on the left plus the boundary of the shaded plaquette, hence they are homologically equivalent. Here, and in the figures that follow, we indicate 0-chains, 1-chains, and 2-chains by shaded vertices, edges, and plaquettes respectively.

where $b \in B_n$. An example of two homologically equivalent 1-chains is given in figure 3.16.

3.2.10 Homology groups

If c and d are homologically equivalent they share the same boundary. However, not all n -chains with the same boundary are homologically equivalent. Homological equivalence partitions the subgroup of chains with a given boundary into equivalence classes. An example is given in figure 3.17

Such partitioning is particularly important for *cycles*. Cycles are chains with (the same) null boundary, and therefore can be partitioned under homological equivalence.

Recall that the n -boundaries B_n , are a subgroup of the n -cycles Z_n which form a subgroup of the n -chains C_n . Partitioning Z_n into equivalence classes under composition with elements of B_n therefore defines a *quotient group*. This quotient group is called the n th homology group H_n .

Written as a quotient group, H_n is defined:

$$H_n = \frac{Z_n}{B_n} \quad (3.12)$$

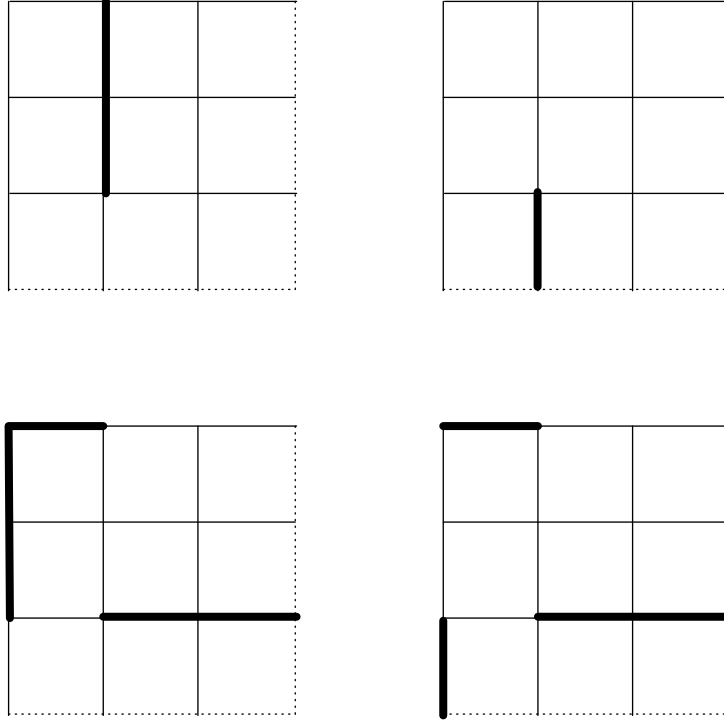


Figure 3.17: The four 1-chains depicted have the same boundary but are not homologically equivalent. They define 4 homological equivalence classes.

The properties of the homology groups depend upon the topology of the surfaces / spaces on which they are defined. On the torus, the first homology group has 4 elements. The group is Abelian, and every element is self-inverse (both properties inherited from \mathbb{Z}_2). It is therefore isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$. In figure 3.18, 1-chains representing each of the 4 equivalence classes are depicted. One equivalence class is equal to B_1 itself (we call this the homologically trivial element of H_1) and represents the identity operator in H_1 .

The 0th and 2nd homology groups on the torus can be easily obtained. In 0-dimensions, $Z_0 = C_0$, as every 0-chain is a cycle. The boundary of every 1-chain is a 0-chain with an even number of non-zero vertices. Moreover, every 0-chain with an even number of non-zero vertices is a boundary. There are thus two homological equivalence classes of 0-cycles, those with an even

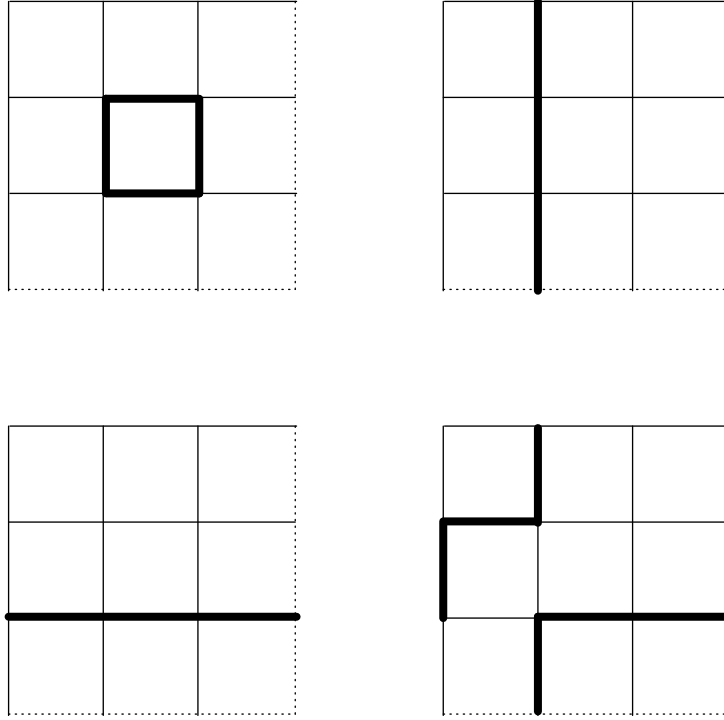


Figure 3.18: Representatives of the four homological equivalence classes of 1-cycles on the torus. These equivalence classes define the elements of the 1st homology group for the torus. The group is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$.

number of non-zero vertices and those with an odd number. The 0th homology group of the torus is therefore isomorphic to \mathbb{Z}_2 .

To find the 2nd homology group, we consider 2-cycles and 2-boundaries. The torus is a 2-dimensional surface, so there are no 3-chains. We define the group B_2 as the trivial one-element group, and thus $H_2 = \mathbb{Z}_2$. There are only two elements in \mathbb{Z}_2 , the null chain and the “all-ones” 2-chain, the product of all plaquettes with covers the entire surface. Thus the second homology group of the torus is isomorphic to \mathbb{Z}_2 .

Homology groups are a function of the topology of the space, and are invariant under changes in cellulation, and in homeomorphisms. They represent an important family of topological invariants.

3.2.11 Betti numbers and the Euler characteristic

There is a direct relationship between homology classes and the Euler characteristic for a topological space. The *rank* of an Abelian group is defined as the number of independent generators of the group. In homology, the n th Betti number β_n is the rank of the n th homology group.

$$\beta_n = \text{rank}H_n = \text{rank}Z_n - \text{rank}B_n \quad (3.13)$$

For the torus, the Betti numbers are $\beta_0 = 1, \beta_1 = 2, \beta_2 = 1$. Remarkably, the Betti numbers are closely related to the Euler characteristic of a surface. Recall that the Euler characteristic is defined as

$$\chi = F - E + V \quad (3.14)$$

where F is the number of faces, E , the number of edges and V the number of vertices. We can now rewrite this in homology terms. The number of faces/edges/vertices is equal to the rank of the 2/1/0-chain group, hence,

$$\chi = \text{rank}C_2 - \text{rank}C_1 + \text{rank}C_0 \quad (3.15)$$

We can rewrite this, using an identity from linear algebra. Since the boundary map ∂_n is a linear map, it satisfies the “rank-nullity theorem” of linear maps. For any linear map M , the dimension of the domain of the map is equal to the sum of the dimensions of the kernel and image of the map. The domain of ∂_n is simply C_n , while the kernel of the ∂_n (the chains it maps to 0) are the cycles Z_n . Finally, the image of ∂_n is B_{n-1} , the set of $(n-1)$ -boundaries.

This implies that

$$\text{rank}C_n = \text{rank}Z_n + \text{rank}B_{n-1} \quad (3.16)$$

Thus,

$$\begin{aligned} \chi &= \text{rank}C_2 - \text{rank}C_1 + \text{rank}C_0 \\ &= (\text{rank}Z_2 + \text{rank}B_1) - (\text{rank}Z_1 + \text{rank}B_0) + (\text{rank}Z_0 + \text{rank}B_{-1}) \\ &= (\text{rank}Z_2 - \text{rank}B_2) - (\text{rank}Z_1 - \text{rank}B_1) + (\text{rank}Z_0 - \text{rank}B_0) \\ &= \beta_2 - \beta_1 + \beta_0 \end{aligned} \quad (3.17)$$

where we used the fact that the ranks of B_2 and B_{-1} are zero, since there are no 3-chains or (-1) -chains.

The Euler characteristic is thus directly related to the ranks of the homology groups, equal to the alternating sum of the Betti numbers. For example, for the torus $\chi = 1 - 2 + 1 = 0$ as required. For higher dimensional topological spaces, the Euler characteristic generalises to $\chi = \sum_k (-1)^k \beta_k$.

3.3 Cohomology

We have now encountered the main ingredients of homology, and they should look familiar. Many of these elements have a high degree of similarity with the mathematical structures encountered in the study of the toric code. We shall make these correspondences more precise in the next section, when we revisit the toric code with this homology machinery.

However, one key aspect of our studies of the toric code has not yet arisen – the dual lattice. As we shall now see, this duality notion is captured and generalised in *cohomology*.

3.3.1 Co-chains

In cohomology, we develop duals for the key aspects of homology, including chains, boundaries and cycles.

We first introduce co-chains. Co-chains are defined as dual to chains. The type of duality we employ is similar of the way bras $\langle \psi |$ are dual to kets $|\psi\rangle$ in quantum mechanics. Recall that the set of bras is formally defined as the set of *linear functionals* $f_\phi()$ on the vector space of the kets, i.e. linear maps from the kets to scalars.

$$f_\phi(|\psi\rangle) = \langle \phi | \psi \rangle \in \mathbb{C} \quad (3.18)$$

We define co-chains in an analogous fashion. The set of n -co-chains corresponds to the set of linear functionals which map each n -chain to the group \mathbb{Z}_2 (or group theoretically as the set of group homomorphisms which map C_n to \mathbb{Z}_2).

This formal definition is hard to parse, so let's unpack it. The linear functionals must assign a value of 0 or 1 to every n -chain, and do so in a way that respects the group composition (cell-wise modular addition) of chains.

Similar to bras, the co-chains can be lifted from abstract functionals to concrete objects, via an “inner product” construction². In quantum mechanics, we associate every linear functional with a element of a dual vector space via the “Dirac product” $\langle \phi | \psi \rangle$. Every functional is associated with a vector $\langle \phi |$ in the dual space.

In cohomology, we can perform the same trick. A n -cochain \tilde{p} (we will always write cochains with roman letters with a tilde) is an element of the n -cochain group such that (via an inner product construction which we will define)

$$\langle \tilde{p}, c \rangle \in \mathbb{Z}_2 \quad (3.19)$$

Every linear functional on c can be uniquely written as an inner product with some member of the cochain group, and we therefore identify cochains and linear functionals (the way we identify linear functionals and bra vectors). This may have seemed very abstract, so far, but the inner product construction allows us to identify cochains, and their corresponding inner product with chains, in a simple and concrete way via the dual lattice we introduced in part 2.

3.3.2 0-cochains

We will start with the 0-cochains. Recall that 0-chains correspond to subsets of vertices – or equivalently to an assignment of 0 or 1 to every vertex. Recall that every vertex in the primal lattice corresponds to a plaquette in the dual lattice.

We define the 0-cochains as an assignment of 0 or 1 to every plaquette in the dual lattice, or equivalently as a specification of a subset of these plaquettes. The inner product between 0-cochain \tilde{p} and 0-chain c

$$\langle \tilde{p}, c \rangle \in \mathbb{Z}_2 \quad (3.20)$$

is defined as the sum modulo 2 of vertices of c which lie inside plaquettes of \tilde{p} . This is illustrated in figure 3.19.

²Strictly speaking, this is not an inner product, as an inner product would be a product between two elements of the same space, whereas the products we consider are between a elements of a space and its dual, but we follow common quantum mechanics practise and homology textbooks in abusing that terminology.

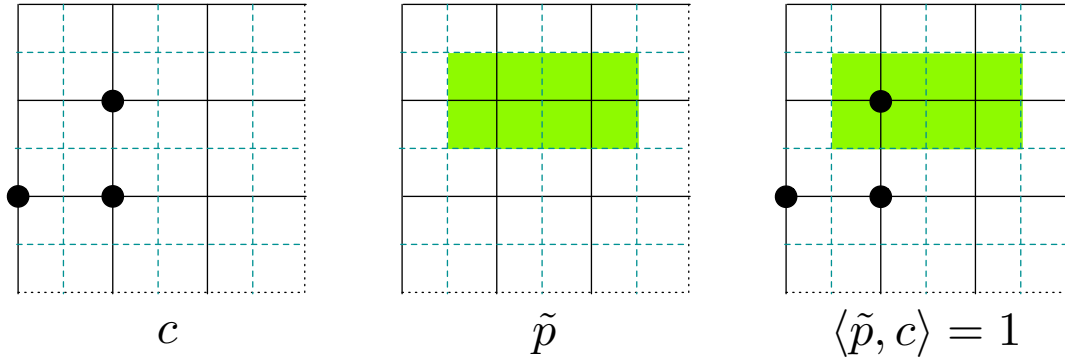


Figure 3.19: An example of a 0-chain c and a 0-cochain \tilde{p} . Their inner product $\langle \tilde{p}, c \rangle$ represents the sum modulo 2 of the number of vertices in c (in the subset picture) which lie inside dual-lattice plaquettes of \tilde{p} .

In plainer terms, we count the number of 1-assigned vertices in c which lie inside the 1-assigned plaquettes in \tilde{p} and return 0 if this count is even and 1 if this count is odd. It is left as an exercise to prove that this construction does indeed generate all possible linear functionals on the 0-chains, and that the 0-cochains form a group. A convenient set of independent generators for the group is the set of individual plaquettes on the dual lattice.

3.3.3 1-cochains

Edges on the primal lattice correspond with edges on the dual lattice. Correspondingly we associate the set of 1-cochains with subsets of *edges* on the *dual lattice*.

The inner product between a 1-cochain \tilde{p} and a 1-chain c is defined as the number of times, modulo 2, that the edges in \tilde{p} intersect the edges of c . This is illustrated in figure 3.20.

Again, the proof that this realises all linear functionals on the 1-chains is left to the reader. A convenient set of generators of the 1-cochain group are the 1-cochains corresponding to single edges on the dual lattice.

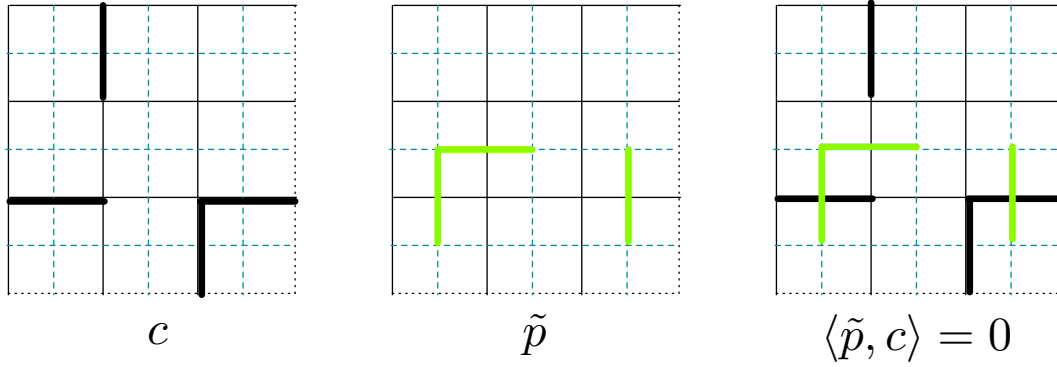


Figure 3.20: An example of a 1-chain c and a 1-cochain \tilde{p} . Their inner product $\langle \tilde{p}, c \rangle$ represents the sum modulo 2 of the number of edges in c (in the subset picture) which cross dual-lattice edges of \tilde{p} .

3.3.4 2-cochains

Plaquettes on the primal lattice correspond with vertices on the dual lattice. Correspondingly we associate the set of 2-cochains with subsets of *vertices* on the *dual lattice*.

The inner product between a 2-cochain \tilde{p} and a 2-chain c is defined as the number of vertices of \tilde{p} that lie inside the plaquettes of c . This is analogous (but interchanging primal and dual lattices) to the relationship between 0-chains and 0-cochains illustrated in figure 3.19.

The proof that this realises all linear functionals on the 1-chains is left to the reader. A convenient set of generators of the 2-cochain group are the 2-cochains corresponding to single vertices on the dual lattice.

We denote the n -cochain group C^n .

3.3.5 Coboundary

The sharper eyed reader will be wondering why we used the dual lattice at all in our previous section. For example, we could have defined these sets of linear functional working solely in the primal lattice. The choice of dual lattice will (I hope) become clearer (and necessary) when we introduce the concept of coboundary.

The n -coboundary map $\tilde{\partial}^n$ is dual to the n -boundary map. It maps n -cochains to $(n + 1)$ -cochains is defined by the following equation:

$$\langle \tilde{\partial}^n \tilde{p}, c \rangle = \langle \tilde{p}, \partial_{n+1} c \rangle \quad (3.21)$$

where \tilde{p} is a n -cochain, and c is an $(n + 1)$ -chain. You may note the similarity between this definition and the definition of Hermitian conjugate in finite quantum mechanics / linear algebra.

In other words, the coboundary of an n -cochain is the $(n + 1)$ -cochain whose inner product with $(n + 1)$ -chain c is equal to the inner product of \tilde{p} with the boundary of c .

The coboundary is a group homomorphism on the cochains and can therefore, similar to the boundary, be specified for a set of generators and extended to general cochains.

3.3.6 0-coboundary

Recall that 0-cochains are identified with plaquettes on the dual lattice. Consider a single plaquette. The edges which surround it are a subset of edges in the dual lattice - a 1-cochain. The coboundary of a single dual lattice plaquette is simply this, the 1-cochain corresponding to the edges surrounding it.

In general, then the 0-coboundary map for a 0-cochain consists of the edges on the dual lattice surrounding the subset of dual lattice plaquettes defining the 0-cochain.

To see that this definition of coboundary does satisfy equation (3.21), see figure 3.21. The inner product $\langle \tilde{p}, \partial_1 c \rangle$ counts the number of vertices (modulo 2) of the 0-chain $\partial_1 c$ which lie inside the dual lattice plaquettes of 0-cochain \tilde{p} . The inner product $\langle \tilde{\partial}^0 \tilde{p}, c \rangle$ counts (modulo 2) the number of times the 1-chain c crosses the coboundary of cochain \tilde{p} . These values must always coincide since if an odd number of boundary vertices of c lie in $\langle \tilde{p}$, an odd number of edges of c must intersect the coboundary of \tilde{p} . This identity is reminiscent of Gauss's law in electromagnetism.

3.3.7 1-coboundary

1-cochains are identified with edges on the dual lattice. We define the coboundary of a 1-cochains analogously to the boundary of a 1-chain, by setting the

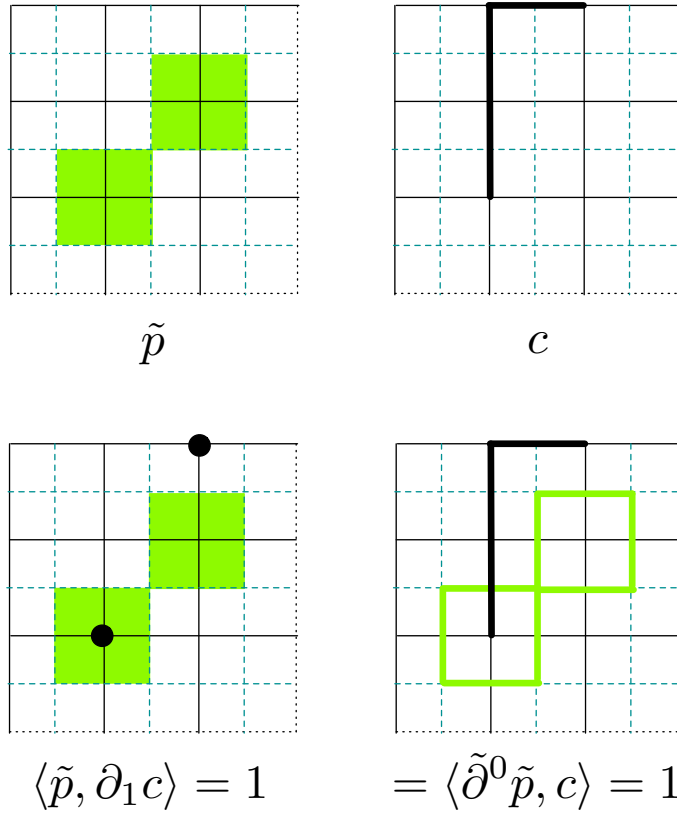


Figure 3.21: The thick green lines in the bottom right subfigure is an example of a 0-coboundary. On the top row, we see a 0-cochain \tilde{p} and a 1-chain c . Agreeing with the definition of co-boundary) we see that the inner product of \tilde{p} with the 0-boundary $\partial_1 c$ is equal to the inner product of 1-coboundary $\tilde{\partial}^0 \tilde{p}$ and c .

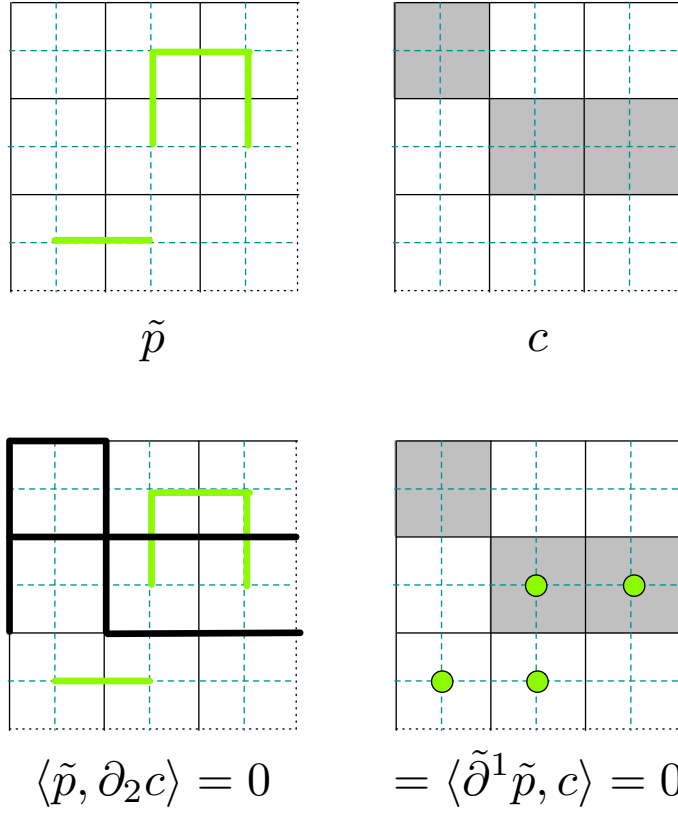


Figure 3.22: The green highlighted dual lattice vertices in the bottom right subfigure are an example of a 1-coboundary. On the top row, we see a 1-cochain \tilde{p} and a 2-chain c . (Agreeing with the definition of co-boundary) we see that the inner product of \tilde{p} with the 0-boundary $\partial_2 c$ is equal to the inner product of 1-coboundary $\tilde{\partial}^1 \tilde{p}$ and c .

coboundary of individual edges on the dual lattice as the (dual lattice) vertices adjacent to them.

In general then, the boundary of any 1-cochain is the set of 2-cochain vertices which lie at the ends of each connected set of edges.

To see that this represents a coboundary, we consider the equation (3.21). The inner product $\langle \tilde{p}, \partial_2 c \rangle$ of 1-cochain \tilde{p} with boundary $\partial_2 c$ of 2-chain c represents the number of times (modulo 2), this cochain crosses this boundary. The inner product $\langle \tilde{\partial}^1 \tilde{p}, c \rangle$ represents the number of dual lattice vertices of the 2-cochain (modulo 2) $\tilde{\partial}^1 \tilde{p}$ that lie inside the plaquettes of 2-chain c .

Via the same argument as above (and as depicted in figure 3.22) these values coincide, confirming that this definition provides a valid co-boundary map.

3.3.8 2-coboundary

The 2-cochains correspond to subsets of vertices on the dual lattice. In analogy to 0-chains we expect that the 2-coboundary of any 2-cochain will be the null 3-cochain. A null (co)chain has the property that the inner product with every (chain)/(cochain) must be zero. This is trivially correct since the only 3-chain we consider in a 2-dimensional space is the null 3-chain, and thus for all 2-cochains p and all the null 3-chain c we have:

$$\langle \tilde{\partial}^2 \tilde{p}, c \rangle = \langle \tilde{p}, \partial_3 c \rangle = 0 \quad (3.22)$$

3.3.9 Cohomology groups

The coboundary allows us to define the group of cocycles Z^n , as the kernels of the coboundary maps, and the coboundaries B^n as the image of the coboundary map. We then can define the n th cohomology group as the quotient group:

$$H^n = \frac{Z^n}{B^n} \quad (3.23)$$

Representatives of the cohomology groups on the torus are illustrated in figure 3.23. The cohomology groups are topological invariants.

We can define Betti numbers β^n as the rank of the n th cohomology group. The relationship between homology and cohomology groups forms a rich

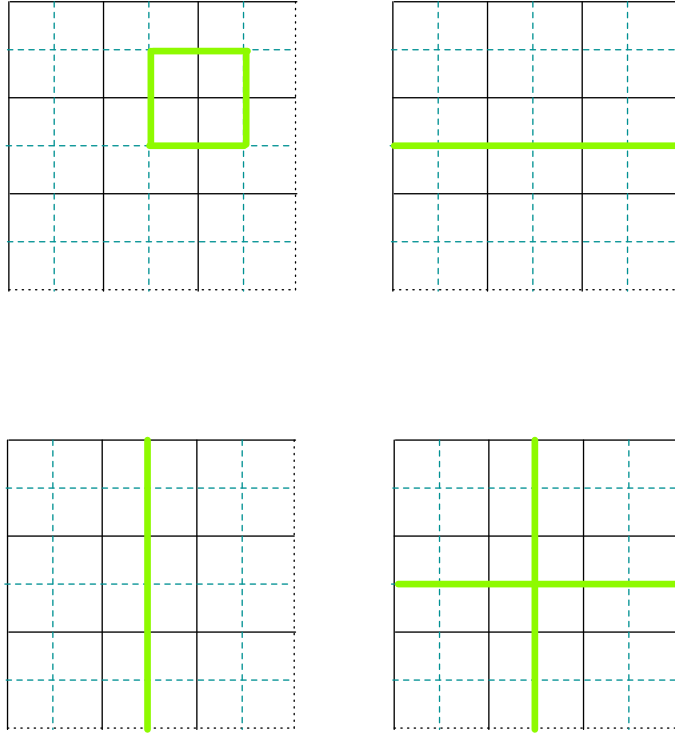


Figure 3.23: Representatives of each equivalence class (each element of Z^1/B^1) in the 1st cohomology group of the torus.

area of study, however, for the surfaces on which we define topological codes, they satisfy a simple relationship. For a

The rank of the n th cohomology group for the compact closed surface is equal to the rank of the n th cohomology group, hence:

$$\beta^n = \beta_n \quad (3.24)$$

3.3.10 Cohomology vs homology

In cellular \mathbb{Z}_2 homology there is a strong symmetry between homology and cohomology, and we can think of cohomology as simply “homology on the dual lattice”. This symmetry does not hold more generally, and in particular additional algebraic structures arise in cohomology which are not present in homology. See this article by Ulrike Tillmann <http://people.maths.ox>.

ac.uk/tillmann/coho.pdf for a discussion of this point and a general survey of cohomology. For topological codes, however, the duality between homology and cohomology is precisely what we need to neatly define a quantum code. We shall see how topological codes are naturally and generally defined using homology concepts in the next part of this course.

3.3.11 Homology for higher dimensional spaces

So far, we have only considered \mathbb{Z}_2 homology for 2-dimensional surfaces. However, the concepts above immediately generalise to spaces of *arbitrary spatial dimension*. At each n -dimension, we can define n -chains and n -cochains, and boundary and coboundary maps down to the $n + 1$ -chains and $n - 1$ -cochains. The machinery of homology, and the above definitions go through unchanged. All that is needed is a modification / augmentation of the definitions of boundaries to include these higher dimensional objects.

For example, we can cellulate a 3-D surface with a cubic lattice. The cubes define a generating set for the 3 – *chains*. The boundary of a cube generator 3-chain is the set of 6 plaquette faces (a 2-chain) of the cube.

The dual of a 3D cubic lattice is a cubic lattices with *vertices* placed at the centre of each cube. Thus the 3-cochain in 3d are generated by vertices on the dual lattice, 2-cochains are generated by edges on the dual lattice, and the 1-cochains by faces on the dual lattice. Notice that the pairings here differ from in 2D. Co-boundaries are defined precisely as one would expect, via boundaries on the dual lattice.

Harder (impossible?) to visualise are hypercubic lattices in higher dimensions, but homology can be constructed in an analogous way- e.g. the “faces” of a hypercube in 4-D are the set of 8 cubes which form its edges. The homology boundary maps can provide the structure to define and study homology in higher dimensional spaces. These higher dimensional homologies can be used to define analogues of the toric code in higher spatial dimensions, which can have important properties that are not shared by the 2-dimensional case. For example, the 4-dimensional toric code, which we will return to, is a self-correcting memory for robust data storage without constant active error correction via stabilizer measurement.

3.3.12 Homology beyond \mathbb{Z}_2

\mathbb{Z}_2 homology suffices to describe qubit topological codes, however it is just one form of homology. Homology can be based on other groups. We shall not introduce these forms of homology here. The main difference is that, in more general groups, the elements are not self-inverse, so the inverse elements need to be treated more carefully. This is using oriented cellulations. The inverse element for each cell can be associated with flipping its orientation. For more details, we refer the interested reader to P. Giblin's *Graphs, Surfaces and Homology* or to H. Anwar et al, <http://arxiv.org/abs/1311.4895>. The latter provides an introduction to \mathbb{Z}_d homology as a natural framework for qudit topological codes.

4 Topological codes from homology

Armed with these homology and cohomology tools, we can revisit the toric code. We will see that the code can be defined almost entirely in homological terms and that homology concepts will describe both logical operators and the relationship between errors and syndromes.

This will be more than just a refresh of terminology. Since homological concepts are properties of the *surface* which hold for *any* cellulation, the homological approach is much more general. The definitions we write down and properties we define homologically immediately apply to generalisations of the toric code based on any cellulation of any genus g closed surface.

We will later see that the family of topological codes defined in this manner can be broadened further, by extending these definitions to codes defined on surfaces with boundary.

4.1 Topological codes on surfaces with no boundary - a homological definition

Recall the definition of the toric code introduced in part 2. We defined the toric code as a stabilizer code on a square lattice with periodic boundary conditions. Qubits were associated with edges of the lattice, and stabilizer generators, and logical Pauli operators defined in terms of properties of the lattice and dual lattice (see part 2 for details).

Let us now recast these definitions in homological terms. The square lattice represents a *cellulation* of the surface, with edges representing 1-cells, vertices 0-cells and plaquettes 2-cells. The dual lattice represents co-cells (0-cocells are dual lattice plaquettes, 1-cocells dual edges and 2-cocells dual vertices). Boundary maps are defined exactly as before.

4.1.1 Qubits, Z-chain operators and X-cochain operators

We now define our code as follows. Let us first set the scene:

- For the chosen cellulation, we associate a qubit with every 1-cell and with the 1-cocell that intersects it.
- We now identify tensor products of Pauli Z operators with 1-chains and tensor products of X operators with 1-cochains.

More specifically, let us associate with every 1-chain c , the Z -chain operator $Z(c)$ which we can write

$$Z(c) = \bigotimes_{x \in c} Z^x \quad (4.1)$$

where x is the group element assigned to 1-cell in c , and $Z^0 = \mathbb{1}$. In other words, for every 1-cell assigned 1, the operator acts as Z on the associated qubit, and for every 1-cell assigned 0, the identity on that qubit.

Similarly, we can define a X -cochain operator $X(\tilde{p})$ as

$$X(\tilde{p}) = \bigotimes_{\tilde{x} \in \tilde{p}} X^{\tilde{x}} \quad (4.2)$$

where \tilde{x} is the group element assigned to 1-cocell in \tilde{p} , and $X^0 = \mathbb{1}$. Here, for every 1-cocell assigned 1, the operator acts as X on the associated qubit, and for every 1-cocell assigned 0, the identity on that qubit.

Note that the group composition for 1-chains and 1-cochains precisely captures the multiplication algebra for these subgroups of Pauli operators.

To summarise, 1-chains are identified with tensor products of Z and 1-cochains with tensor products of X . The commutation relation between these operators can be characterised homologically. An X -cochain operator will commute with a Z -chain operator if and only if the support of the two operators coincide on an even number of qubits. This is captured precisely by the inner product between cochains, and thus

$$X(\tilde{p})Z(c) = (-1)^{\langle \tilde{p}, c \rangle} Z(c)X(\tilde{p}) \quad (4.3)$$

The two operators commute if and only if the inner product between cochain and chain is zero. Note also that every n -qubit Pauli operator can be written $P = \alpha X(\tilde{p})Z(c)$ where $\alpha = \pm 1$ or $\alpha = \pm i$ and \tilde{p} and c are 1-(co)chains.

4.1.2 Stabilizer generators

The stabilizer generators will be associated with the boundaries and co-boundaries of 2-cells, and 2-cocells.

Let us consider first the plaquette generators. These were defined, for each plaquette in the lattice, as Z operators acting on the edges of the plaquette. The homological definition is thus very natural

- A *plaquette generator* is the operator associated with a 2-cell c and defined as the Z -chain operator $Z(\partial c)$ corresponding to the boundary of c .

We identify vertices on our lattice with 0-cells and corresponding 0 – *cocells*, vertex generators are then defined as X -cochains:

- A *vertex generator* is the operator associated with a 0-cocell \tilde{p} and defined as the X -cochain operator $X(\tilde{\partial} \tilde{p})$ corresponding to the boundary of c .

We can now prove that this set of operators commute for any cellulation. From equation (4.3), the operators commute if the inner product of chains and cochains are zero, i.e. if for all 2-cells c and for all 0-cocell \tilde{p} ,

$$\langle \tilde{\delta} \tilde{p}, \delta c \rangle = 0 \quad (4.4)$$

But via the definition of co-boundary,

$$\langle \tilde{\delta} \tilde{p}, \delta c \rangle = \langle \tilde{p}, \delta \delta c \rangle \quad (4.5)$$

but further, the fundamental lemma of homology states that $\delta \delta = 0$, hence

$$\langle \tilde{\delta} \tilde{p}, \delta c \rangle = \langle \tilde{p}, 0 \rangle = 0 \quad (4.6)$$

where we use the fact that the inner product with a null chain is zero.

Thus, defined in these terms, the commutation of the stabilizer generators follows from two foundational definitions in homology and cohomology theory. Since the above definitions are valid for any closed compact surface and any cellulation, the proof holds for any topological code defined according to any cellulation of a closed compact surface in the above manner.

Note that the plaquette/vertex generators are (by definition) the generators of the 1-boundary / 1-boundary groups B_1 and B_0 .

4.1.3 Logical operators

Having established that the code stabilizer can be defined (and its commutation verified) solely in homological (and cohomological) terms, we now wish to determine the number of encoded qubits in the code and complete their definition by specifying the logical Pauli operators.

Previously we determined the number of logical operators by counting independent stabilizer generators and qubits. However, these numbers are cellulation dependent. We need a more general homological approach.

This approach is given to us by the first homology and first cohomology groups. Recall that the logical Pauli operators are operators within the centraliser of the code - i.e. they commute with every stabilizer - and that they are divided into equivalence classes, where equivalence is up to multiplication by an element of the stabilizer.

It suffices to define encoded Z and X operators for every encoded qubit. Let us consider the encoded Z operators first. We wish to find the set of Z -chains $Z(c)$ which commute with every stabilizer generator. We are thus looking for the set of 1-chains c which satisfy:

$$\langle \tilde{\partial} \tilde{p} | c \rangle = 0 \quad (4.7)$$

for every 0-chain \tilde{p} . Now let us apply the definition of co-boundary:

$$\langle \tilde{\partial} \tilde{p}, c \rangle = \langle \tilde{p}, \partial c \rangle = 0 \quad (4.8)$$

for every 0-chain \tilde{p} . The only n -chain whose inner product is zero with *all* n -cochains is the null n -chain. Hence, equation 4.8 can only hold for every 0-chain \tilde{p} if $\partial c = 0$.

This is the set of 1-chains with zero boundary. Thus the set of Z -chains $Z(c)$ which commute with every stabilizer generator is the set where c is a 1-cycle, $c \in C_1$. Let us call such an operator a Z -cycle.

Analogously, one can show that the set of X -chains $X(\tilde{p})$ which commute with every stabilizer generator is the set of X co-cycles, where \tilde{p} is a 1-cocycle, $\tilde{p} \in C^1$.

Since every Pauli operator is a product of a Z -chain and an X -cochain up to a phase, we have identified the full centralizer of the code, consisting of products of a Z -cycle and an X -cocycle.

Thus 1-cycles and 1-cocycles on the lattice represent logical operators, but to complete the analysis we must group them into equivalence under

4.1. Topological codes on surfaces with no boundary - a homological definition

multiplication by a stabilizer element. Above we identified that the stabilizer operators consist of Z -chains defined by 1-boundaries, and X -cochains defined by 1-coboundaries. Equivalence up to stabilizer multiplication is therefore equivalence under addition with a (co)boundary - this is nothing other than the definition of (co)homological equivalence we encountered previously.

Therefore we can describe the equivalence classes of logical operators as the equivalence classes under homology. This is nothing other than the elements of the 1st homology group. Hence:

- The logical Z operators are identified the elements of the 1st homology group.
- The logical X operators are identified with the elements the 1st cohomology group.

One can verify that these sets of operators commute and anticommute as is necessary to represent these operators (proving this is left as an exercise - one can choose a set of chain generators for H_1 and a set of cochain generators for H^1 such that each chain generator intersects with just one cochain generator).

The logical Pauli operators are thus fully characterised by the 1st homology groups of the surface. The number of encoded qubits is then determined by the rank of this group - it follows that the 1st Betti number β_1 , the rank of H_1 determines the number of qubits encoded in the code. (Note that isomorphism between H_1 and H^1 ensures that the two groups have equal rank, as is necessary (since we need equal numbers of independent encoded X and Z operators)).

Note that the distance of the code - the minimal weight of the set of non-identity logical Pauli operators - is not determined by the homology of the code and is therefore not a topological property. It depends upon the detailed implementation of the code (e.g. for the toric code on a square lattice, the size of that lattice).

4.1.4 Error detection

The error detection and correction properties of these codes can also be understood in homological terms. From the analysis in part 2, the conclusion is almost immediate.

Let us consider first Z -type errors only. Here the error operator is now represented by a Z -chain $Z(c)$. Errors are detected by measuring generators of the stabilizer, and the syndrome for this error will be represented by the set of vertex generators which anticommute with $Z(c)$, i.e. the set of 0-cocells \tilde{v} which satisfy,

$$\langle \tilde{\partial} \tilde{v}, c \rangle = 1 \quad (4.9)$$

but from the definition of coboundary:

$$\langle \tilde{\partial} \tilde{v}, c \rangle = \langle \tilde{v}, \partial c \rangle = 1 \quad (4.10)$$

This is only satisfied when the primal lattice vertices corresponding to \tilde{v} lie in the boundary of c . Hence the set of vertices whose outcomes are flipped by this error are the 0-cocells which intersect the boundary of c . On the primal lattice, we'd thus identify the error syndrome corresponding to c with the set of vertices at its boundary ∂c .

Similarly, the syndrome associated with an X -cochain is the set of plaquettes at the coboundary of this cochain.

4.1.5 Error correction

Recall that to *correct* an error, it suffices to apply a correction operator which is equal to it up to multiplication with an element of the stabilizer. This has a direct homological interpretation. A correction chain $Z(c')$ will correct error chain $Z(c)$ if c and c' are homologically equivalent.

If a decoder returns a correction chain d which shares the boundary of c but is in a different homological equivalence class, then applying this correction will lead to an encoded error and represents a correction failure.

This provides a compact description of the optimal decoder - it is the decoder which returns the most likely homological equivalence class for the error consistent with the observed syndrome.

The minimal weight perfect matching algorithm, on the other hand, returns the shortest weight correction chain (not a homological concept) which is usually, though not always in the most likely homological equivalence class. This is the reason why this decoder performs well - but is not optimal.

4.1.6 Some examples and some thresholds

The above homological formulation is valid for any cellulation of any closed (i.e. boundaryless) compact surface. We therefore can immediately generalise the toric code.

For example, while keeping the torus as the defining surface, we can modify the cellulation to other lattices. For example, we can replace a square lattice with a hexagonal lattice, a kagome lattice, or indeed an irregular lattice.

The square lattice has the special feature that it is self dual - e.g. if the dual of a square lattice is a square lattice. However, most lattices are not self-dual. For example, the dual of the hexagonal lattice is a triangular lattice, while the dual of the kagome lattice is the rhombic star.

The effect of this asymmetry is that codes defined by such cellulations protect against Z and X errors to different degrees and have different code thresholds. The following thresholds were estimated recently by Fujii and Tokunaga, Physical Review A, 86, 020303(R) (2012), <http://arxiv.org/abs/1202.2743>. Their estimates of thresholds for the minimum weight perfect matching decoder (so lower bounds to optimal thresholds) results are summarised in the table below:

Primal lattice	Z-error threshold	Dual Lattice	X-error threshold
Square	0.103	Square	0.103
Kagome	0.116	Rhombic star	0.095
Hexagonal	0.159	Triangular	0.065

This emphasises that the threshold of the toric code is *not* a topological property.

4.1.7 High genus codes

As already remarked, high genus g closed surfaces can be used to define codes. The 1st Betti number for such surfaces is $\beta_1 = 2g$ which tells us there are $2g$ encoded qubits is $2g$, twice the number of handles in the surface.

4.1.8 Higher dimensional codes

In this lecture course, we will focus on 2-d surfaces, due to time constraints. However, the above homological definitions generalise immediately to higher

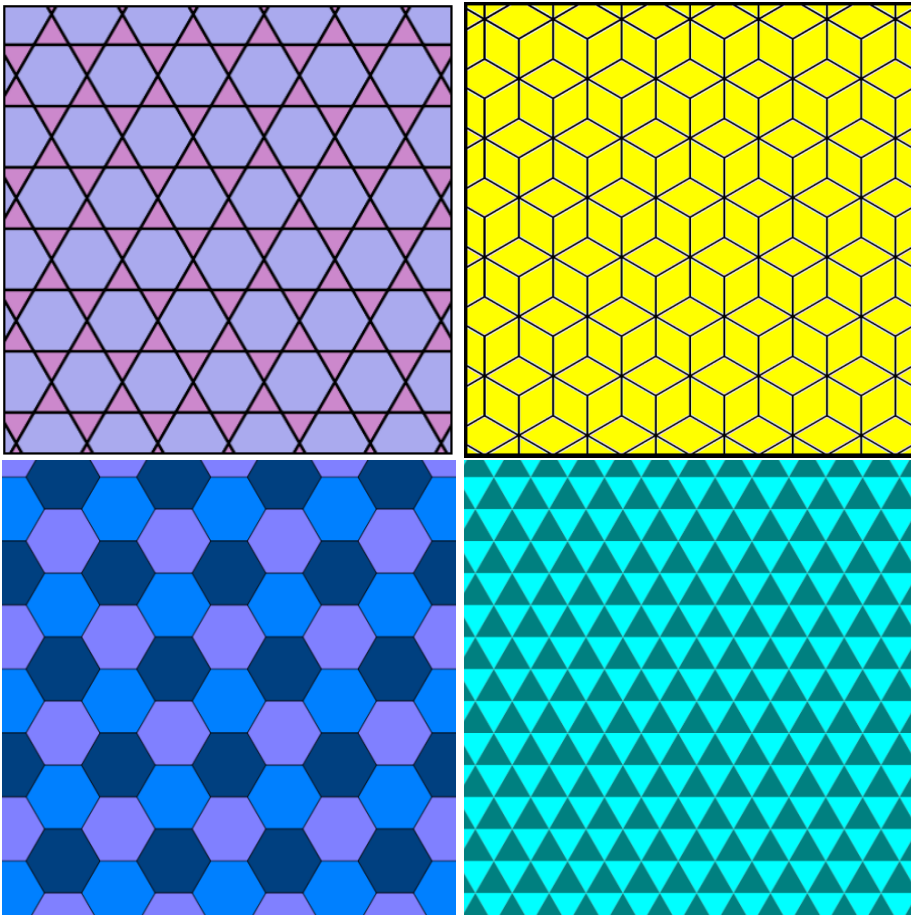


Figure 4.1: Examples of some regular lattices. The kagome lattice is dual to the rhombic star lattice. The hexagonal lattice is dual to the triangular lattice. Images from Wikipedia.

dimensional spaces. Where we identified qubits and qubit operators with 1-chains and 1-cochains, in higher dimensions, we can define codes where qubits are identified with n -chains and n -cochains. All of the above homological definitions are derivations go through unchanged.

For example, in the 4-dimensional toric codes, qubits are identified with 2-*chains* and 2-*cochains*, the 2 dimensional faces of the three-dimensional cubes which form the “faces” of the hypercube 4-cells. The stabilizer generators correspond to the boundary of 3-cells and 1-cocells (cubes on the primal and dual lattices) and thus are weight 6 (the cube has 6 faces). Logical operators here are identified with the 2nd homology and cohomology groups, which have rank 6. The 4-dimensional toric code thus encodes 6 logical qubits.

This change in dimensionality gives the 4-dimensional toric code some properties not shared by its 2-d equivalent. Most importantly, it can be used to define a self-correcting quantum memory. For more details, see Dennis et al, Physical Review A, J. Math. Phys. 43, 4452-4505 (2002) <http://arxiv.org/abs/quant-ph/0110143>.

4.2 Surfaces with boundary – Planar codes

So far, we have focussed almost entirely on *closed* surfaces, surfaces without a boundary, both in terms of homology and in terms of the codes we consider. A disadvantage of this from the code point of view, is that the practical advantage of using local stabilizers, stabilizers where the measurement is on a small subset of close-lying qubits, is lost by the need to construct a toric surface. It would be desirable to construct a code on a flat *planar surface*. However, finite planar surfaces can never be closed – they always have a *boundary*.

In this section we will consider homology on surfaces with boundary and show how they can be used to define topological codes. We will see in the following section that the presence of boundaries provides extra structures that give us new ways to define qubits.

4.2.1 Homology of a simple planar surface

Consider the simple cellulation illustrated in figure 4.2. This surface has a single boundary surround it, and is homeomorphic to a disk. In common

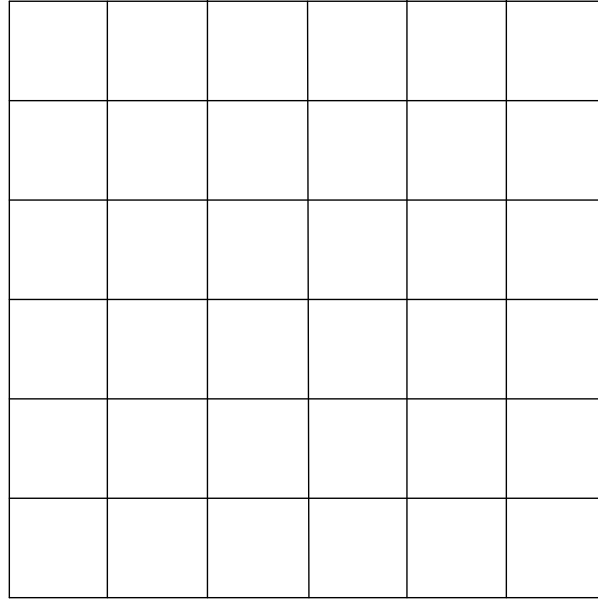


Figure 4.2: A cellulation of a disk, its single boundary surrounds the entire surface.

with the surfaces we have seen before, we can identify 0-cells (vertices), 1-cells (edges) and 2-cells (plaquettes), and thus can define 0-chains, 1-chains and 2-chains entirely analogously to before.

Just as before we can identify cycles, boundaries and thus homology groups. The 0th homology group is very similar to the cases we saw earlier, we divide the 0-chains into subsets of even numbers of vertices (in B_1) and odd numbers of vertices (not in B_1). H_0 is thus isomorphic to \mathbb{Z}_2 , and thus $\beta_0 = 1$.

The 1st homology group for this surface is trivial, *every* closed loop of edges is the boundary of a 2-chain. Thus H_1 is also isomorphic to the trivial group \mathbb{Z}_1 , and thus $\beta_1 = 0$.

Both of these examples so far have been very similar to the homology groups of a sphere. The second homology group H_2 exposes the first difference with respect to closed surfaces. The set of 2-cycles for the sphere consisted of two inequivalent 2-chains, the null 2-chain and the “all ones” 2-chain. The all-ones 2-chain was a cycles since the surface had no boundary. If we now return to our example of a surface with boundary we see that the

“all-ones” 2-chain is no longer a cycle – it has a boundary – the boundary of the surface. Thus the second homology group H_2 contains just one element and is isomorphic to the trivial group. Thus $\beta_2 = 0$.

We saw previously (and you can verify by counting the figure) that $\chi = 1$ for a surface with a single boundary. We can verify that the Betti numbers and Euler characteristic satisfy the relation $\chi = \beta_0 - \beta_1 + \beta_2$.

Since the number of encoded qubits in a surface is equal to β_1 , there would be no encoded qubits on a topological quantum code defined by a disk. Before we consider cohomology of planar surfaces, therefore let us consider a more complicated surface - a surface with 2 boundaries.

To summarise the homology groups of a disk:

- $H_0 = \mathbb{Z}_2, \beta_0 = 1$
- $H_1 = \mathbb{Z}_1, \beta_0 = 0$
- $H_2 = \mathbb{Z}_1, \beta_0 = 0$

4.2.2 Homology of a disk with a hole

Now consider the cellulation illustrated in figure 4.3. This surface is homeomorphic to a disk with a single hole in it. The hole provides a second boundary, (thus this surface has two boundaries) and means that the “plaquettes” inside the whole is *not part of the surface*.

Let us first compute the Euler characteristic of the surface. After a count we compute that $\chi = 32 - 80 + 48 = 0$. The zeroth homology group H_0 and the second homology group H_2 are unchanged by this new hole. The reasoning used to derive them for the disk applies here as well. It is unaffected by the hole.

The first homology group, however, is changed by the hole in the surface. Consider the 1-chain indicated in figure 4.4. This is clearly a 1-cycle, however it is the boundary of no 2-chain. This surface therefore has a non-trivial first homology group.

You can convince yourself that all 1-cycles which encircle the whole an odd number of times are inequivalent to the 1-cycles which encircle the hole an even number of times. The 1-st homology group therefore consists of two equivalence classes and is isomorphic to \mathbb{Z}_2 . The first Betti number is $\beta_1 = 1$.

For surfaces with no boundary, the first Betti number was equal to the number of qubits encoded on any topological code defined via a cellulation

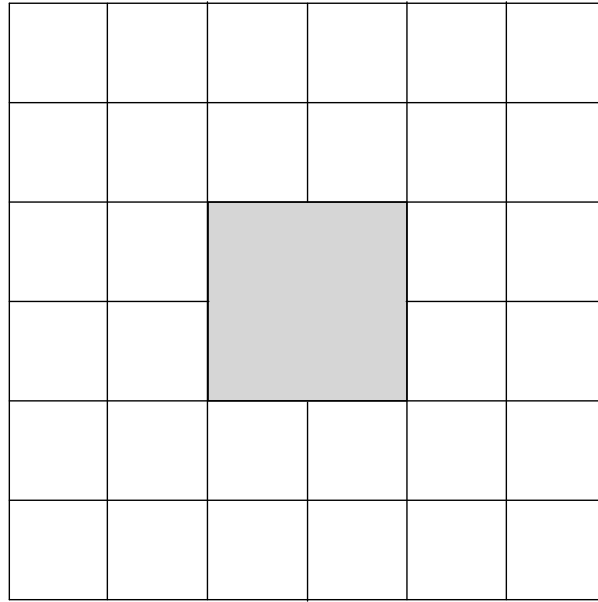


Figure 4.3: A cellulation of a disk with a single whole. The grey area inside the whole is not part of the surface. This surface has two boundaries – around the edge and around the hole.

on that surface, using the definitions in part 4.1. We might therefore hope that the disk with a whole would encode a logical qubit in a similar manner. To generalise our definition of topological codes, however, we first need to explore the cohomology of surfaces with a boundary.

Before we do so, let's summarise the homology groups for a disk with a hole (a surface with 2 boundaries)

- $H_0 = \mathbb{Z}_2, \beta_0 = 1$
- $H_1 = \mathbb{Z}_1, \beta_0 = 1$
- $H_2 = \mathbb{Z}_1, \beta_0 = 0$

4.2.3 Cohomology of surfaces with a boundary.

We define cochains for surfaces with a boundary in the same way as for closed surfaces. Formally the cochains represent linear functionals (group

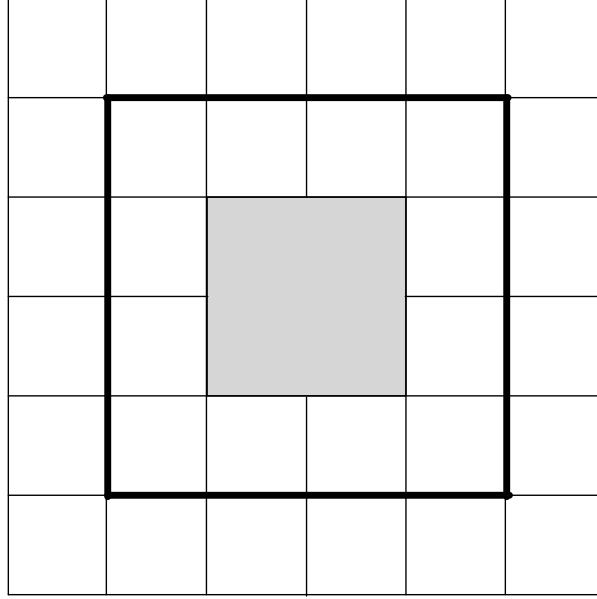


Figure 4.4: A non-trivial 1-cycle. This 1-cycle is not the boundary of any 2-chain. Note that this non-trivial cycle “wraps around” the hole, similar to how the non-trivial cycles on the torus wrapped around the handle. In contrast to the torus, however, there is only one way to encircle the hole. All cycles which encircle the hole are homologically equivalent. The first homology group is thus isomorphic to \mathbb{Z}_2 and the first Betti number $\beta_1 = 1$.

homomorphisms to \mathbb{Z}_2) on each chain group. More concretely, we can make the same identification as before – identifying cocells with vertices, edges and plaquettes on the dual lattice.

Thus the 0-cocells are associated with plaquettes on the dual lattice dual to the vertices on the primal lattice, the 1-cocells are associated with dual lattice edges, and 2-cocells with dual lattice vertices. You can verify that the n -cocells defined in this way do generate the full n -cochain groups via the inner product definitions that were defined in part 3 – to recap, the 0 and 2-cochain / chain inner products count the number mod 2 of vertices of the (co)chain lying inside plaquettes from the (co)chain. The 1-chain / cochain inner product is the number mod 2 of edges of the 1-chain which intersect the edges of the cochain.

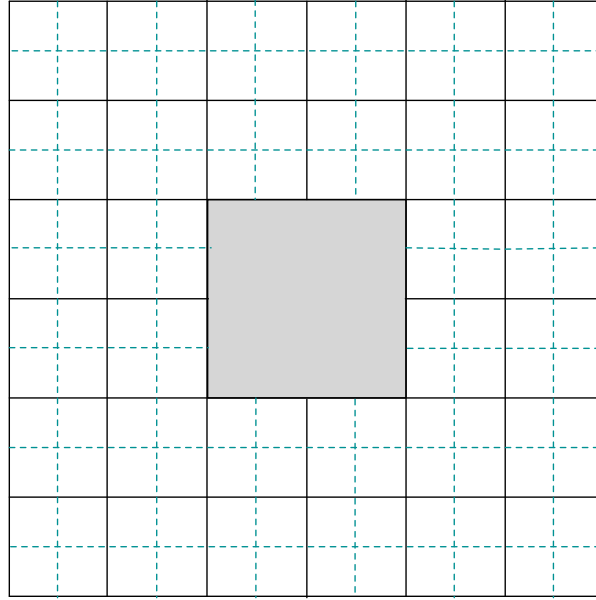


Figure 4.5: The dual lattice cellulation depicted together with the primal cellulation for a disk with a hole.

So far, the presence of a boundary on this surface has not had a large effect on our definitions. However, the presence of boundaries has important implications for the *coboundary map*. Consider, a dual lattice edge (1-cocell) adjacent to the boundary. Normally the coboundary of a dual lattice edge would be the *two* dual lattice vertices adjacent to the edge. For a dual lattice edge adjacent to the boundary, however, there is only *one* adjacent dual vertex. The other “end” of the edge is beyond the boundary of the lattice where there is no dual vertex. We therefore must define the coboundary map for this dual lattice edge as the *single* dual lattice vertex adjacent to it, as depicted in figure 4.7. The reader can verify that this is a valid coboundary map satisfying the defining equation (3.21).

The 2-boundary map for 0-cocells (dual lattice plaquettes) adjacent to the edge is similarly affected. Such plaquettes may have one or two of its expected coboundary dual lattice edges outside of the surface. We thus redefine the boundary map for these 0-cocells accordingly (see figure 4.8). 2-coboundary maps are unaffected by the presence of a surface boundary. The 2-coboundary map sends every 2-cochain to 0.

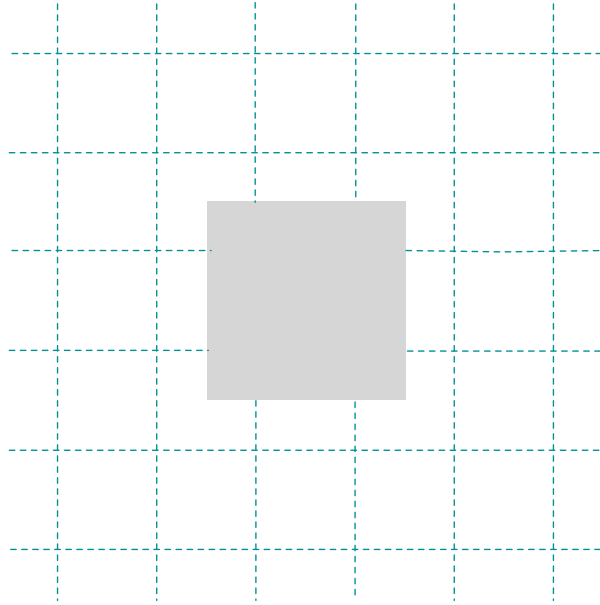


Figure 4.6: The dual lattice cellulation of a disk with a hole depicted on its own. Note the effect that a boundary on the primal lattice has on the “form” of the dual lattice near that boundary.

$$\tilde{\partial} \left(\begin{array}{|c|} \hline \text{---} \\ \hline \end{array} \right) = \begin{array}{|c|} \hline \bullet \\ \hline \end{array}$$

Figure 4.7: When a surface has a boundary, the 1-coboundary map for the 1-cocells adjacent to the boundary must be modified as shown.

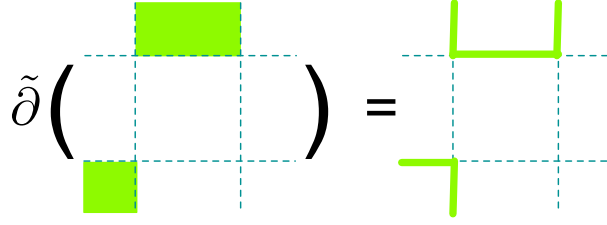


Figure 4.8: An example of the 0-coboundary map for a 0-cochain consisting of two dual plaquettes adjacent to the surface boundary. The coboundary map is modified as shown to remove dual lattice-edges which would lie outside the surface.

The redefined coboundary maps have a profound affect on the cohomology groups. Let us first consider the group of 1-cocycles. Recall that the 1-cocycles are the group of 1-cochains with no coboundary (i.e. their coboundary is the null cochain). As before, closed “loops” of edges represent 1-cocycles. However, there is now a new type of possible 1-cocycle. A “string” of edges which passes starts and ends at a boundary also has null coboundary (due to the modified coboundary map). It is thus also a 1-cocycle. See figure 4.9 for examples of 1-cocycles.

Of these 1-cycles, which “begin” and “end” at the primal lattice boundaries can be split into two homological equivalence classes. Those whose source and termination is the *same* boundary are homologically equivalent to the null 1-cocycle, while those that connect two *different* boundaries form a distinct homology class, inequivalent to the null 1-cocycle. There are, in fact, just two homological equivalence classes of 1-cocycles. Those homologically equivalent to the null cocycle, and those homologically equivalent to a string between the two different boundaries. The group H_1 is therefore isomorphic to Z_2 and its rank, the second Betti number β^2 is 1.

For completeness, we should briefly consider the group H^0 . We find that this group is isomorphic to Z_2 , since, just like the closed surface, the “all-ones” 0-cochain has null boundary (this is due to the “missing” dual edges which would lie outside the surface), and, hence, like a closed surface, H^0 has two inequivalent elements.

We see that the non-trivial H_0 group – reflecting the fact that in the primal lattice, every edge (1-cell) has a boundary chain containing 2 vertices –

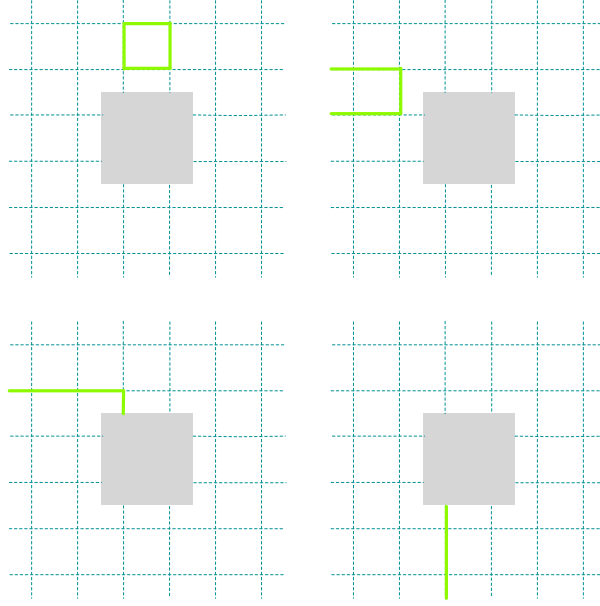


Figure 4.9: 1-cocycles on a surface with a pair of boundaries. As well as forming loops, 1-cocycles can “begin” and “end” at the primal lattice boundary. The upper two 1-cocycles are representatives of the trivial homological equivalence class, the lower two of the (only) nontrivial class.

goes hand in hand with the non-trivial H^0 group – which reflects the lack of a “boundary” on the dual lattice. At the same time, the trivial H_2 group – reflecting the fact that every 2-chain except the null chain now has a boundary, so Z_2 has only one element – is matched by a trivial H^2 group, where the fact that some dual lattice edges have only one dual vertex in their coboundary means that every 2-cochain is a 2-coboundary. This symmetry is just a hint of the elegance of cohomology.

To summarise:

- $H_0 = Z_2, \beta_0 = 1$
- $H_1 = Z_1, \beta_0 = 1$
- $H_2 = Z_1, \beta_0 = 0$

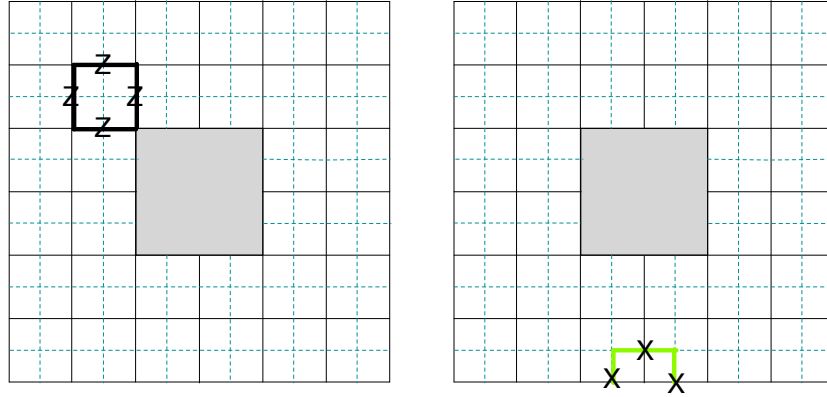


Figure 4.10: Examples of plaquette and vertex stabilizer generators for the “disk with hole” planar code. Note that the presence of the boundary lowers the weight of the adjacent vertex operators, but does not do so for adjacent plaquette operators (reflecting the fact that the presence of the boundary modified the coboundary maps not the boundary maps).

For us, these symmetries have a fortunate consequence, since we need H_1 to be isomorphic to H^1 to define a quantum code. The Betti numbers $\beta_1 = 1$ implies that this code encodes 1 qubit, and we shall see an explicit construction in the next section.

4.2.4 A planar surface code on a disk with a hole

Using the definitions of boundary and coboundary defined at the beginning of this chapter, we can immediately define a quantum code. This is the *planar surface code* for the disk with a hole.

Plaquette stabilizer generators are defined as the Z-chains $Z(\partial(c))$ for every 2-cell c . Vertex operators are also defined via the coboundary map, and their form is modified at the boundary reflecting to modified coboundary maps here (see figure 4.10).

A single qubit is encoded in the code (since the 1st Betti number is 1) and logical operators are defined by the generators of the first homology and first cohomology groups defined above.

Notice that we did not need to consider the independence or otherwise of the plaquette and vertex operators to determine the number of encoded

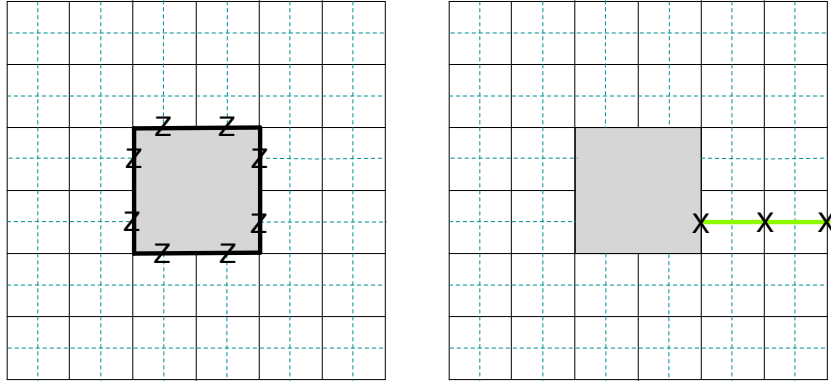


Figure 4.11: Examples of logical Z and logical X operators with minimum weight. Note that this minimum weight is determined by the “radius” of the hole and how far it lies from the boundary of the surface.

qubits. The homology and cohomology – specifically the 1st Betti numbers – take care of this for us. (As an exercise, work out the relationship between the independence of the plaquette and vertex generators here, and the 0th and 2nd Betti numbers).

Looking at figure 4.11 we see that the weight of the lowest weight Z operator is given by the *circumference* of the hole and the weight of the lowest weight X operator is given by the minimum (spatial) *distance* (plus one) between the boundary of the surface and the boundary of the hole. We call this type of qubit a *primal hole qubit* (below we will meet “dual hole qubits”).

Error detection and correction for such a qubit is analogous to the toric code, and similar decoder strategies (e.g. minimum weight perfect matching) can be used. Note, however, that there are now only 2 homology classes for errors on this code. The code threshold for such qubits was found (by Dennis, Landahl, Kitaev and Preskill) to be similar to the (square lattice) toric code at approximately 11%.

4.2.5 Surfaces with many holes

What if add further “holes” to the surface? This is straightforward to analyse using the same methods as above. The number of qubits is equal to the first Betti number of the surface. We find that, in general, such a code with n holes (and thus with $(n + 1)$ boundaries has Betti number $\beta_1 = n$ and will

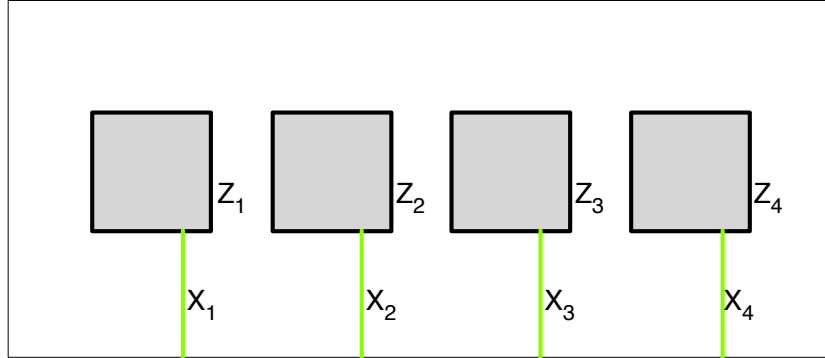


Figure 4.12: A surface with (primal) boundary and n (primal) holes encodes n qubits. Minimal weight logical operators are indicated, the thick black lines, 1-cycle Z operators which enclose the holes, and the green lines 1-cocycle X operators stretching from the hole boundary to the surface boundary. It is clear that these operators respect the commutation relations for X and Z since each X -cochain intersects only with its respective Z cochain.

thus encode n logical qubits. Logical Z and X operators for such a code are illustrated in figure 4.12.

4.2.6 A different kind of boundary - “dual boundary” and “dual holes”

There was a striking asymmetry between the effect of adding a boundary to the surface between the boundary and coboundary maps and the first homology and first cohomology groups.

The boundary operators and first homology group for this surface was very similar to what we saw for closed surfaces. The coboundary operators, however, needed to be modified to take the boundary into account, and the weights of the boundary chains for 1-cells and 2-cells adjacent to the boundary was reduced. This had an effect on the structure of the cohomology groups. For example, the 1st cohomology group now consisted of stringlike 1-chains stretching from one boundary to the other, rather than “loop-like” chains.

This asymmetry can be very clearly seen in the “shape” of the boundary itself. See figure 4.13 for an example. On the primal lattice, we see a “smooth” boundary, but on the dual lattice, the boundary is “spiky” or

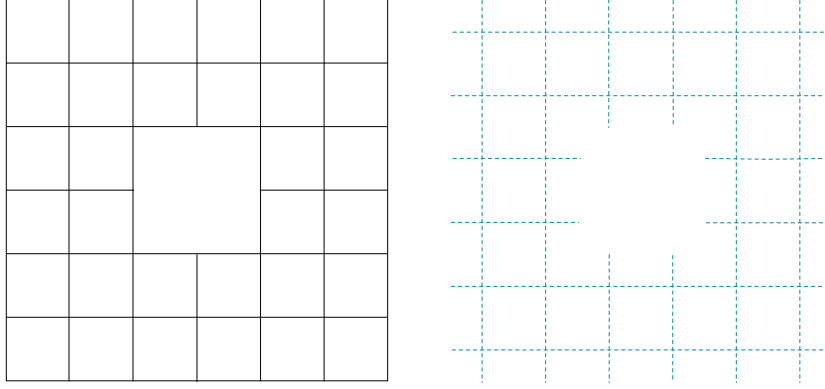


Figure 4.13: The primal lattice (left) and the dual lattice (right) for a surface enclosed by a (primal) boundary with a single (primal) hole.

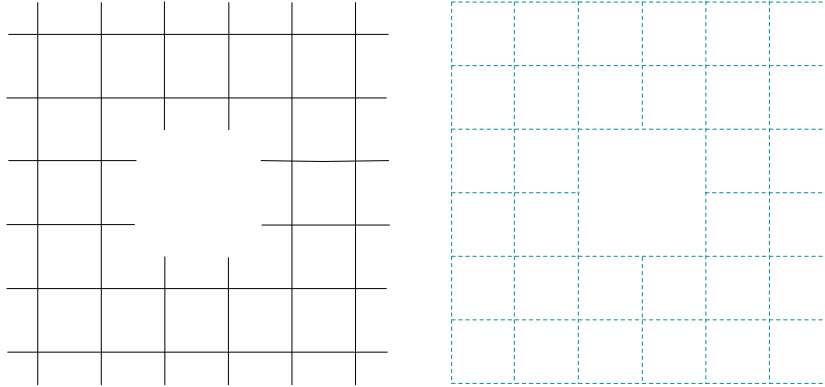


Figure 4.14: The primal lattice (left) and the dual lattice (right) for a surface enclosed by a dual boundary with a single dual hole.

“rough” with adjacent dual lattice edges connecting with no dual vertex, and adjacent dual lattice plaquettes missing one or more sides – both effects forcing us to redefine the boundary map. Similarly, around the hole in the lattice, we see a smooth boundary on the primal and a spiky boundary on the dual lattice.

We can, however, reverse these properties and define a *dual boundary*. A dual boundary creates a smooth boundary on the dual lattice and a spiky

boundary on the primal lattice – see figure 4.14. We call a hole which has a dual boundary a *dual hole*. To distinguish these two types of boundary, we call the type of boundary introduced previously a *primal boundary* and a hole with such a boundary a *primal hole*.

When a surface has a dual boundary or a dual hole, the boundary maps for 1-cells (edges) and 2-cells (plaquettes) along that boundary need to be redefined - removing the vertices and edges which would lie beyond this dual boundary and which are not part of the surface.

We can construct the homology and cohomology groups for this surface – a cellulation of a disk with dual boundary and a dual hole – and construct a corresponding quantum code. This code is depicted in figure 4.15. This surface encodes a single bit, as expected, due to the similarity with a primal hole qubit, and we see that the logical operators share the structure of the primal hole qubit, just with the logical X and logical Z exchanging roles. The 1st cohomology group generator (and hence logical X) is now a loop around the dual hole, while the 1st homology group generator (and hence logical Z) is a stringlike chain connecting the two dual boundaries. We call a qubit defined in this way a *dual hole qubit*.

4.2.7 Surfaces with mixed boundary

Having defined primal and dual boundaries, we can now consider surfaces with *both* types of boundary. Consider the surface in figure 4.16. The boundary of this surface is divided into 4 regions, alternating primal - dual - primal - dual. The alternating boundary allows for the existence of a non-trivial first homology and cohomology groups – (as an exercise, convince yourself that a surface whose boundary is divided into a single primal boundary and a single dual boundary has only trivial 1st (co)homology groups with Betti number $\beta_1 = 0$ and thus encodes no qubits).

We find that the first Betti number for this surface is 1, and that 1st homology and cohomology groups are isomorphic to \mathbb{Z}_2 - a single qubit can therefore be encoded. The generators of the (co)homology groups and hence the encoded logical operators are depicted in figure 4.17. The distance of this code is determined by the width or height (whichever is smallest) of the lattice.

By alternating the boundary type further one can encode more qubits. Determine, as an exercise, how many qubits are encoded on a disk where the boundary alternates n times (where n is an even number).

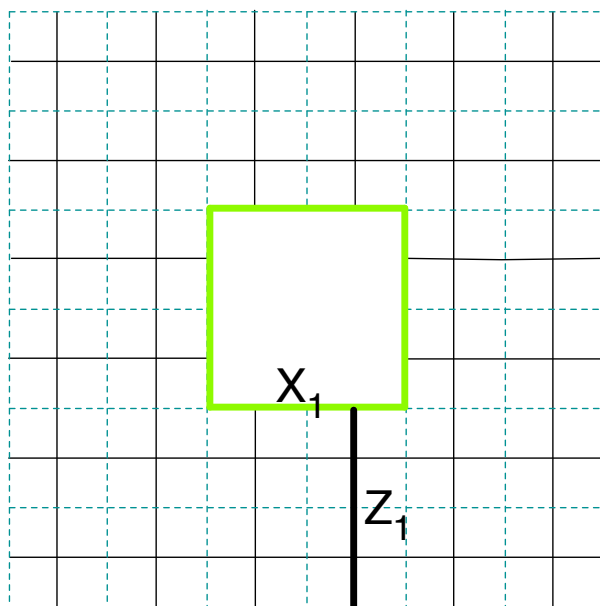


Figure 4.15: A dual hole qubit, defined on a surface enclosed by a dual boundary with a single dual hole.

4.2.8 Twin-hole qubits

We have seen that the presence of primal and dual boundaries provides new ways to create qubits on a planar surface code, and that this can be achieved by inserting primal or dual holes into the surface, and by alternating the boundary type along a single boundary.

The planar surface code is the basis for a fully fault tolerant universal model of quantum computation - developed by Raussendorf, Harrington and Goyal within an equivalent (but different in some details) measurement-based quantum computation framework – which we will survey in the final part of this course.

In fault tolerant quantum computation, it is useful to be able to change the distance of an encoded qubit, initialise fresh ancilla qubits and remove unneeded measured qubits. In the next part, we will show explicitly how this can be achieved via *code deformation*, changing the shape of holes and boundaries, to change a codes properties. On a toric code or the mixed boundary planar code illustrated above, to change the code distance you

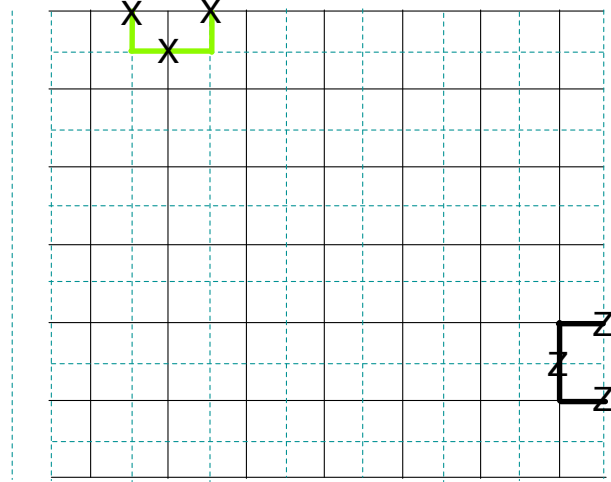


Figure 4.16: A cellulation of a disk with 4 alternating boundary types - primal, dual, primal, dual. The surface defines an encoding of a single qubit as a topological code. Some example stabilizer generators (reflecting the redefined boundary maps due to the boundaries) are depicted.

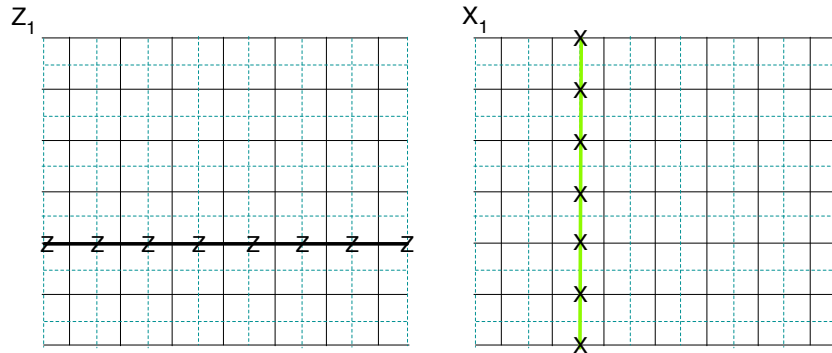


Figure 4.17: The encoded logical X and Z operators for the cellulation of a disk with 4 alternating boundary types - primal, dual, primal, dual.

need to change the lattice dimension or change the boundaries. While possible in principle, this would be cumbersome in practise.

A more elegant solution is provided by twin-hole qubits. In the next section we will see how such qubits can be robustly created, initialised and how their distance can be changed. Here, we shall define the qubits.

The basic idea in a twin-hole qubit is to use two holes of the same type – a primal with a primal or a dual with a dual – to represent a single qubit. This is an example of a very simple repetition encoding.

Consider the 2-qubit repetition code, with code words

$$|0\rangle_L = |0\rangle|0\rangle \quad |1\rangle_L = |1\rangle|1\rangle \quad (4.11)$$

The stabilizer for this code is Z_1Z_2 and the logical operators are $\bar{Z} = Z_1$ and $\bar{X} = X_1X_2$.

We can create such a repetition code using two primal hole qubits on a planar surface code. Consider the two primal holes depicted in figure ?? . We can define a redundantly encoded qubit from these two holes by defining encoded $\bar{Z} = Z_1$ and $\bar{X} = X_1X_2$. In this way we have a single qubit represented by the pair of holes.

There are a number of advantages in this representation. Notice that logical operators longer contains strings to the surface boundary and thus the encoded qubit is more “self-contained” with the pair of holes. This becomes useful when one wants to encode many qubits in a surface and implement encoded quantum gates.

Notice also that the weight of the code depends only on properties of the pair of qubits. The minimal weight of the logical Z is given by the radius of the smallest hole, while the minimal weight of the logical X is determined by the distance between the holes.

The two-qubit repetition code has a single stabilizer generator Z_1Z_2 . To fully protect this encoding, we would need to measure this operator - and add it to our set of stabilizer generators. But this operator is of very high weight - consisting of Pauli Z 's around the boundaries of both holes. We would not want to have to measure such an high-weight operator to check errors. Fortunately, we don't need to. The surface code is protecting the individual hole qubits, and the sort of error which the Z_1Z_2 would detect would be a logical X on one of the qubits. Thus if the individual hole qubits are protected against X errors (which they are, in the surface code) the twin hole qubit is protected, and the encoded Z_1Z_2 operator does not need to be measured.

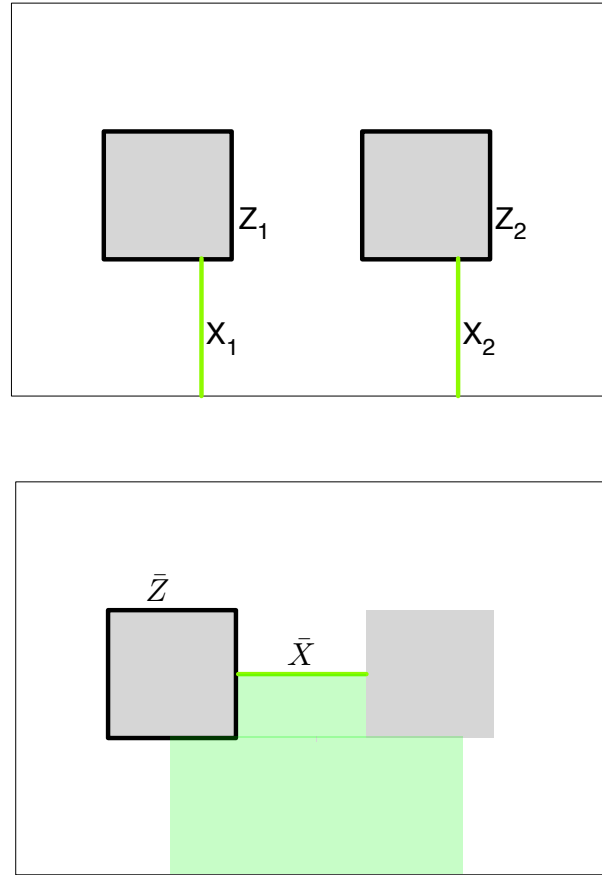


Figure 4.18: An example two hole qubits, and the corresponding twin (primal) hole qubit, created from them by using a 2-qubit repetition code, where $\bar{Z} = Z_1$ and $\bar{X} = X_1 X_2$. The shaded green area represents the 2-cocell whose boundary operator transforms the product of operators X_1 and X_2 to the homologically equivalent, but lower weight, form depicted here.

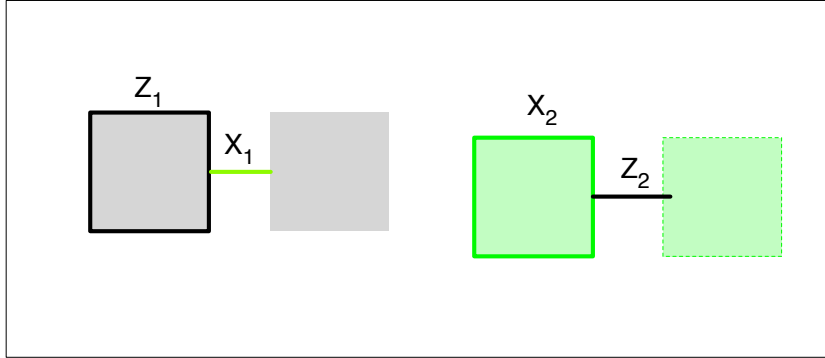


Figure 4.19: This surface encodes two qubits via a twin primal hole qubit and a twin dual hole qubit. The grey shaded holes represent primal holes and the green shaded holes represent dual holes.

This does not mean that we can ignore the $Z_1 Z_2$ altogether. As we see in the next section, we need to measure it to initialise a twin qubit in a certain state.

Via an analogous construction, we can define a twin dual hole qubit, from a pair of dual hole qubits. We use a repetition code in the $|\pm\rangle$ basis, i.e.

$$|+\rangle_L = |+\rangle|+\rangle \quad |-\rangle_L = |-\rangle|-\rangle \quad (4.12)$$

which has stabilizer generator $X_1 X_2$ and logical operators are $\bar{Z} = Z_1 Z_2$ and $\bar{X} = X_1$. In the fault-tolerant quantum computing scheme we will consider in the next part, we shall use twin primal qubits and twin dual qubits to encode our quantum bits. An example of a surface with both sorts of qubits is depicted in figure 4.19

4.2.9 A remark on terminology

Before we complete this section, a brief note on terminology. Primal and dual boundaries are not concepts widely used in homology theory – mathematicians usually consider only a single type of boundary (our primal boundary) on the surfaces they study – therefore the quantum researchers studying planar surface codes have invented their own terminology. Unfortunately there is no consistency in the literature and different authors use different terms.

For example, in the first paper on planar surface codes, Bravyi and call primal boundaries Z-boundaries and dual boundaries X-boundaries (since the logical Z/X operators can “start” and “finish” at these boundaries). Other authors call primal boundaries “smooth” and dual boundaries “rough” due to their appearance.

In the final section of this course, we will outline how planar surface codes with twin primal hole and twin dual hole qubits may be used to implement fault tolerant quantum computing.

5 Elements of fault-tolerant quantum computation with planar surface codes

5.1 Overview

For an architecture to achieve fault-tolerant quantum computation a number of criteria need to be satisfied. There are different ways such criteria can be specified, and different architectural approaches. Here we shall focus on an approach to fault tolerant quantum computing called the “magic state distillation” architecture. Such an architecture needs to include an encoding of quantum bits which:

- allows repeated *fault tolerant detection and correction* of errors
- allows the *distance* of the code to be varied and the *number of encoded qubits* to be *varied*, including
 - *preparing* encoded ancilla qubits in the states $|0\rangle$ and $|+\rangle$,
 - and *measuring* encoded qubits in the Z and X basis
- provides a way to perform *CNOT* and *Hadamard* on encoded qubits fault-tolerantly - i.e. protected by the error detection and correction process.
- allows one to “inject” or prepare qubits in other states, in particular so-called *magic states*.

Planar surface codes provide the basis of an architecture for fault tolerant quantum computation which satisfies all of these requirements. In this final part of the lecture course we will give an overview of how this is achieved.

The planar surface code architecture has a number of advantages with respect to rival fault-tolerance approaches. It has a very high error threshold (on the order of 1% error rate for standard error models), it has a planar structure, which is formed of repeated modules each with identical structure, and no “long-distance” quantum gates are required - every quantum gate need be implemented only between spatially neighbouring qubits. These features make it arguably the leading architecture for realising fault tolerant quantum computation at the present time. The principal disadvantage of this architecture is the high overhead (the number of physical qubits needed for every logical qubit).

Due to time constraints, we will not be able to cover, in detail, all aspects of this model, but will focus on certain key ideas - state injection, repeated error correction, and code deformation. For a detailed and clearly written introduction to this architecture I recommend, Austin Fowler *et al*, *Surface codes: Towards practical large-scale quantum computation*, <http://arxiv.org/abs/1208.0928>.

5.2 Achieving universal quantum computation

5.2.1 State injection

Fault tolerant architectures are usually limited in the number of quantum gates they can fault tolerantly support. Often, such gates are confined to the *Clifford group* of gates, containing gates such as (and generated by) H , the Pauli operators, CNOT and the single qubit phase gate $S = \text{diag}(1, i)$.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad (5.1)$$

Fault tolerant encoded qubit preparation is usually limited to a small family of states ($|0\rangle$ and $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$) and small family of measurements X , Z basis.

The above operations all belong to the set of stabilizer operations (since they can be described within the stabilizer formalism). Stabilizer operations on their own do not achieve universal quantum computation - indeed it is possible to efficiently simulate them classically (the Gottesman-Knill theorem).

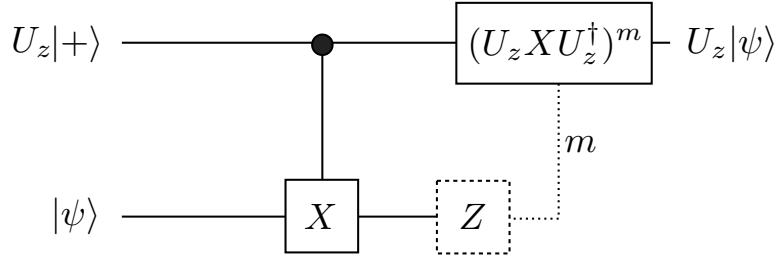
Remarkably, if the Clifford group is augmented by *any* single extra unitary quantum gate and its inverse, it becomes universal. Usually the most

convenient gate to choose is the so-called T -gate (or “ $\pi/8$ ” gate)

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \quad (5.2)$$

Note that $T^2 = S$ and that $T^\dagger = TSZ$. The addition of the T -gate promotes the stabilizer operations to universal quantum computation.

One method of achieving a T -gate is via state injection, where the gate is achieved via a combination of ancilla preparation and stabilizer operations implements the gate. Consider the ancilla state $U_z|+\rangle$ where U is any single qubit unitary which commutes with Z . This state can be “injected” into a circuit which implements U via the following circuit:



where, here, the dotted box represents a measurement of observable Z , with the output of the measurement $(-1)^m$ encoded in bit value m . The “correction” operator $(U_z X U_z^\dagger)$ must be applied if $m = 1$. If we only have access to Clifford group unitaries, to make this deterministic $(U_z X U_z^\dagger)$ must itself be in the Clifford group. This property is satisfied only in special cases, but, in particularis satisfied by T , due to the identity

$$T X T^\dagger = e^{-i\pi/4} Y S \quad (5.3)$$

where global phases can be ignored. Thus, provided the corrections Y and S , which are both Clifford gates, can be implemented, we can implement T deterministically via injection of the state $T|+\rangle$.

In the surface code computation that we introduce below, we will *not* have a full set of Clifford group gates, and not be able to directly implement the S -gate directly. Fortunately, since $S X S^\dagger = Y$, S can itself also be achieved with state injection. Notice that the state injection circuit only requires a CNOT, a measurement in the Z basis and the implementation of the correction. The state $S|+\rangle$ can be injected to achieve the desired S gates.

5.2.2 Magic state distillation

The state injection procedure enables universal quantum computation with this limited set of gates, provided $S|+\rangle$ and $T|+\rangle$ can be prepared. Unfortunately, the preparation of these states is not within our set of fault tolerant operations. The process will lead to the preparation of noisy approximations to these states.

Fortunately, a process called *magic state injection* can fix this. It is a distillation procedure which distills states very close to $T|+\rangle$ and $S|+\rangle$ given noisy copies of such states. Furthermore it requires only Clifford gates for its implementation (and does not need the S gate). It can therefore be implemented fault tolerantly with the gate set available here.

We do not have time to describe magic state distillation in detail here, but there is a detailed description of it in the paper which introduces it, Bravyi and Kitaev, *Universal Quantum Computation with ideal Clifford gates and noisy ancillas*, <http://arxiv.org/abs/quant-ph/0403025>.

Magic state distillation is very robust to noise, and thus can tolerate noise far higher than fault tolerant quantum computing. Thus, if we are above the fault tolerance threshold, magic state distillation will succeed.

5.3 Fault tolerance

5.3.1 Repeated error correction with error-prone measurements

How much noise can be tolerated? We have seen that the code error threshold for the planar surface code is approximately 11%. This computation, however, assumed that measurements were error free – a particularly strong assumption when we recall that these measurements are projecting into the code space, and effectively “making” the code. A more pragmatic threshold analysis – which would demonstrate true fault tolerance – would include errors in all components. In particular considering errors in the stabilizer generator measurements.

Remarkably, the surface codes can tolerate errors in those measurements in almost exactly the same way that they tolerate errors on the qubits. Here we will not consider a detailed microscopic error model, but a simple one. We assume that each measurement, independently with probability p , returns the opposite value to the correct outcome - e.g. if it should have returned $+1/0$ - it returned $-1/1$. We assume that these errors are “memoryless” - e.g.

that the presence of such an error at time t does not change the probability of such an error at time $t + 1$.

There is a simple way to combat such errors, repeated stabilizer measurement. One would want to repeatedly make stabilizer measurements in any case, to correct errors before they build up too much, so this is easily incorporated. The most naive way to use repeated measurements, would be to then consider the majority vote of such measurements - but this would not be effective if the “correct” error syndrome changed between these measurements, which you might expect if the measurement rate was of similar magnitude to the error rate. Surface codes provide a more elegant and more effective way.

5.3.2 Three-dimensional homology from error-prone measurements

Consider the two-dimensional surface - and for now, let us just consider vertex stabilizer measurements. Represent the outcomes of these measurements at time t as a 2-dimensional surface. Now consider the outcomes of the next round of measurements, at time $(t + 1)$. These *also* represent a 2-dimensional surface. Now consider a 2-D surface representing the *outcome-wise difference* between these rounds - place a 0 on every vertex where the outcome is the *same* as the previous measurement and a 1 when the outcome is *different*.

We now repeat this for many measurement rounds – for each round creating a surface representing the *differences* with respect to the previous round – and stack these up into a three-dimensional structure.

Let us simplify our mental image of this structure by assuming 2-D planar surface codes. The structure then has the form of a three-dimensional cubic lattice. On every 2-D plane in this structure, the vertices represent the *difference* between the outcomes of the stabilizer at time t and time $(t - 1)$.

Now consider the effect of different kinds of errors on this 3-D lattice. A single *new* qubit Z error at time t will trigger the adjacent vertex operators to flip - these operators will change their value at this point in time, and thus this will be recorded as a pair of 1 values on the associated vertices of the plane associated with time t in the 3-D lattice. We shall assume that the error is not corrected, and thus vertices on other planes for other time-slices do not record this.

What is the effect of a faulty measurement? Consider the case first where the measurement should outcome 0 at each step, but reports a faulty outcome at one time-step e.g. 0,0,1,0,0,0 – the corresponding difference is 2 1

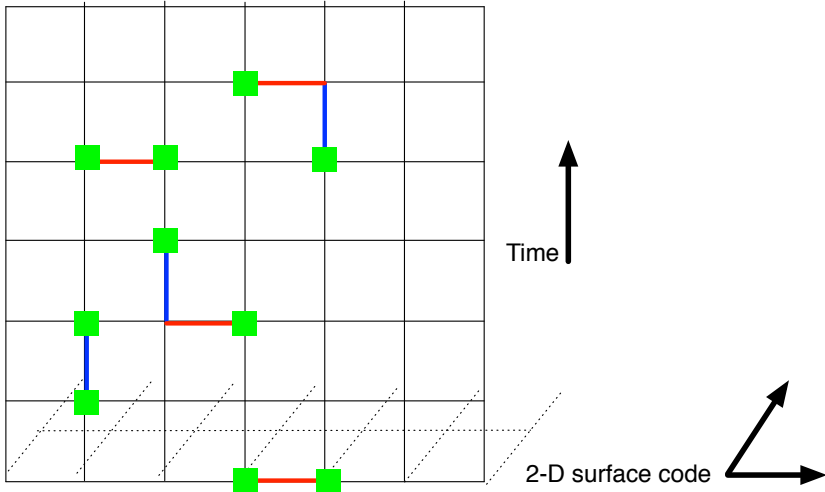


Figure 5.1: Homological detection and correction of errors including measurement errors. Each 2-D layer represents the *change* in the surface code syndrome – 1 if measurement outcome has changed, 0 if it is unchanged – at each measurement round. Measurement errors are depicted as blue vertical edges (1-cells), errors on qubits as red horizontal edges. The error syndrome (depicted using green squares) then lies on the boundary of the error-history 1-chain, and decoding etc. is fully analogous to the 2-d case for perfect measurements.

values on adjacent vertices in the time direction 0,0,1,1,0,0. The same would happen if the measurement should be recording 1, e.g. 1,1,0,1,1,1 – would be represented as 0,0,1,1,0,0,0. Thus fault measurements generate pairs of 1 values on our lattice in adjacent vertices in the time direction!

Qubit errors and measurement errors have almost identical effects on the 3-D lattice, qubit errors flip a single pair of vertices adjacent on a plane (so space-like neighbours), while measurement errors flip vertices on adjacent vertices in the time-like direction.

The resulting 3-dimensional structure can be understood (and error corrected!) in *homological terms*. Both kinds of errors, can be represented as a 1-chain in the 3-d lattice. 1-Cells which lie along a “space-like plane” represent physical errors, while 1-cells which lie in the “time-like” direction, represent measurement errors. The syndrome *differences* at each time step,

now correspond to the 0-chain boundary of this 1-chain.

Homologically, qubit errors and measurement errors are not distinguished. They can thus be corrected in the same way. A combination of qubit and measurement errors will create a string of 1-cells which is both space-like and time-like, but homologically, it is just a string like any other. The same error correction decoding strategies will work on the 3-D lattice. The optimal decoder will compute the most likely homology class of the error chain (since again, any correction chain homologically equivalent to the error chain will correct it) while the minimum weight perfect matching algorithm will find the most likely error chain.

For an independent noise-model, the threshold for minimum weight perfect matching on this model was found to be 2.9%. More detailed and physically justified error models also report thresholds with error rates on the order of 1%. The fault tolerance threshold for surface codes is thus the one of the highest reported (the only competing architecture with a similarly high threshold is the teleportation-based architecture of Knill <http://www.nature.com/nature/journal/v434/n7029/full/nature03350.html>).

5.4 Code deformation

The surface code can thus protect quantum information in a fault tolerant manner, provided measurements are made repeatedly. To demonstrate universal quantum computation, we also need the ability to initialise qubits, “inject” states and perform a limited set of quantum gates on them. *Code deformation* is the key technique to many of these.

The planar surface code architecture we are considering encodes quantum information in *twin primal hole qubits* and also *twin dual hole qubits* – as introduced in the previous section.

Recall that the distance of these codes depends on the separation of the codes and their circumference. In fault tolerant quantum computing it can be useful to *change* the *distance* of a qubit, increasing or decreasing its protection against error. For example, reducing a qubit’s code distance to 1 effectively unencodes it - allowing one to act directly on the qubit in an unencoded, but unprotected way. This is an important ingredient of *state injection* which we will meet later on. It is thus useful to be able to change a hole’s *size*.

It will also prove useful to be able to *move* holes around. This will allow us to implement some encoded logic gates in a robust manner.

Creating holes, enlarging them, deforming them and moving them are all examples of *code deformation*. In fault tolerant quantum computing, you should imagine every stabilizer generator being measured at every time step. For the time being, let us assume that any errors detected are immediately corrected. The stabilizer measurements (+ corrections) will therefore repeatedly project the system into the code space.

We can now change the code properties by *code deformation*. Code deformation amounts to manipulating the code by changing the set of stabilizer generators (which are still being repeatedly measured in between these changes). More specifically, we use a combination of techniques:

- Removing a generator from the generating set – i.e. “turning off” the measurement of that operator.
- Changing a generator’s observable – i.e. changing the corresponding measurement circuit.

5.4.1 Creating a hole

We shall focus here solely on primal holes, but everything that follows can be adapted to dual holes by interchanging X and Z operators, plaquette and vertex generators. Let us imagine that we start with a code defined by a single planar surface with a primal boundary. This surface has first Betti number $\beta_1 = 0$ and encodes no qubits - measurement of the stabilizer operators is thus projecting the into a unique pure stabilizer state. (As above we neglect errors at present, later we will show how they can be corrected). Let us call this state the *blank surface state*.

We wish to create a primal hole in this surface - and hence create a primal hole qubit. How do the stabilizers of the surface code with a primal hole qubit differ from the hole-less surface? They differ in three ways:

- The plaquette operators for plaquettes inside the primal hole are removed.
- The vertex operators for vertices in the interior of the hole are removed.
- The vertex operators which lie along the (primal) boundary of the hole are modified to lower weight operators, with the X operators which would lie inside the hole deleted from the observable.

Imagine, then that we start off in blank surface state and that measurement operators are being repeatedly measured. What happens if the stabilizer measurements are modified in precisely these ways - i.e. the plaquettes and vertices in the hole “switched off” and the vertices at the edges modified. What will happen to the state?

The repeated stabilizer measurements are now projecting the system into the desired primal hole surface code subspace. Thus the primal hole has been created. But what is the state of the encoded qubit? The encoded logical Z operator for this qubit consists of a product of Z ’s around the boundary of the hole. But this operator (representing the boundary of the hole) is equal to the product of the plaquette stabilizer operators which have been switched off. The product of the outcome of the measurement of these operators is thus equal to the measured eigenvalue of this boundary operator. Since this operator commutes with both the old set of stabilizer generators and the new set, it retains its eigenvalue after the generator measurements have changed. Thus the qubit is initialised in state $|0\rangle$ or $|1\rangle$ depending on this measured eigenvalue. In the error-free case considered here, this preparation will always create the $|0\rangle$ state.

To summarise, holes can be created by “turning off” stabilizer measurements for the plaquette and vertex operators in the interior, and modification of the plaquette operators around the boundary of the hole. The logical value of the qubit will correspond to the product of the outcomes of the plaquette operators in the hole interior immediately prior to being switched off.

5.4.2 Extending a primal hole

Now that we have created a hole, we may wish to enlarge it, e.g. to increase the qubit code distance. We can do this by, as above, modifying the (repeatedly measured) stabilizer generators.

To enlarge the hole by one plaquette, we must turn off that plaquette operator, and modify the vertex operators along the new boundary of the hole as appropriate.

The effect of this is to increase the size of the hole, while maintaining the logical encoded qubit associated with a hole.

One can verify that the remaining encoded logical operators for the qubit commute with the modified vertex operators and the removed plaquette, meaning they remain unchanged (and unmeasured by this process).

5.4.3 Contracting a primal hole

Now suppose we wish to make a hole smaller, e.g. to measure a logical qubit, or to remove it from the surface entirely. This can be achieved by the reverse steps of extending the hole. The plaquette operator which is to be removed from the whole is “switched back on”, and the vertex operators on the part of the hole boundary which has changed are updated accordingly.

It can be verified (similarly to above) that this makes the hole smaller without measuring or affecting the encoded logical qubit.

5.4.4 Moving a hole

Holes can now be moved via a combination of expansion and contraction. Moving the hole may change the qubits code distance but will not measure the qubit. Perhaps surprisingly, then, motion does not always leave the logical state of the qubit unchanged. We will see below, that moving holes relative to one another – in particular braiding holes around each other – can be used to implement certain encoded logical quantum gates.

5.4.5 Preparation of a double primal hole qubit

The fault tolerant scheme we consider encodes qubits as double primal holes and double dual holes. We can use code-deformation to create a double hole qubit and initialise it in the state $|0\rangle$. We switch off the plaquette and vertex generators in the interior of the holes and modify the vertex operators at the boundary. The distance of the qubit is set by the circumference of the holes and their separation.

Double dual hole qubits may be prepared in an analogous way (interchanging plaquette for vertex operators in the above description). If there is no error, their initial state will correspond to $|+\rangle$.

Since the only encoded qubits we will consider in this chapter are double primal hole qubits and double dual hole qubits, we will refer to them as *primal qubits* and *dual qubits* for short.

Being able to prepare ancillary qubits in the $|0\rangle$ state and the $|1\rangle$ state is an important component of fault tolerant quantum computation.

5.4.6 Measurement of a logical qubit

Fault tolerant measurement of qubits is performed in the reverse way to their creation. A primal qubit is measured in Z when its logical Z operator is measured. Switching on the plaquette and vertex stabilizers inside the holes, while modifying the vertex operators on the edge back to their “hole-less” configuration, removes the qubit, but also measures it in the Z basis. The measured eigenvalue will be the product of the “switched on” plaquettes which were previously in the interior of the hole. The product of plaquette outcomes from each hole should give the same outcome (recall our encoding is equivalent to a repetition code). If not, an error has been detected and that qubit measurement outcome lost.

5.4.7 A braided CNOT gate

Another important component of fault tolerant quantum computation is, of course, the execution of encoded logical gates. So far in this course, we have focussed on the error correcting properties of surface codes and not considered any logical gates.

Here, we shall see an example of a gate implemented entirely through code-deformation. A CNOT gate between a primal and a dual qubit is implemented by merely braiding one of the holes of the first qubit between the holes of the second qubit.

To see why this implements a CNOT, we should work in the so-called Heisenberg picture of quantum computation. When a unitary U acts on a quantum register it equivalently transforms a logical Pauli operator σ for that register via conjugation, i.e.

$$\sigma \rightarrow U\sigma U^\dagger \quad (5.4)$$

We can therefore describe the action of a gate via the way it transforms the logical operators. Let X_1 and X_2 , Z_1 and Z_2 be logical X and Z operators for qubits 1 and 2, and let qubit 1 be control and qubit 2 be target. The CNOT gate transforms these operators as follows:

$$\begin{aligned} X_1 &\rightarrow X_1 X_2 & Z_1 &\rightarrow Z_1 \\ X_2 &\rightarrow X_2 & Z_2 &\rightarrow Z_1 Z_2 \end{aligned} \quad (5.5)$$

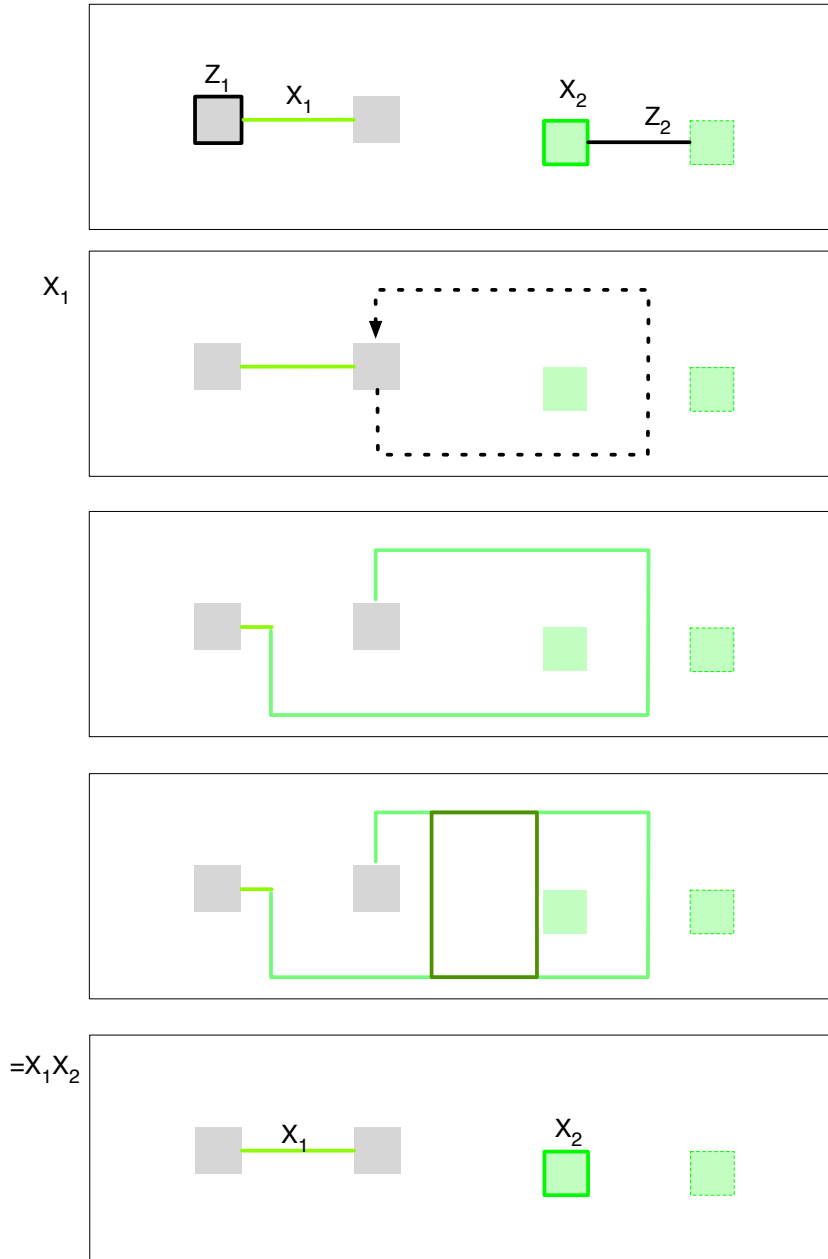


Figure 5.2: This braiding transformation (achieved via code deformation) implements a CNOT gate between a primal qubit and a dual qubit. The transformation is illustrated here for logical X_1 , once the braid has occurred this logical operator is now wrapped around and between the dual holes. This “wrapped” operator is homologically equivalent to X_1X_2 . A suitable stabilizer operator to “split” the “entangled” X-cochain into the “string” representing X_1 and the “loop” representing X_2 is illustrated as a darker green loop.

This transformation is demonstrated in figure 5.2. Braiding the primal hole between the two dual holes by code deformation is a continuous deformation which leaves the logical X_1 wrapped between the two dual holes. The operator is now homologically equivalent to the X_1X_2 operator - the transformation of a CNOT.

Similarly one can show that the Z_2 operator becomes similarly “entangled” by the braid, and transforms to the Z_1Z_2 operator, while the loop-like Z_1 and X_2 operators are left unchanged by the braid.

Performing quantum gates by braiding is a key method of topological quantum computing with anyons, and this illustrates that the holes in the surface code have anyonic statistics.

The braided CNOT is an elegant fault tolerant realisation of a quantum gate through code deformation. However, it has a disadvantage - it can only be applied between a primal and a dual qubit.

In fault tolerant quantum computing we’d like to be able to perform CNOT gates between any 2 qubits. Fortunately, this can be achieved in a relatively straight-forward manner. It requires some ingredients we haven’t seen yet - measurements and preparations of primal qubits in the $|\pm\rangle$ basis and dual qubits in the $|0\rangle, |1\rangle$ basis.

5.4.8 X-basis measurement on primal qubits

The braided CNOT demonstrated an element of anyon-like behaviour in the surface holes. If you are familiar with anyons you will know that, in addition to braiding, they can be fused and pairs of anyons created from the vacuum. The measurement of logical X for a primal qubit (a double primal hole qubit) is reminiscent (to me at least!) of anyon fusion.

Recall that the encoded logical X for this qubit consists of stringlike X -cochain operators stretching from one hole’s boundary to the other hole. We don’t want to measure that $d + 1$ qubit operator directly, as we wish for our measurements to remain low weight. If we could measure these $d + 1$ qubits individual in X their product would reveal the value of logical X . However such individual measurements would conflict with (since they anticommute with) the plaquette measurements on adjacent plaquettes, and be detected as errors.

The solution then is to “merge” or “fuse” the two holes, joining them together by disabling the plaquette measurements on the plaquettes between

them, and at the same time modifying the vertex measurements on the new boundary.

After this “merging” has been done, the qubits inside the hole are no longer entangled with the remaining code qubits, and error detection measurements are no longer being applied to them. These are precisely the qubits we wanted to measure individually in X . We can now do this, and the product of the outcomes of the individual X measurement will reveal the measured eigenvalue of the logical X operator we wished to measure.

The hole that is left over can then be shrunk and finally removed altogether using the standard code deformation methods described above.

This method appears to leave the qubits in the code vulnerable – how can they be protected when they are no longer entangled in the code. It is an illustration of the robustness of code deformation that errors on these qubits can still be detected via standard surface code error detection.

We need only consider Z errors since only these can affect the outcome of our logical X measurement. A Z error on one of the qubits which is individually measured will flip its outcome. If this qubit was still part of the code, the error would flip the outcomes adjacent vertex operators and be detected. After code deformation, these vertex operators no longer measure the qubits in question. However, we are measuring the qubits individually in X . Multiplying this outcome with the outcomes of the adjacent vertex operators, we can reconstruct the equivalent outcomes of full 4-qubit vertex operators. Thus Z errors on the qubits measured individually in X can still be corrected using the standard surface code technique.

Measurement in the Z basis on dual qubits can be achieved in an analogous manner.

5.4.9 Preparation of $|\pm\rangle$ on a primal qubit

When we create primal qubits by “switching off” plaquette measurements, as described above, they are prepared in the state $|0\rangle$. We can prepare logical state $|\pm\rangle$ by reversing the actions in the X measurement described in the previous section. We create a primal hole large enough to contain both of the holes in our primal qubits and the “tube” between them which will represent a logical X for the qubit. We prepare the qubits inside the “tube” in the $|+\rangle$ state. We then remove the “tube” by code deformation. The holes now have the form of a primal qubit and that qubit is in state $|+\rangle$.

The reason for this is that measured observable, the logical X commutes with all of the stabilizer measurements made, both before and after the tube is removed. Thus the system remains prepared in the $+1$ eigenstate of this operator - the logical $|+\rangle$. Similarly to measurement, errors in this process will be detected via the standard surface code error correction (which, remember, is going on repeatedly, throughout).

Again, the preparation of state $|0\rangle$ for a dual qubit achieved in an analogous manner.

5.4.10 State injection

We have seen how to fault tolerantly prepare $|0\rangle$ and $|+\rangle$. For state injection we need to prepare states of the form $T|+\rangle$ or $S|+\rangle$. We cannot achieve this in a fault tolerant way, but we know that magic state distillation will allow us to distill states arbitrarily close to these states, if noisy versions can be created, and encoded into the code.

To perform state injection, we thus “turn off” part of the protection of the qubit. Recall that minimum weight of logical X for these qubits is equal to the distance between the holes, plus one. If we bring these holes together, so the edges of the holes touch, the distance is then 1. (Note this is different to merging the holes - see figure 5.4.

In the Heisenberg picture, the effect of T is to transform the X operator to TXT^\dagger . Thus, if the logical X is reduced to a single qubit X , it can be transformed to TXT^\dagger by a single qubit T gate.

Thus state injection proceeds as follows – create two “touching” holes, apply T to the qubit at the place where the holes “touch”, and then increase the code distance, by moving the holes apart, to protect this qubit. This operation injects a noisy copy of $T|+\rangle$ into the circuit. Once the qubit code distance has been increased to the appropriate level, this qubit is protected by the code, but it may have undergone errors while the distance was low, and hence magic state distillation must be used to create a low noise state suitable for use in generating a low-error-rate T -gate.

5.4.11 Teleportation from a primal to a dual qubit

We have seen how a CNOT gate can be implemented between a primal and a dual qubit by braided code deformation. We need, however, to be able to

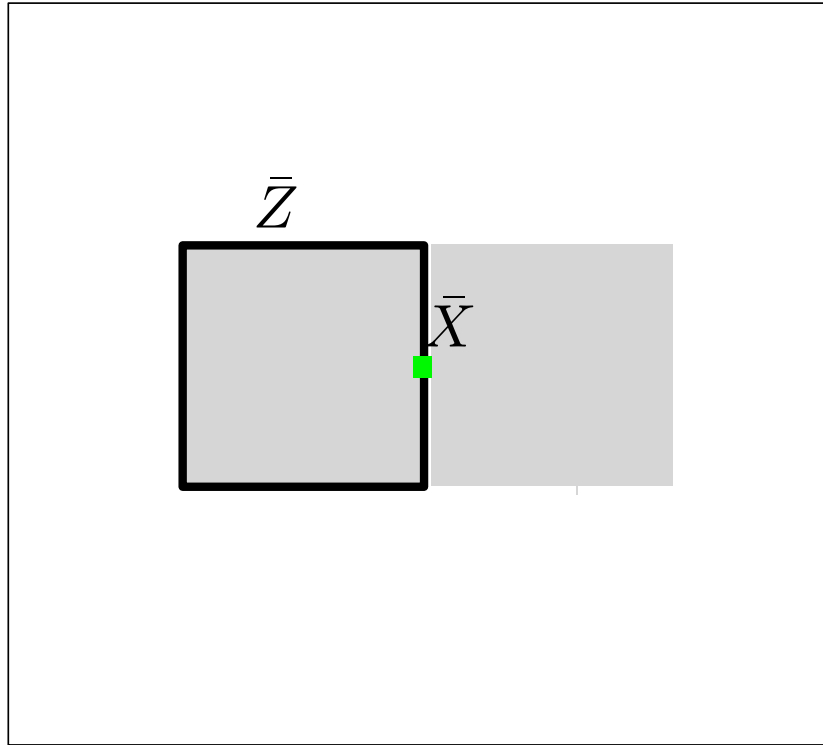
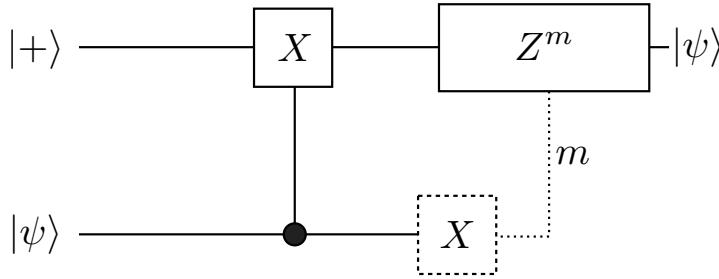


Figure 5.3: The minimum weight of the logical X for a primal qubit is the distance (plus one) between the holes. If the holes are made to touch, the weight is just 1. Then we can apply a logical S or T gate to a qubit prepared in $|+\rangle$ to “inject” the state $T|+\rangle$. The protection can then be restored to the qubit by moving the holes apart by code deformation. This process is not fault tolerant, so magic state distillation is then required to remove the errors from the injected states.

implement CNOTs between arbitrary qubits, so between primal and primal qubits. A simple solution is to use one-bit teleportation.

One-bit teleportation is a simple, but powerful protocol (which can also be used, for example, to derive the state-injection circuit above, and to derive measurement-based quantum computing on cluster states). The following circuit “teleports” the logical qubit from the upper physical qubit to the lower physical qubit.



where the controlled gate (control-X) is a CNOT, and where the Z correction is implemented if the outcome of the X-measurement is -1 .

We see that this can be implemented using CNOT, $|+\rangle$ state preparation, logical Z operators and X measurement, all ingredients we have introduced above. We can thus implement a CNOT between two primal (or two dual) qubits by first “teleporting” one qubit onto an ancilliary dual qubit, performing the CNOT, and then teleporting back.

5.5 Completing the set of gates

5.5.1 Hadamard Gate

We have now seen all ingredients of fault tolerant quantum computation on the surface code, except one - the Hadamard gate. We did not need this gate to achieve the operations above, but we need it to have a universal set of quantum gates. Fortunately, Hadamard can be achieved in a straightforward way.

We have seen a clear relationship between X and Z operators in the surface codes, representing Z operators on 1-chains and X operators on 1-cochains. We use this to our advantage here. Consider a planar surface code with a single double primal hole qubit. Apply Hadamard gates to every single qubit. We now have a code which looks very similar to the code we had

before, but now where had X 's we have Z s and vice versa. For example, in the stabilizer generators, we have X operators on the boundaries of plaquettes and Z operators on the co-boundary of vertices. Our logical Z has transformed into a cycle of X 's.

We can, however, manipulate the code back to a standard surface code in a simple way - recall that we can map from primal to dual lattice by shifting the lattice down and left by half a cell. If we move our qubits in this way, we shift our new Z stabilizers back onto the boundaries of plaquettes, and our X stabilizers back onto the coboundaries of vertices.

This shift has also transformed our primal boundary to a dual boundary, and thus our primal holes have become dual holes. The former logical Z has transformed to a dual qubit logical X , while the logical X has transformed to a logical Z .

This combination of Hadamard plus half-cell shift completes the logical Hadamard gate, together with the conversion of the qubit from primal to dual.

If we wish to implement this on a surface with many qubits, we would implement the Hadamard on all the logical qubits - not what we'd want to do. However, areas of surface code containing a single qubit can be "detached" from the rest via code deformation. Thus the sequence to achieve a single qubit Hadamard is somewhat - detach qubit's section of lattice, perform local Hadamards, shift qubits by half a lattice cell, reattach qubit into full lattice.

At the end of this, the primal qubit has been converted to a dual qubit - and a final teleportation step would be needed to restore the qubit back to primal form.

This method of implementing the Hadamard gate works therefore, but is cumbersome. A number of improved approaches have been suggested, in particular, in A. Fowler <http://arxiv.org/abs/1202.2639> presents an elegant approach to achieving the Hadamard gate is presented almost entirely in figures.

5.6 Summary and outlook

We have seen that the planar surface code with dual hole qubits provides all the ingredients needed for fault tolerant quantum computation. Dual hole qubits provide robust storage of qubits protected by the surface code. Error correction robust to measurement errors is achieved homologically, using the

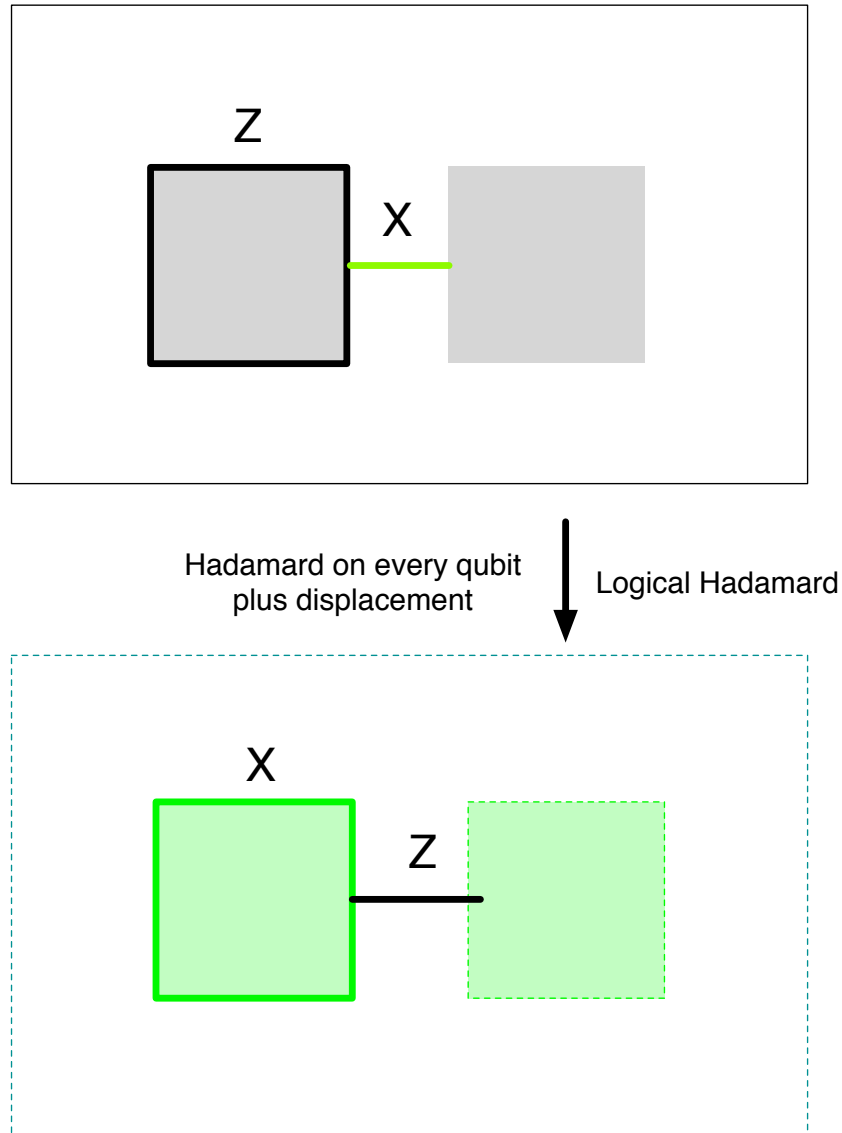


Figure 5.4: A Hadamard is implemented on a surface code with a single double hole qubit via Hadamards on every qubit, plus a displacement left and down by half a cell. This also changes the qubit-type from primal to dual.

same methods as in the perfect-measurement case. Code deformation provides a suite of tools. Qubits can be fault tolerantly prepared and measured in Z and X eigenstates, magic state qubits can be injected and then protected, and CNOT gates can be achieved through braiding.

This architecture has a number of significant advantages for experimental realisation. Its high fault tolerance threshold for errors is within reach of a number of experimental settings. Its modular architecture on a 2-dimensional surface well placed for scaled up manufacture. It might seem like the ideal way to build a fault tolerant quantum computer, but, there are disadvantages too. The overhead in physical qubits to logical qubits is high. To increase the code distance linearly, the number of qubits must be increased quadratically. Thus formally, the rate of encoded qubits to physical qubits goes to zero as the distance goes to infinity. In addition, magic state distillation itself comes with a high overhead cost. Putting this all together Raussendorf, Harrington and Goyal estimated an overhead of from 10^5 - 10^{11} physical qubits to logical qubits. It remains an open research question to determine whether the significant advantages of topological codes for fault tolerant quantum computation can be maintained while significantly reducing this overhead, and this may determine whether the planar surface code remains the front running architecture for scalable and fault-tolerant quantum computing.