Embedded $D(\mathbb{Z}_2)$ Surface Codes
&
A decoder for imperfectly measured Planar Code

János R. Winkler
Universität Basel
(Dated: Friday, December 18, 2015)

New topological error correction codes are defined using embedding techniques, where $D(\mathbb{Z}_2)$ Surface Codes serve as the building blocks. The properties of these new codes are thoroughly investigated and we show that they do not all share the same anyon model.

We developed a decoder under the assumption that noiseless measurement of the stabilizers is not possible. This decoder is based on the MWPM algorithm and is designed to fix errors on a Wen style planar code. The decoder was numerically tested and a threshold was obtained for two rounds of measurements, where measurement errors occurred with a probability of $p_m = 1\%$. The thresholds for the decoder were found in the range of $\approx 6\%$ for independent noise as well as $\approx 9\%$ for depolarized noise. Furthermore, we show that the decoder is significantly better compared to regular MWPM for very low error rates.

Contents

I. INTRODUCTION

It does not matter how good we are at storing any given quantum state. Noise will occur and there is no way to completely prevent it. Once we accept the fact that decoherence will happen we can think about what causes the noise and how we can remove its effects.

Discovering the best codes to store quantum information and the best methods to fix the errors; that is the goal of quantum error correction, a crucial feature in order to realise a quantum computer. Quantum information is stored on a so-called quantum error correction code. We use measurements to detect a trace of the errors implied by nature of quantum systems and try to come up with operations to correct them. Finding the right recovery operations is what we call decoding. This thesis is arranged to tackle both parts of this problem.

The first part introduces methods to construct surface codes embedded into other surface codes. The idea of doing such was first introduced in [1], where the color code [2] was derived using embedding techniques. Here it is used to build up new, unknown codes. We explore their properties with a special focus on how the new anyon model is related to one of the two codes used to define it. We find, that surprisingly different anyon models are found, even tough the different $D(\mathbb{Z}_2)$ surface codes all share a common anyon model.

The second part covers the decoding side of quantum error correction. Not even the most advanced code one could imagine is of any use without a decoder. The decoder is a piece of classical software implemented to use the measurement outcome to compute the best way of fixing errors on a code. These errors happen according to different error models. In this thesis we present the derivation of two versions of a decoder and the numerically obtained results. We do this for different error models and assuming noisy measurement.

II. PRELIMINARIES

A. Stabilizer Formalism

For classical computers the usual approach is to use a physical system where errors happen with a probability $p \leq 50\%$ and then store (N) physical bits together as a logical bit. We assume that the majority voting on all the physical bits yields the logical value. This assumption is good since the probability of a logical bit flip is given by the Chernoff bound [3], where N is the number of bits and p the physical noise acting on the system:

$$P_{\text{Bit Flip}} \leq \exp\left(\frac{-1}{2(1-p)} \cdot N(p-\frac{1}{2})^2\right)$$

We can reduce the chance of a bit flip happening as much as we want by increasing the amount of physical bits N per logical one.

We want to find something similar for qubits. In order to do such we have to keep in mind that there are some differences between the classical and the quantum case. First of all, the classical approach would require copying an unknown quantum state which can not be done (according to the No-cloning theorem [4],[5]).
The second difference is that for two level quantum systems there are infinitely many possible errors, whereas classically there is just one. But every error that affects our qubit can be expressed as a combination of Pauli matrices [3]. The two remaining errors are the bit flip (as before) but and the phase flip

$$\begin{aligned} \text{Bit flip: } \sigma_x |1\rangle &= |0\rangle \\ \text{Phase flip: } \sigma_z |1\rangle &= -|1\rangle \end{aligned} \tag{1}$$

These differences force us to use a more evolved method. The one will be used here is the Stabilizer Formalism.
As in the classical case the goal is to store logical qubits in multiple physical ones. Lets call the amount of logical qubits k and the the amount of physical ones n.
A stabilizer S_j is a set of commuting hermitian operators [6]

$$[S_i, S_j] = 0 \ \forall i, j$$

We select $n-k$ (independent) operators from the set of all n qubit tensor product of Paulis [3]:

$$\begin{aligned} \{S_j\} &= \sigma^1_{a_1}\sigma^2_{a_2}\sigma^3_{a_3}...\sigma^n_{a_n} \\ a_j &\in \{0, x, y, z\}, \sigma_0 = \mathbb{1} \end{aligned}$$

This is a valid set because two different Paulis anticommute if acting on the same qubit, but commute if acting on different ones.
Therefore its possible to construct many-body operators, called the stabilizers, using the commuting subset of Paulis.
With the same assumption as in the classical case, that less errors are more likely than many errors, the stabilizers identify the qubits which are affected by bit and/or phase flips, by using a changed basis state as the indicator that something happened.
± 1 are the eigenvalues of the Paulis because they are hermitian and square to the identity it follows that the stabilizers [5] are observables with the same possible outcomes.
Because of their eigenvalues the $\{S_j\}$ correspond to projective measurements which are called syndromes [3].
The Pauli matrices acting on n qubits corresponds to a set of 2^n eigenstates which can be clearly identified using the behaviour of all the stabilizers [6].
The stabilized space, the so called stabilizer space, is the space of spanned by all the states which are in the $+1$ eigenspace of all $\{S_j\}$:

$$S_j |\psi\rangle = +1 \cdot |\psi\rangle, \ \forall \ j$$

The stabilizer space is used to store k logical qubits. It is possible to do so because we selected $n-k$ independent stabilizers. So, the dimension of the stabilizer space is 2^k.

$$\begin{aligned} \#\text{ logical qubits} = &\ \#\text{ physical qubits} \\ &- \#\text{ independent stabilizers} \end{aligned} \tag{2}$$

If the system is untroubled by nature (no errors occurred), this means all measurement outcomes are $+1$ and the state of the system is within the stabilizer space. However if errors have happened some outcomes will be -1. So the goal is now to use this trace of errors to correct them.
But there are also errors which leave no trace in the system, the so-called logical Pauli errors. They do not force the system out of the eigenspace. This is possible since those errors commute with all the stabilizers [3]. These logical Pauli errors change the state of the logical qubit so we can think of them as logical operators in respect to the code they are involved with, denoted with X and Z. Logical operators do not only act on one physical qubit. The minimal amount of qubits which, if affected, change the logical state of the system is called the code distance d.

For any stabilizer based code the logical operators are only valid if they satisfy the following conditions [3]:

- Logical Pauli operators must commute with all stabilizers.
- Logical Pauli operators for the same logical qubit anticommute.
- Logical Pauli operators for different logical qubits commute.

The logical Paulis are not uniquely defined

B. Surface Codes

Surface codes are one possible application of stabilizer formalism introduced above and an important class of error correction codes. These are specific examples we use in the following sections.

1. The Planar Code

The Planar code is defined on a $L \times L$ sized square lattice where one qubit lives on each edge [3]. Two kinds of stabilizers are defined around each plaquette and each vertex, which we will call B_p and A_v. For both stabilizers there is an additional altered version which is necessary to handle the boundary.
The top and bottom boundary is called rough. The plaquette operators B_p are not complete there and symmetrically there is the smooth boundary on left and right side where the vertex operators A_v are incomplete:

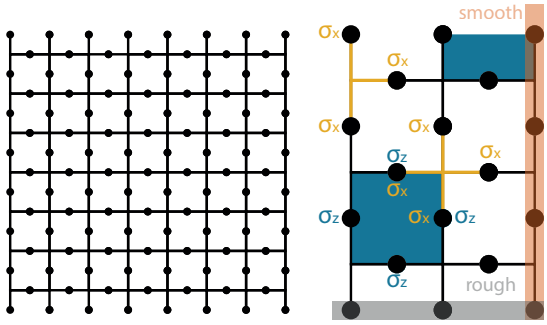

FIG. 1: Planar code with $L = 8$ and examples for both bulk/edge plaquette B_p and vertex A_v stabilizers

$$\begin{aligned} B_p &= \prod_{i \in \text{ plaquette}} \sigma_z^i \\ A_v &= \prod_{i \in \text{ vertex}} \sigma_x^i \end{aligned} \tag{3}$$

Since the Pauli matrices square to the identify stabilizers of one kind commute with each other

$$[B_p, B_p'] = [A_v, A_v'] = 0$$

Two different stabilizers can only share two or zero qubits (as shown in fig. 1), thus using $\left[\sigma_x^i, \sigma_z^j\right] = \left[\sigma_x^i \cdot \sigma_x^j, \sigma_z^i \cdot \sigma_z^j\right] = 0$ we see:

$$[B_p, A_v] = 0$$

Let's think about eq. 2 for the planar code. It is possible to count all physical qubits. Starting on the left side, there are $L + (L-1) + L + (L-1) + ... + L$ qubits where the L-term occurs L times and the other one $L-1$ times.

The same argument can be used to count the stabilzers there are $2 \cdot L(L-1)$ stabilizers, for which one half are B_p's and the half A_v's. All of them are independent [3].
The amount of logical qubits is obtained using eq. 2:

$$L^2 + (L-1)^2 - [2 \cdot L(L-1)] = 1$$

So it is possible to store one logical qubit on a planar code. As usual, this qubit is stored in the $+1$-eigenspace of all stabilizers.

The states $|0\rangle_L$ and $|1\rangle_L$ are associated with the eigenspace of the plaquette operators. A B_p is only in a $+1$-eigenstate if it has been affected by an even amount of σ_x's. This requirement stretched out to the whole code is met by loops of ones around plaquettes and an even amount strings of ones from left to right. All these allowed states can be constructed by applying A_v's, which always result in an even amount of ones if applied to a state polarized to hold only zeros. This suggests that we sort all the B_p-eigenstates into two classes. All

of the states mentioned above are in the first class. The superposition of all states of the first class defines the logical zero $|0\rangle_L$.

The second class is the same except it holds all states with an odd amount of strings from left to right. States of this second class can not be created by applying A_v's to the zero polarized state. Its only possible to generate such states if A_v's affect states that already have an odd amount of strings. The superposition of all states of the second class defines the logical one $|1\rangle_L$.

The logical states of the planar code are therefore:

$|0\rangle_L$ = The superpostion of all states of class one:

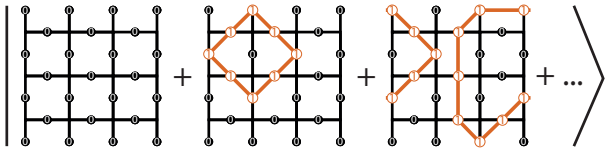

FIG. 2: $|0\rangle_L$ has an even number of strings from left to right

$|1\rangle_L$ = The superpostion of all states of class two.

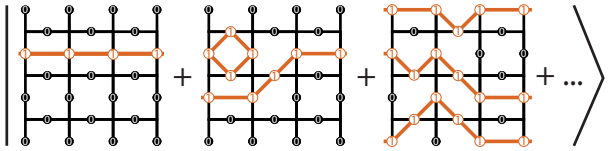

FIG. 3: $|1\rangle_L$ has an odd number of strings from left to right

Note that it is not necessary to distinguish where the strings end. Any vertical line is fine since every configuration (of ones) of it could be moved to any wanted line by applying B_p's. From now on the left edge will always be considered as the position where the strings are counted.

The logical operators are supposed to have the same effect on the whole code as if it was a single qubit affected by the Pauli matrices as mentioned in eq. 1. To introduce the logical Z we use the fact that the logical states are identified by the number of stings ending on the left edge. Applying a σ_z on every left edge qubit yields ± 1 depending on if a string ends or not. The relation to the plaquette operators is trivial and it commutes with the vertex operators as well since there are always zero or two shared qubits.

By replacing vertex with plaquette operators and the stings of ones with strings of minuses, and likewise, the zeros with pluses, the same argument can be used to define the logical $|+\rangle_L$ and $|-\rangle_L$ states. The logical X is simply a line of σ_x's acting on every qubit in the top (or any other horizontal) line. Both logical operators have characteristics similar to the Pauli matrices: they anticommute, are hermitian and unitary.

Since the Pauli matrices together with the identity form a complete basis for the 2×2 matrices, everything nature can possibly throw at the physical qubits can be expressed using this basis. The identity has obviously no effect on the qubits and one of the Paulis can always be expressed using the other two. Therefore it is only necessary to care about effects the two Paulis could have on our states. Considering that we defined the stabilizers using σ_x's and σ_z's we study only their effect on the code and handle the σ_y's as if both a σ_x and a σ_z happen. Every qubit is affected by two σ_x's from two vertex operators and two σ_z's from the plaquette operators.

Error	B_p	A_v
σ_x	anticommute	commute
σ_y	anticommute	anticommute
σ_z	commute	anticommute

For all cases where the stabilizer is anticommuting its eigenvalue is changed to -1 and is, therefore, no longer part of the eigenspace. Such a stabilizer can be thought of as holding a quasi-particle. Every qubit always contributes two times to one sort of stabilizer. That is why we can consider errors as pair creation. Pair annihilation happens vice versa if by undoing an error the eigenvalues of two connected stabilizers are changed back to $+1$. Creation/annihilation on stabilizers with different eigenvalues is used to move the particles across the code always connecting them by a string of errors. These quasi-particles are called anyons and there are two different kinds of them living on a planar code:

- A pair of m-anyons is created by σ_x's and lives on plaquettes. Strings of m's end on left and right edges where the boundary is smooth and the operator complete.

- A pair of e-anyons is created by σ_z's and lives on vertices. Vice versa strings of e's end on the top and bottom edge.

On each of the edges exists a so-called edge-anyon. Those are delocalized across the whole edge.

The different endpoints of the strings allow us to think of an alternative way to define the basis states. There can only be one or zero anyons on an edge because the strings are counted using modulo 2 because anyons are their own antiparticle. The amount of m-anyons on the left (or right) edge is associated with $|0\rangle_L$ and $|1\rangle_L$. And the amount of e-anyons on the top (or bottom) edge determines if the code is in a $|+\rangle_L$ or $|-\rangle_L$ state. The logical operator's job is now to change the anyon occupancy of the corresponding edges.

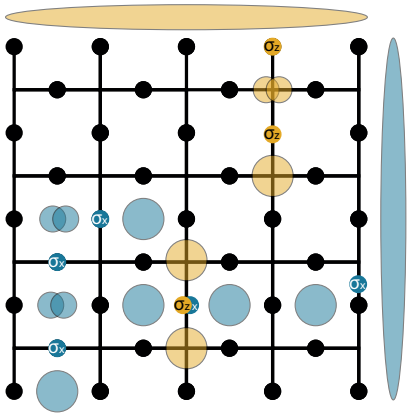

FIG. 4: A 5×5 planar code affected by both possible kind of errors with edge anyons living on top and right edge

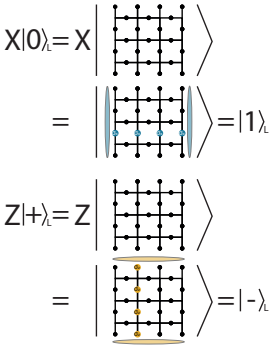

FIG. 5: The logical operators for the planar code

Therefore:

- The logical X creates/annihilates edge pairs on left and right edges
- The logical Z creates/annihilates edge pairs on top and bottom edges

2. The Toric Code

The Toric Code is a Planar Code with periodic boundary conditions. The Planar Code in the section above is defined with a smooth and rough boundary but instead of those we could wrap it around a torus to get rid of all boundaries. Now there are no more incomplete stabilizers so all of them belong to the four-body operator class. There are no edge anyons either, which is why the Toric Code is translationally invariant.

We can count the dimension of the Toric Code by an index L corresponding to the number the plaquettes in each direction, in total there are L^2 plaquettes on a Toric Code. Each consists of four physical qubits but every physical qubit belongs to two plaquette operators. Therefore, there are

$$n = L^2 \cdot 4 \cdot \frac{1}{2} = 2L^2$$

physical qubits on the code. One of the plaquette operators is dependent. This can be seen by applying all B_p's but one, called $\tilde{B}_p$. It has the same effect overall on the code as if it would have been only affected by the non-applied stabilizer. The same argument can be used for the vertex operators. So in total there are two dependent stabilizers. Using eq. 2 we find that it is possible to store:

$$\#\text{ logical qubits } = 2L^2 - \left[\left(L^2-1\right) + \left(L^2-1\right)\right] = 2$$

logical qubits on the Toric Code.

The logical operators used for the Planar Code do not work any more since it is no longer possible to count edge anyons. A set of logical operators according to the usual conditions introduced in section II A can be defined relative to the Planar Code. Again, some strings across the whole surface, here the torus, do the job but with the difference that this time it is necessary for the strings to align with the end/starpoint. As seen above, there is room to store two logical qubits on the torus. Consequently we define four logical operators - two for each qubit.

X_1 and Z_2 are therefore horizontal, gapless (and X_2 and Z_1 vertical) strings.

Regarding all other matters the Toric Code has the same properties as the Planar Code.

3. The Wen Plaquette Model

As before the Wen plaquette Model is defined on a square lattice, it has periodic boundary conditions the same as the Toric Code. The difference between the two is that for the Wen case the qubits are placed on the vertices and not on the edges of the code. Once again the dimension of the code is given by an index L used to count the the number of plaquettes along one line. For even L it is possible to bicolour the plaquettes. This code has only one sort of stabilizer called W_p:

$$W_p = \sigma_z^1 \cdot \sigma_x^2 \cdot \sigma_z^3 \cdot \sigma_x^4$$

All such plaquette operators commute because they can share no qubit or one of

- an edge:

$$\begin{aligned}[W_{p_1}, W_{p_2}] &= \left[\sigma_x^{(1)}, \sigma_z^{(2)}\right] + \left[\sigma_z^{(2)}, \sigma_x^{(1)}\right] \\ &= i \cdot \sigma_y - i \cdot \sigma_y = 0\end{aligned}$$

- a corner: then they share two times a σ_x or a σ_z. In both cases the operators commute.

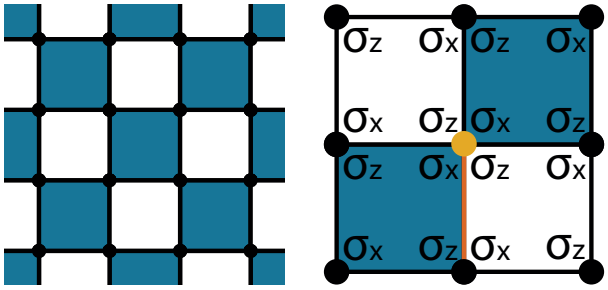

FIG. 6: An example for the periodic boundary conditions of a bicoloured Wen Code with $L = 4$ and its stabilizers which share either a edge or a corner.

The amount of logical qubits which can be stored on such a code depends on L. Once again using eq. 2 and keeping in mind how the product over all W_p's acts we find:

- L even: space for 2 logical qubits
- L odd: space for 1 logical qubit

From now on let's only consider the bicoloured case, which is unitary equivalent to the Toric Code. This can be proven by assigning one class of plaquettes to the plaquettes of the Toric Code and the other one to the vertices of it. This implies that on the Wen code the same anyons exist as on the toric/Planar Code. Two logical qubits as well as the equivalence to the Toric Code strongly suggest four logical operators as well. Phase/bit flips for both. This can be achieved by a set of alternating σ_x/σ_z on horizontal and vertical strings across the whole code:

$$X_1 = \sigma_x \cdot \sigma_z \cdot ... \cdot \sigma_z, \quad Z_1 = \begin{matrix} \sigma_z \\ \sigma_x \\ \cdot \\ ... \\ \cdot \\ \sigma_x \end{matrix}$$

$$X_2 = \begin{matrix} \sigma_x \\ \sigma_z \\ \cdot \\ ... \\ \cdot \\ \sigma_z \end{matrix}, \quad Z_2 = \sigma_z \cdot \sigma_x \cdot ... \cdot \sigma_x$$

We now introduce a second adaptation of the Wen code, this version is supposed to have the boundary conditions known from the Planar Code. We do the same we did to get from the Planar Code to the Toric Code but in reverse. We remove all stabilizers that stretch over the boundary and add two-body stabilizers to the edges to mimic the known rough/smooth boundary. Like the codes before this one inherits the anyon configuration from the Planar Code. One sort of anyon exists on each type of plaquette.

Note since this code is not defined on a torus the bicolouration is no longer needed to provide the same colour for a start/end plaquette. Therefore it is possible to bicolour the plaquettes for odd L as well. This code

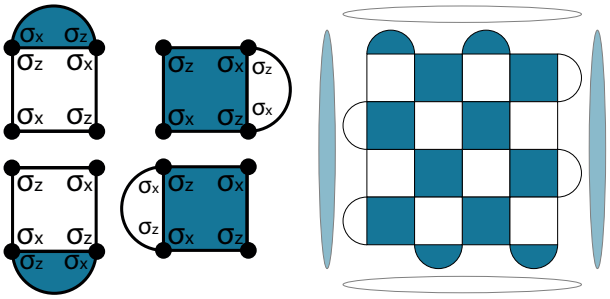

FIG. 7: The four different two-body stabilizers necessary to introduce planar boundary conditions and consequently the delocalized edge anyons

now stores only one logical qubit. This corresponds to two logical operators. We reuse the two, which affect the first stored qubit for the periodic case, X_1 and Z_1. These commute as required.

C. $D(\mathbb{Z}_2)$ anyon model

All the surface codes introduced above inherited the anyon configuration of the Planar Code. It is specified by their

- Fusion rules:

$$e \times e = 1, \quad m \times m = 1, \quad \epsilon \times \epsilon = 1,$$
$$e \times m = \epsilon, \quad \epsilon \times e = m, \quad \epsilon \times m = e$$

- Braiding rules:

$$R_{ee} = R_{mm} = 1, \quad R_{\epsilon\epsilon} = -1,$$
$$R_{em}R_{me} = R_{e\epsilon}R_{\epsilon e} = R_{m\epsilon}R_{\epsilon m} = -1$$

Note anyons on different codes do not see each other.

D. Error correction and decoding

Above we introduced several different codes to detect errors happening due to nature being nature. In this section we give an overview of how to fix such errors to preserve the information stored on a code.

Two different sorts of noise will disturb a code. One sort is the randomly appearing $\sigma_{x,y,z}$ this results in creating anyons. We measure the configuration of all stabilizers to receive the anyon distribution, this distribution is called a syndrome.

But this measurement is not perfect and is, therefore, the second source of noise. So an operation to correct the errors is needed. We could either find and apply this operation for every syndrome or wait until the end and then apply it then to fix the most recent errors. Unfortunately, fixing at the end does not work because the last syndrome does not specify errors on the actual last syndrome, but instead it tells us everything that happened including stuff that is no longer relevant represented as if it would be the final syndrome. We use an approach that combines best of both worlds. It is applied at the end, but knows what happened at each time step. We only look at changes between the syndromes of each time step consequentially and we get independent syndromes. For each we need to find an independent correction operator. In the end we apply the product over all of them.

Let's think about how to handle the measurement noise, which we often call lies. The syndrome of different points in time is independent. One can think of the lies as living on edges in time. We look at changes of the syndromes. Lies always occur as a pair of anyons on different points in time, so the measurement errors are linked through time compared to the spin errors which only ever can live at the same time and, therfore, are linked in space. So the problem is now a three dimensional one (two spatial dimensions for spin errors and another one for the lies) and the goal is now to find the best correction operator to apply at the end.

Once the syndrome is known it is necessary to determine the exact Pauli error E that caused it. In general this is not possible since several different E's can result in the same syndrome. Every syndrome is the result of four different classes of errors. These differ by the combination of all the possible logical operations applied to them.

$$1)\ E \quad 2)\ X \cdot E \quad 3)\ Z \cdot E \quad 4)\ X \cdot Z \cdot E$$

Choosing a configuration from the wrong class causes a logical error on the code. To most accurately correct the error, its necessary to determine the most probable equivalence class for the measured syndrome.

The step where the syndrome is (classically) processed to find the class of the error and fix it by applying to right Paulis is what we call decoding. We assume that all needed operations to move the anyons to pair them can be done noiselessly. A good decoder does not change the information stored on the code while correcting the errors. Therefore a decoder is benchmarked by its ability to fix errors without applying a logical error.

$$\text{logical error rate} = P_{X\text{or}Z} = \frac{\#\text{ of } X \text{ or } Z \text{ errors}}{\text{samples}} \tag{4}$$

This $P_{X\text{or}Z}$, where p is the physical error rate, L the grid size and $\alpha(p)$ describes the decay of the error rate dependent of the noise p, should behave [7] as:

$$P_{X\text{or}Z} = \mathcal{O}\left[e^{-\alpha(p)\cdot L}\right], \quad \alpha(p) \geq 0$$

This $P_{X\text{or}Z}$ is obtained for different L's as a function of p. The P's are plotted against each other and their intersection point is considered (up to finite size effects) to be the threshold of the investigated decoder. Threshold means a critical p_c where for

$$p < p_c$$

error correction is successful.

The p depends on the error model which is investigated. We use two different models. Here the $p_{x,y,z}$ denote the probability that the corresponding $\sigma_{x,y,z}$ occur.

- **independent bit and phase flips**: bit and phase flip have the same probability $p_{\text{bit flip}} = p_{\text{phase flip}}$, therefore:

$$\begin{aligned} p_x &= p_b \cdot (1 - p_p) \quad \text{and} \quad p_z = p_p \cdot (1 - p_b) \\ py &= p_p \cdot p_b \end{aligned} \tag{5}$$

- **depolarizing noise**: All Paulis happen with the same probability:

$$p_x = p_y = p_z = \frac{p}{3} \tag{6}$$

III. EMBEDDEDING $D(\mathbb{Z}_2)$ SURFACE CODES

The idea of this embedding technique is to take a stabilizer of a surface (or any other) code and interpret it as a quantum system with levels corresponding to the number of eigenvalues of said stabilizer. Then the vacuum and the anyon states can be understood as qubits.

For a given stabilizer, called S, all qubits are just in the groundstate $|0\rangle$ since $S|\psi\rangle = \psi\ \forall S$. For that reason it is necessary to define a new set of stabilizers for the underlying code. A new set allows the new stabilizer-qubits to act as actual qubits. The new, embedded code is then defined as an already known surface code.

A. Examples for $D(\mathbb{Z}_2)$ codes embedded into $D(\mathbb{Z}_2)$ codes

We focus on the $D(\mathbb{Z}_2)$ Surface Codes introduced in II B as building blocks for the embedding.

1. Wen plaquette model embedded into the Toric Code

The underlying code is the Toric Code introduced in section II B 2, where physical qubits live on the edges. We want them to stabilize a bicoloured Wen style code with qubits on the vertices. The only stabilizer of the Wen code has the form $W_p = \sigma_x \cdot \sigma_z \cdot \sigma_x \cdot \sigma_z$. The goal is to define these W_p's on plaquette operators of the Toric Code.

The stabilizer of the embedded code is defined on the two level system provided by the plaquettes of the underlying code. Therefore, we define operators, called X and Z, which act as the corresponding Paulis on the affected plaquette-qubits of the Toric Code. Each Wen plaquette requires a Z on the two plaquette qubits. This can easily be done using the plaquette operators of the Toric Code, called B_p and defined in eq. 4. Complementary to the Z's the stabilizer also needs two X's, those should anticommute with the the Z's. Consequently, an operator which creates a pair of anyons on the right plaquettes is introduced:

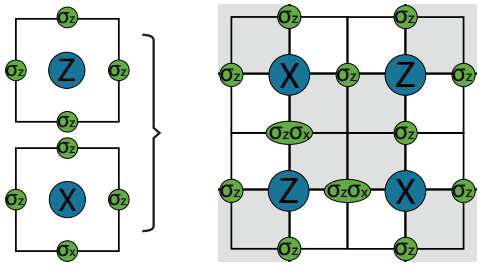

FIG. 8: We use the regular Pauli Matrices on physical qubits to implement Pauli-like operations on the plaquette plaquette qubits. With these its possible to embed the Wen plaquette operator, W_p on top of the Toric Code.

Those new stabilizers are centred around the vertices of the underlying Toric Code therefore they can be considered as an additional vertex stabilizer supplementary to the original toric vertex stabilizer defined in eq. 4. A_v and W_v commute as they should and define

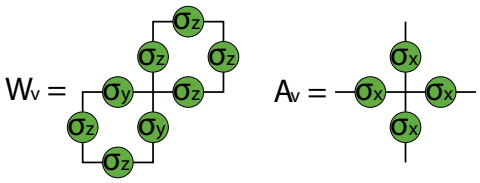

FIG. 9: Two stabilizers A_v and W_v are defined around each vertex of the original Toric Code.

a new stabilizer code. The question now is what is the anyon model on such a code and how is it related to the two codes used to build it.

We start by applying all three Paulis on horizontal and vertical edges of the underlying code.

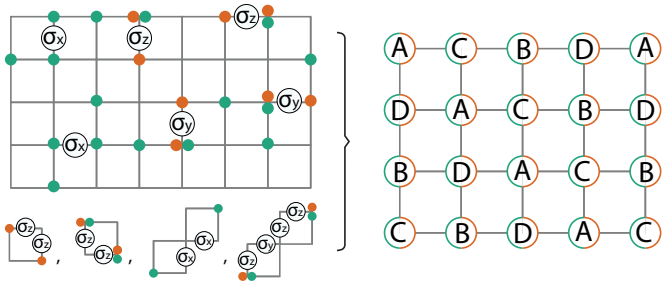

FIG. 10: On the left side we troubled the new stabilizers with all six configurations of Paulis on physical qubits. From that we followed the operations to move the two different kind of anyons around. On the right side there are the positions where the anyons are restricted to live

- σ_x on horizontal or vertical edges creates four anyons living on the W_v's, we call them teal.
- σ_z on horizontal or vertical edges creates two anyons living on the A_v's, we call them orange, and one of type teal.
- σ_y on horizontal or vertical edges creates two anyons and three anyons.

All create more than two anyons on the embedded Wen code. Two would imply a particle/antiparticle-pair situation. But, unfortunately, it is slightly more complicated. We construct operations of more than one Pauli to get the anyons moving. Both type of anyons there are restricted to move on diagonal lines. We distinguish the lines by calling them slash/backslash depending on what kind of movement they allow. Using the slashes (as introduced in fig. 10) it is possible to move both types of anyons across a plaquette of the underlying code. The backslashes are more restricted and only allow movement across two plaquettes.

Considering all this we conclude that there are four different kind of anyons, from now on called A, B, C, D for both stabilizers. The places were they are able to move to (as seen on the right side of fig. 10) are the places where they are allowed to live. The σ_y's in the W_v's link the stabilizer to A_v and both are sensitive to σ_z's. Using this its straightforward to derive the fusion rules for this code:

$$
\begin{aligned}
o_D \times o_A \times t_A = \mathbb{1} \quad &\rightarrow \quad o_D \times o_A = t_A \\
o_C \times o_B \times t_B = \mathbb{1} \quad &\rightarrow \quad o_C \times o_B = t_B \\
o_A \times o_C \times t_C = \mathbb{1} \quad &\rightarrow \quad o_A \times o_C = t_C \\
o_B \times o_D \times t_D = \mathbb{1} \quad &\rightarrow \quad o_B \times o_D = t_D
\end{aligned}
$$

Using slash and backslash for all four kinds of anyons of each stabilizer it is possible to braid them around each other to derive their braiding behaviour:

↻	t_A	t_B	t_C	t_D
t_A	+1	+1	−1	−1
t_B	+1	+1	−1	−1
t_C	−1	−1	+1	+1
t_D	−1	−1	+1	+1

↻	o_A	o_B	o_C	o_D
o_A	+1	−1	+1	+1
o_B	−1	+1	+1	+1
o_C	+1	+1	+1	−1
o_D	+1	+1	−1	+1

It is obvious that the usual e, m-anyons are not sufficient to reproduce such braiding behaviour. If we think of both surface codes as separate codes, both have their own set of anyons existing on them according to section II C. Lets call the anyons living on the Toric Code e_1 and m_1 and the ones on the Wen code e_2 and m_2 respectively. The embedded code has, therefore, an anyon configuration where both codes contribute.

A possible anyon configuration has to satisfy the fusion rules and the braiding logic. One that does both is:

$$\left.\begin{aligned} o_A &= e_1 \\ o_D &= m_1 \end{aligned}\right\} t_A = e_1 \times m_2 \qquad (7)$$

$$\left.\begin{aligned} o_C &= e_2 \\ o_B &= m_2 \end{aligned}\right\} t_B = e_2 \times m_1$$

$$\rightarrow t_C = e_1 \times e_2$$

$$\rightarrow t_D = m_1 \times m_2$$

We conclude that the anyon model of this new, embedded code is

$$D(\mathbb{Z}_2 \times \mathbb{Z}_2) \qquad (8)$$

Let's now think about how many anyons can be stored on such a code. The first step is to work out the properties of its unit cell. The periodic boundary condition on both sides is inherited from the underlying Kitaev. So the embedded Wen is living on a torus too. The operations, slash and backslash, to move around anyons restrict them to exist only in certain places. For that reason we need at least four (physical) qubits in each direction of the unit cell as pictured in fig. 11:

Now lets think about how many qubits we can store here. This number can be obtained by counting the amount of dependent stabilizer. There is one for each set of products of stabilizers which apply the identity on each involved physical qubit. A possible way of such a product is pictured in fig. 12. We require the used sets of stabilizers to be independent of each other. The class of the stabilizer is identified by which type of anyon exist on the right boundary according to fig. 10.

We find four such classes of combined sums which

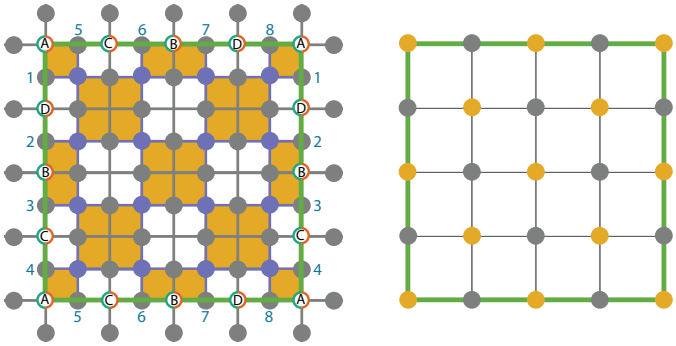

FIG. 11: On the left side is the unit cell, restricted by its outline, for a Wen code embedded into a Toric Code. Here the numbers from 1 to 8 indicate the same physical qubit on the torus. Circles are physical qubits lying on the Kitaev lines. Circles are the Wen qubits. On the right side are the bicoloured vertices of the original code.

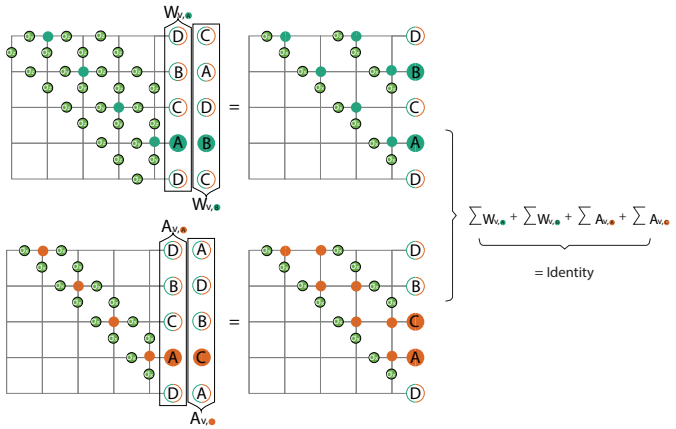

FIG. 12: The effect of the sum over all W_v's belonging to A-rows and B-rows has the same effect as the sum over all A_v's belonging to A-rows and C-rows.

The final effect of all the applied stabilizers is therefore the identity on the physical qubits.

apply the identity on the physical qubits.

(1) $\sum W_{v,A} + \sum W_{v,B} + \sum A_{v,A} + \sum A_{v,C}$

(2) $\sum W_{v,B} + \sum W_{v,A} + \sum A_{v,B} + \sum A_{v,D}$

(3) $\sum W_{v,C} + \sum W_{v,D} + \sum A_{v,C} + \sum A_{v,B}$

(4) $\sum W_{v,D} + \sum W_{v,C} + \sum A_{v,D} + \sum A_{v,A}$

For each of those classes there is one dependent stabilizer, we use it to store a logical qubit in.

Those elaborate products of stabilizers are consistent with the fusion rules of this model. To test it we use the fact that the sum of all anyons corresponding to a product of stabilizers which acts as the identity needs to be even.

We show how its done for the above pictured case (fig. 12), classified as class (1). The anyons living on this product are t_A, t_B, o_A and o_C. Using the fusion rules from above we find:

$$\begin{aligned} &\text{for } k \in \mathbb{N} \\ \#\text{anyons} &= k \cdot (t_A + t_B + o_A + o_C) \\ &= k \cdot (o_D \times o_A + o_C \times o_B + o_A + o_C) \\ &= \text{even} \end{aligned}$$

o_A and o_C occur $k \cdot 2$ times and are consequently even. There are k remaining o_D's and o_B's. But since the dimension of the unit cell is even in both directions the k is even as well. Both kind of remaining o's occur therefore with an even number and the equation is satisfied.

Similar arguments can be used for all the other classes and we always find that it is possible to store 4 logical qubits on this code.

2. *Toric Code embedded into the Toric Code*

In section III A 1 a Wen code was embedded in to a Toric Code but now we show how to embed another Toric Code instead. For the other model it was enough to use the orientation of the codes as they are. Here an additional step is required in order to embed a Toric Code into a Toric Code. We do not want to just have one code on top of another one, which would happen if we would use the same embedding as above. It is necessary to rotate the lattice of the code-to-embed to be compatible with the way the plaquettes of the undlying code are defined.

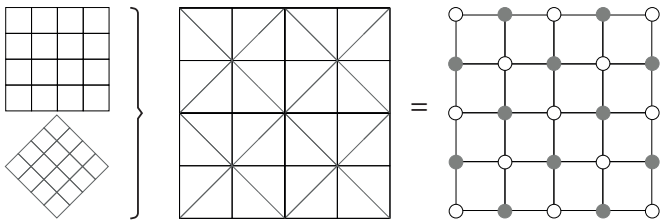

FIG. 13: We overlay the original Toric Code with the rotated one. The final code is therefore defined on a lattice with qubits on edges and bicoloured vertices. The grey vertices are aligned with the vertices of the embedded code, the black ones correspond with the plaquette operators of the embedded code.

Each stabilizer of the embedded code is centred on a vertex. The two different stabilizers correspond with bicoloured vertices on the original code. For further purposes we call stabilizers associated with vertex stabilizers, qubits in fig. 13, by the index v. The stabilizers corresponding with the embedded plaquettes, black qubits in fig. 13, are denoted by the index w. Caused by there embedding there is a second stabilizer acting on each vertex.

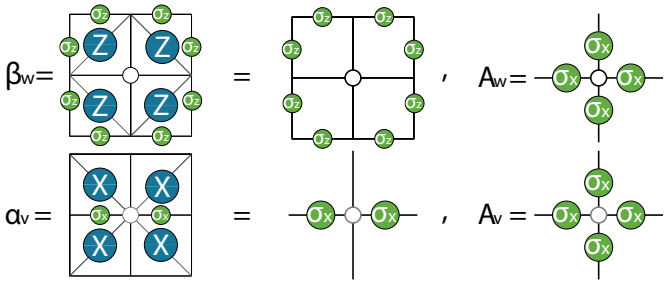

FIG. 14: As before qubits belong to the embedded code and qubits are the physical ones. Each vertex is now stabilized by two stabilizers. $A_{v \text{ or } w}$'s are the vertex stabilizers of the original code, they dont depend on whether it is a v or w type of vertex.

Now that the stabilizers are defined lets think about what kind of anyons live on such a code. First note that there are two different kind of β's. The reason for such behaviour is that the β's are sensible to σ_x's. If a σ_x is applied it acts as a X on the plaquette qubit and its neighbours. The plaquette qubit in the middle receives two X's and is therefore free from errors but the adjacent plaquettes are not so lucky. On both of them an anyon lives now. Therefore, there are two different kind of β's, where same β's show up on every second horizontal line.

For vertical lines similar behaviour occurs, but the reason is different. From now on we call the two different lines by the color of the first contributing vertex. There are black and grey lines. The difference between them is how they are affected by the surrounding α's, the A's act the same way on both lines. We find if a σ_z acts on vertical edges it bothers only the A's but if it affects horizontal edges it triggers an α as well. So from now on we no longer distinguish anyons living on A's by their stabilizer but by their vertical line instead.

According to the vertical lines we define the indices of the β's. β_1's live on all horizontal lines which start with a black vertex and the β_2's live on the other ones. This is pictured in fig. 15.

For bookkeeping reasons give the anyons some names.

- On the β_w's live the anyons β_1 and β_2
- On $A_{v \text{ or } w}$ of the vertical lines starting with a black vertex, live the anyons v_B

- On $A_{v \text{ or } w}$ of the vertical lines starting with a gray vertex, live the anyons v_G
- On the α_v's live the anyons α

The fusion rules for this code are straightforward with this new definition for the where the anyons live:

$$\begin{aligned} \beta_1 \times \beta_1 = \beta_2 \times \beta_2 = \mathbb{1} \\ v_B \times v_G \times \alpha = \mathbb{1} \rightarrow v_B \times v_G = \alpha \\ v_B \times v_B = v_G \times v_G = \mathbb{1} \end{aligned}$$

As before its possible to braid all anyons around each other to obtain their braiding behaviour:

↻	β_1	β_2	v_B	v_G	α
β_1	+1	+1	+1	−1	−1
β_2	+1	+1	−1	+1	−1
v_B	+1	−1	+1	+1	+1
v_G	−1	+1	+1	+1	+1
α	−1	−1	+1	+1	+1

A possible anyon configuration satisfying this braiding logic and the fusion rules above is:

$$\begin{aligned} \beta_1 = e_1, \quad \beta_2 = e_2 \\ v_B = m_2, \quad v_g = m_1 \quad \rightarrow \quad \alpha = v_B \times v_G = m_1 \times m_2 \end{aligned} \tag{9}$$

We conclude that the anyon model of this new, embedded code is the same as before where we embedded the Wen code into the Toric Code.

$$D(\mathbb{Z}_2 \times \mathbb{Z}_2)$$

Now let's look at the properties of the unit cell in order to understand how many logical qubits can be stored on such a code. We know:

- This code uses the periodic boundary conditions from the underlying code.
- There are two different kind of β's, where same β's show up on every second horizontal line. Each sort stretches over two lines of original plaquettes. At least two horizontal lines of each sort are needed. The third can be the first again

Considering this we find the unit cell:
Now let's think about logical qubits. As in the last model we look at the amount of depending stabilizers to determine how many logical qubits we can store here.

- **σ_x-Stabilizers:** Applying all the A_v's, A_w's, and α's acts as

$$\begin{aligned} \sigma_x \cdot \sigma_x \cdot \sigma_x &= \sigma_x \quad \forall \text{ qubit } \in \text{ vertical lines} \\ \sigma_x \cdot \sigma_x &= \mathbb{1} \quad \forall \text{ qubit } \in \text{ horizontal lines} \end{aligned}$$

This is exactly the same as if only the α's would have been applied.

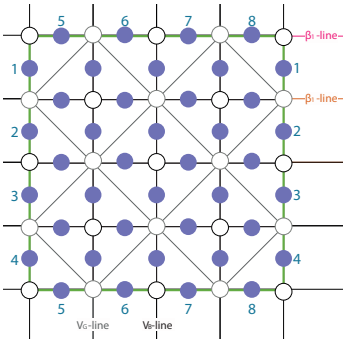

FIG. 15: This is the unit cell for a Toric Code embedded into a Toric Code. Here the numbers from 1 to 8 indicate the same physical qubit on the torus. Circles are physical qubits lying on the black lines of the original code. Black/ grey circles are the qubits of the embedded code. The frame is the outline of the unit cell.

Removing one of the $A_{v \text{ or } w}$'s applies the unity on all qubits on horizontal lines belonging to the removed stabilizer and a single σ_x on qubits of its vertical lines. This is the the same effect as if the removed stabilizer had been applied on the grid of pre-applied α's.

$$A_{\tilde{v} \text{ or } \tilde{w}} = \prod_{v \neq \tilde{v} \text{ or } w \neq \tilde{w}} A_{v \text{ or } w}$$

This implies that both the A's each have one dependent stabilizer and therefore we can store two logical qubits on those three stabilizers built up from σ_x's.

- **σ_z-Stabilizers:** The β's are only applied around qubits associated with embedded plaquette operators, they act as the unity on each physical qubit:

$$\sigma_z \cdot \sigma_z \cdot \sigma_z \cdot \sigma_z = \mathbb{1}$$

Two of the σ_z's are from the β_1's and the other two from the β_2's. So due to linewise bicoloration and the fact that all the β's behave as if they were regular plaquette operators just stretched across four plaquettes, it follows for both sorts of them:

$$\beta_{1 \text{ or } 2,\tilde{w}} = \prod_{w \neq \tilde{w}} \beta_{1 \text{ or } 2,w}$$

This implies that of all the β's one β_1 and one β_2 are dependent. It follows that we can store one logical qubit in each.

Again we find space to store four logical qubits for this first case of the model.

3. Planar Code embedded into the Planar Code

We are embedding one Planar Code into another Planar Code. This model is different to the ones above. Earlier

we defined the the embedded code exclusively on the plaquette qubits of the original code. But for this model it is required to use both the plaquette qubits as before and the vertex qubits. On those vertices there might live an (e-)anyon, if thats the case the vertex is associated with the state $|1\rangle$. Therefore a Z acts as an X on vertex plaquettes, it lets an e-anyon (dis-) appear. The plaquette-qubits behave the same as in previous models.

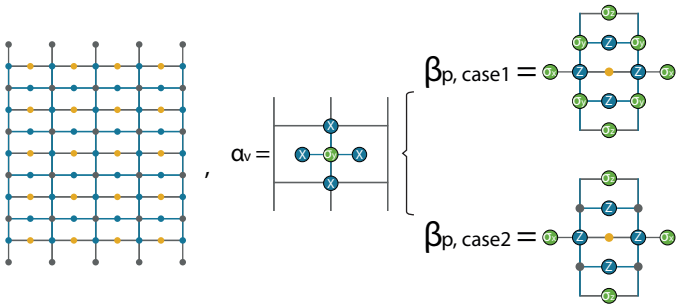

FIG. 16: On the left side is embedded code for this model. The embedded code and its qubits lie on the original code. Its necessary to distinguish between physical qubits of the original code associated with qubits on embedded vertices and the ones on embedded plaquettes.
On the right side are the stabilizers for such a code. Operation(s) are applied to the physical qubits and imply the operation(s) on the embedded code. Note there are two different ways to define the plaquette stabilizer β.

The qubits in fig. 16 are associated with the vertices of the embedded code. We want an X on all the qubits surrounding the grey one. This happens by applying a Z on the qubits on the vertical line and a X on the two qubits of the horizontal line (those are the ones living on a plaquette qubit).
So we end up with the α_v stabilizer in fig. 16, which is nothing else but a single σ_y on all physical qubits centred by an embedded vertex.
Now we look at two different cases of possible stabilizers for plquettes of embedded Planar Code:

Case 1: We start by looking at the first case. This one is more complicated, so afterwards the first case is almost straightforward.

In this case the stabilizers for the yellow qubits are as pictured on top of the the right side of fig. 16. These are characterized by the contribution of the four σ_y's to the stabilizer. In this version of the model the vertex and plaquette operators are linked by the σ_y's which are part of both.

Let's look at what happens if one of the physical qubits is bothered by either a σ_x's or a σ_z's:

If a grey qubit is affected, all of them are stabilized by a σ_y's, then there are four anyons on the surrounding yellow qubits popping up. In addition to the four, there are two more anyons appearing on places centred by the adjacent embedded vertices. The position of those depends on the nature of the applied Pauli matrix:

- σ_z generates anyons on the vertices connected to the affected σ_y by a horizontal line
- σ_x generates anyons on the vertices connected to the affected σ_y by a vertical line

The yellow qubits only belong to the β's. But as before the effect on the involved stabilizers depends on the sort of chosen Pauli:

- σ_x generates anyons on the left and right adjacent yellow qubit centred by plaquettes
- σ_z generates anyons on the top and bottom adjacent yellow qubit centred by plaquettes

Using these anyon-pairs on yellow qubits it is possible to define the operations to move around the anyons living on grey qubits. From now we refer to anyons living on grey qubits by A and to the other ones by B.

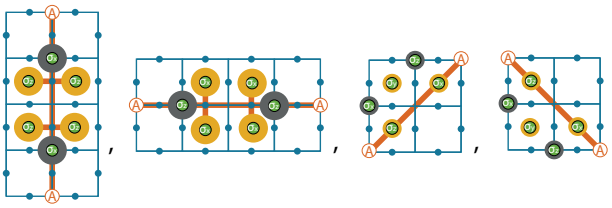

FIG. 17: The four operations to move the A's in respect to the lattice of the embedded code. The operations on physical qubits have a grey/yellow background depending on what kind stabilizer centres them.

The operations to move around the A's, allow us to move them mod 4, so one would expect eight different kind of A's. But we have a second set of operations to move anyons around on grey qubits, slash and the backslash. Using them it follows:

$$A_1 = A_5,\ A_2 = A_6$$
$$A_3 = A_7,\ A_4 = A_8$$

This corresponds with the fact that there are also only four different kind of B's (straightforward since σ_x and σ_z generate pairs on every second plaquette) living on the embedded plaquettes, we call them B_1 to B_4.
Without loss of generality from now on we use the following order to stay consistent, where the A's live on the

vertices and B's on plaquettes:

$$\begin{array}{ccccc} A_1 & & A_2 & & A_1 \\ & B_1 & & B_2 & \\ A_3 & & A_4 & & A_3 \\ & B_3 & & B_4 & \end{array}$$

We braid them all around each other and find:

$$A \circlearrowleft B = B \circlearrowleft A = -1 \quad \forall \quad A,\ B$$

$$A \circlearrowleft A = 1 \quad \forall \quad A$$

For the B's we find the same braiding logic as for the anyons in the model where a Toric Code was embedded in to another one:

$\circlearrowleft$	B_1	B_2	B_3	B_4
B_1	+1	+1	+1	−1
B_2	+1	+1	−1	+1
B_3	+1	−1	+1	+1
B_4	−1	+1	+1	+1

The anyon configuration of the B's is therefore the same and the A's correspond to a composite anyon:

$$\begin{aligned} B_1 &= e_1, \quad B_2 = e_2, \quad B_3 = m_1, \quad B_4 = m_2 \\ A_1 &= A_2 = A_3 = A_4 = e_1 \times m_1 \times e_2 \times m_2 \end{aligned}$$

This is very similar to what we found before, where the fifth anyon is also a composite of the other ones generated by the same error. Here the anyons are linked trough the Pauli-y. Consequently the fifth is a composite of all of them.

Case 2: This case is characterized by the other stabilizer for the yellow qubits in fig. 16. To derive it one takes the product of the two from the first case and use it to replace the old β. This new stabilizer which is free of the σ_y-linking-behaviour.

The operations for moving around pairs are the same as above for B's and similar ones for A's:

- σ_z on grey qubits generates anyons on vertices on the horizontal line and if applied to a yellow qubit it generates anyons on plaquettes on the vertical line

- σ_x on grey qubits generates anyons on vertices on the vertical line and if applied to a yellow qubit it generates anyons on plaquettes on the horizontal line

Here we have again four different anyons on each kind of qubit. Lets use the same names as before and do the braiding. The B's braid the same way as before. And the A's braid as the B's.
We end up with the same logic we found for the B's in the first case which now applies to the A's too.

$$\begin{aligned} A_1 &= B_1 = e_1, \quad A_2 = B_2 = e_2 \\ A_3 &= B_3 = m_1, \quad A_4 = B_4 = m_2 \end{aligned}$$

We conclude that the anyon model of this new, embedded code is the same for both definitions of the plaquette stabilizers as in the models before:

$$D(\mathbb{Z}_2 \times \mathbb{Z}_2)$$

B. Breaking the theory

So far all the codes we investigated share the properties of the $D(\mathbb{Z}_2 \times \mathbb{Z}_2)$ anyon model. But to conclude that this is true for all $D(\mathbb{Z}_2)$ is wrong. We show in this section how to construct an embedded code with a different anyon model.

The first thing we do is revisit section III A 2 in order to use our knowledge of how the codes above get $D(\mathbb{Z}_2 \times \mathbb{Z}_2)$ anyons to construct or find one that doesn't have these anyons.

Square lattices: The code we explored in section III A 2 is a Toric Code embedded into another one. It is defined on a square lattice, which can be bi-coloured similar to fig. 6, from now on called □ and □, but with qubits living on edges instead of the vertices.

Applying the product over all plaquette operators (as defined in fig. 13) of one color has the same effect for both colors, it act as a Z on all plaquette qubits:

$$\prod_{p \in \square} \beta_p = \prod_{p \in \square} \beta_p = \prod_j Z_j$$

In this case the effect of Z is not the same as the plaquette operator of the original code (defined in eq. 4), since on the original code the product over all B_p's is the unity. Putting it all together it follows:

$$\overbrace{\prod_{p \in \square} \beta_p = \underbrace{\prod_{p \in \square} \beta_p = \prod_j Z_j}_{\text{condition two}} = \prod_j B_p = \mathbb{1}}^{\text{condition one}}$$

Therefore, for this model of an embedded code there are two conditions for the plaquette operators rather

than the single one we obtain in a regular Toric Code. It implies that the bicoloured plaquettes hold different kinds of anyons. Which leads to the same conclusion that this code indeed has the anyon model $D(\mathbb{Z}_2 \times \mathbb{Z}_2)$.

Other lattices: We want to construct an embedded code that does not satisfy the conditions above. This means a code where no possible subset of the plaquette operators, such as the product over the subset, acts as the product of Z's on all plaquette qubits. One way of doing this is to embed a code on a hexagonal lattice. This requires the original code to have six- and three body vertices connecting the plaquettes. A code that satisfies these requirement can be seen on the left side of fig. 18. The qubits living on the edges and the vertices are bicoloured to distinguish between the two sorts of them:

It is not possible to define single α stabilzers be-

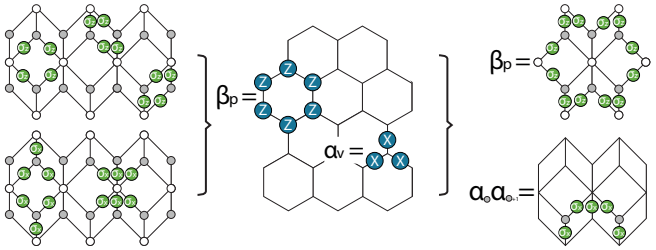

FIG. 18: On the left side is the code we constructed to embed a hexagonal code. It has three different vertex/plaquette stabilizers. As before green operations are the ones acting on the physical qubits. The vertex stabilizers are in the bottom row (plaquettes on top), the three body ones center grey vertices and the black vertices are center the six body operators.
In the middle there are the stabilizers of the embedded code. On the right side what physical Paulis are needed to perform the operators.

cause they need three X's which would imply creating three plaquette anyons. To prevent that we use the product of two neighbouring α's as defined in fig. 18.

So it looks like the code is completely defined and it is possible to get the anyon configuration from here onwards. But thats not true. The six-body-vertex stabilizers are the problem, from now an called $A_{v\circ}$. Applying one of them is fine, but all of them generate loops of anyons on the plaquettes, which are overlapped by other $A_{v\circ}$'s. In order for it to work we need a set of vertex stabilizers with no overlap, which is equivalent to the ones in fig. 18. In fig. 19 we show such a set:

The set of vertex stabilizers defined in fig. 19 along with the β_p's as seen in fig. 18 are the final stabilizers where we think our anyons live. For simplicity we just denote the anyons living those stabilizers by the name of the

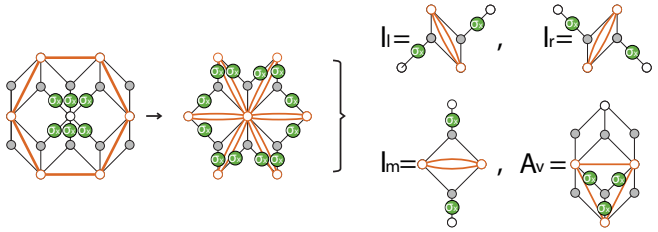

FIG. 19: On the left side is the loop generated by a single $A_{v\circ}$, applying all of them generates loops on all plaquettes as shown in the middle. Each $A_{v\circ}$ is in the center of six plaquettes and, consequently, overlaps the six loops on them. On the right side are the operators we use instead. The three different l's are used to handle the loops in-between the three-body-vertex stabilizers, called $A_{v\bullet}$.

stabilizer and find the following braiding logic:

↻	β	$A_{v\bullet}$	l
β	+1	−1	+1
$A_{v\bullet}$	−1	+1	+1
l	+1	+1	+1

This corresponds with the

$$D(\mathbb{Z}_2)$$

anyon model, where

$$\beta = m, \quad A_{v\bullet} = e, \quad l = m \times m \quad \text{or} \quad l = e \times e$$

We showed a way to embed a $D(\mathbb{Z}_2)$ surface code into an other one where only original anyons exist. This disproves the conjecture that $D(\mathbb{Z}_2) \times D(\mathbb{Z}_2)$ anyons always come out of these codes.

IV. DECODER

None of the codes in the first section is of much use if there is no decoder available to understand if the embedded codes provide advantage over regular ones. The second part of this thesis is to present a decoder which could be further developed to decode embedded code.

The decoder is set up to decode syndromes of a Wen style Planar Code as introduced in section III A 1. The reason for choosing that specific code is that, since the qubits live on vertices, their positions can be understood as an entry of a matrix. We modified program code for the simplest decoder introduced in [8] to suit our purposes.

A. Bookkeeping

This section is a bit technichal but necessary in order to understand how this decoder works. This code is bicolored and defined similar to fig. 7. This corresponds to a matrix of dimension $L + 4 \times L + 4$. The $+4$ are dummy anyons (or entries of the matrix) needed to store information.

It is crucial to separate the real anyons, residing on the actual code, from the so called virtual anyons which live on the edge anyons. Whatever a stabilizer learns about the spins contributing to its value, is always stored on the top left corner of the plaquette. This is also true for the two-body stabilizers and it happens that their value is stored on a matrix entry which is outside of the code.

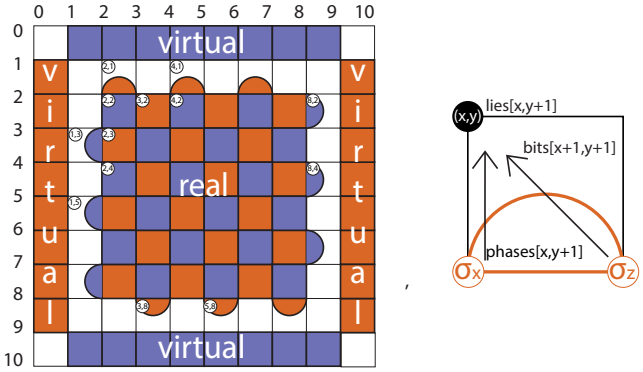

FIG. 20: On the left side is a bicoloured Wen style Planar Code where $L = 6$. This corresponds with a matrix of dimension 10×10. White squares correspond with empty dummy entries. On the left and top side the value of the two-body stabilizers is not stored on the inner matrix. On all four sides are places to store the edge anyons, which are the virtual ones. The right side shows how the stabilizers were implemented. This is an example for one of the top class, the syndrome of this two body plaquette is stored in the matrix entry corresponding to the top left entry. The others work similarly.

A important number for any decoder is the minimal amount of errors on qubits needed to create a logical error. This is called the distance of the of the code d. For this code it is:

$$d = \frac{L-4}{2}$$

It's equivalent to the number of flips one would need in order to create a pair of anyons for which it takes more flips to pair them with each other than with the opposite edges. It is equal because if errors occur on more than half the qubits, then it takes less operations to decode the syndrome in a wrong way, resulting in a logical error, then it would take to decode it the right way.

B. first version, the classless model

This first version of the decoder is called classless for the reason that it does not distinguish between different error classes. Its basically standard MWPM decoding for a the Planar Code. The difference to the regular one is that its able to handle imperfect measurements.

1. Simulation

The simulation consists of two important parts, first we create noise then we try to fix it:

```
createProblem(px, py, pz, pm);
```

and

```
correct(px, py, pz, pm);
```

Lets look at both in a bit more detail. The first one generates a set of errors on all qubits belonging to the code. The errors are applied according to the chosen error model. In fig. 20 those are the entries belonging to the inner matrix plus the outside ones corresponding with the top and left two body plaquette stabilizers. These errors are saved in three different arrays and since we do not assume perfect measurements, there is a coordinate to count how many measurement rounds happen:

```
int bits[T][Lx][Ly];
int phases[T][Lx][Ly];
int lies[T][Lx][Ly];
```

Those errors are then computed to find the syndrome. Every plaquette's stabilzer is computed according to its sensibility to the different errors, for example the top stabilizer from fig. 20:

```
syndrome[t][x][y] = phases[t][x][y+1] +
    bits[t][x+1][y+1] + lies[t][x][y];
```

The syndrome array is then processed by the correct function. It alters the logical operators based on the syndrome in an attempt to correct the effects of the errors.

The correct function loops over all syndrome entries to find all the anyons living on the code. Then for every real anyon it adds a corresponding virtual one. If it is a real anyon on the left half it adds a virtual anyon on the left, and the same for the right half. And similar for top/bottom half and the real anyons. Adding those virtual anyons is necessary in order to tell if a logical should be applied or not. The matching is done for both kind of plaquettes individually in the function called

```
blossom (...)
```

Given the list of where the anyons live this function uses MWPM to find the matches and alters the logical operators accordingly. For the matching we use Edmonds's MWPW algorithm [9]. It is able to (efficiently) find pairing of minimal weight for a graph with weighted

edges and an even number of vertices. Luckily there is already an available programm package, called Blossom V (introduced and avalaible at [10]) which implements this algorithm.

We set up Blossom to accept a problem according to how many anyons (called by n_{anyon}) and connecting edges there are:

$$\begin{aligned} \text{edges} &= \frac{n_{\text{anyon}} \cdot (n_{\text{anyon}} - 1)}{2} \\ \text{vertices} &= n_{\text{anyon}} \end{aligned}$$

Then we tell Blossom about each pair of syndrome elements and the weight between them. This weight depends on the distance in space and time. Virtual anyons match free from any cost, consequently the distance between them is set to zero. For all others we compute the distance but there are things to keep in mind:

- anyons move zigzag-like from one plaquette to another of the same sort, so its required to alternate between the cost for σ_x and σ_z-flips.
- the distance from a real anyon to a virtual has to always be straight zigzag because there is actually only one delocalized virtual anyon stretched over each side of the code. So there are no diagonal movements if a real anyon matches with a virtual.
- if two real anyons match the spatial distance is diagonal followed by a zigzag.
- time distance is 'straight' from one time-slice to another.
- its necessary to keep track of the dummy entries, the distance is changed in a way that does not add weight for dummys.

Blossom finds the MWPM. Now we loop over every anyon and check if the anyon and its partner cross a boundary. If so the logicals are changed accordingly.

This procedure is repeated for different levels of noise and different L's in order to get the threshold.

2. Results

The numerical results for this simulation are plotted in fig. 21 and fig. 22. Up to finite size effects the threshold for two measurement rounds and a constant probability for measurement errors $p_m = 1\%$ is:

$$\left.\begin{aligned} T_{cX,\text{classless, indep.}} &= 6.33\% \\ T_{cZ,\text{classless, indep.}} &= 6.13\% \end{aligned}\right\} T_{c,\text{classless, indep.}} = 6.23\%$$

These results are obtained for an independent noise model as introduced in eq. 5. There there is no correlation between σ_x and σ_z-errors. And since MWPM considers both sorts of anyons independently it is well suited (but also restricted) to handle this kind of noise. If one wants to investigate more advanced noise models such as depolarized noise (as introduced in eq. 5) it can only be done under the assumption that bit and phase flip errors happen independent of each other:

$$\begin{aligned} p_b &= p_x + p_y \\ p_p &= p_z + p_y \end{aligned}$$

Consequently, we developed a second decoder, called classy, specifically designed to tackle this issue.

C. second version, the classy model

The suboptimal behaviour of the classless decoder is caused by the fact that σ_y is the product of the two other Pauli matrices up to a constant factor. By assuming independent bit and phase errors it can not differ if a $\sigma_{x \text{ or } z}$ is actually caused by a $\sigma_{x \text{ or } z}$ or by a σ_y. So the new feature we implement is that this second decoder is able to distinguish all three possible errors.

Lets think about the behaviour of σ_y errors. Generally they are identified by their effect on the plaquettes. Since they bother both kind of stabilizers, anyons appear on both sorts of plaquettes. Consequently a string of $\sigma_{x \text{ or } z}$ is accompanied by anyons of the other type if the string consists σ_y's. These additional anyons can be used to identify the σ_y's in the string. So, if there are some, the string does not contain errors of only one sort, which would be needed to result in a logical error. Therefore, a string of errors is only able to cause a logical error if there is maximally one σ_y in it.

1. Simulation

A logical error applies to a code if the decoder chooses a correction operator of the wrong error class (section II D). There are four different error classes and each of them corresponds with different strings of errors. We force the decoder to calculate MWPM for each class. We then know which anyons are paired and this information is used to compute the corresponding string of errors that would result in this configuration.

This results in certain amount of flips for each syndrome. Those flips happen according to the physical error rate. A single σ_y gets us two errors for the price of one. Consequently, it is more probable that a σ_y has happened than both a σ_x and a σ_z. This calls for several modifications in regarding the classless decoder.

To force the decoder to do the matching for all classes of errors it is necessary to classify these four classes. A logical state is specified by the occupancy of

its edge anyons. We can change this by altering how many virtual anyons there are. To get all four classes it is enough to add one virtual anyon on corresponding edges:

- no anyon is added

```
// error class 0 - regular
```

- two anyons (one on the left and one on the right edge) are added

```
//error class 1 - black +2
```

- two anyons (one on the top and one on the bottom edge) are added

```
//error class 2 - whites + 2
```

- class one and two combined, four anyons are added

```
//error class 3 - both +2
```

Note: we stored for every class of errors the logical operators required to fix it so we can later choose the one that corresponds with the actual error chain. All the anyon pairs are stored in their respective arrays. We use those to estimate the string of errors that caused the each pair for all classes. This is done via the functions

```
zigzag(...)
slash(...)
backslash(...)
```

Those estimated syndromes are then processed to count all the bit and phase flips that caused them. The problem is now that each sort of anyon is computed as a separate case. But its possible to link those two cases using a function to loop over all positions and compare the errors that occurred.

```
countvector[ec][0]+= sx*((sx+sz)%2);
countvector[ec][1]+= sx*sz;
countvector[ec][2]+= sz*((sx+sz)%2);
countvector[ec][3]+= sm;
```

We now know for each class of errors the corresponding error chain and can calculate its probability. We then chose the most probable one and apply the logical operators accordingly.

2. Results

The first set of results were computed for the same configuration as in the classless case. This was done to compare if classy works, since it is supposed to perform similar to classless if used to decode errors according to the independent error model where classy has no advantage. The numerical results for this simulation are plotted in fig. 23 and fig. 24. Up to finite size effects the threshold for two measurement rounds and a constant probability for measurement errors $p_m = 1\%$ is:

$$\left.\begin{aligned} T_{cX,\text{classy, indep.}} &= 5.42\% \\ T_{cZ,\text{classy, indep.}} &= 5.56\% \end{aligned}\right\} T_{c,\text{classy, indep.}} = 5.49\%$$

It is not as good as classless but good enough for our purposes. One reason for the discrepancy of the thresholds is that we used only grid sizes up to $L = 24$ for the classy decoder, but bigger ones for classless.

Overall both perform almost similarly this is noticeable from fig. 25 where both deliver $\pm$ the same error rates only depending on the chosen grid size.

Let's now compare the two decoders for depolarized noise. Once again we obtain a higher threshold for classless (plot is fig. 26) :

$$\begin{aligned} T_{c,\text{classless, depol.}} &= 9.61\% \\ T_{c,\text{classy, depol.}} &= 8.87\% \end{aligned}$$

We do not offer any explanation for this behaviour.

It is most interesting below threshold, where it matters (see fig. 26 for plot of $p \to 0$). The closer the simulation gets to this limit the better classy performs compared to classless. We find

$$\text{classy}(L-4) \approx \text{classless}(L)$$

for all L's investgated. Therefore it is possible to have an equally good error correction with significantly fewer qubits by using this improved decoder compared with regular MWPM.

V. CONCLUSION

In this thesis we considered different examples of $D(\mathbb{Z}_2)$ surface codes and showed how to embed them into other surface codes where new stabilizers were defined to fit the requirements of the embedded code.

Different stabilizers are accompanied by different anyons. We investigated the anyon model of those codes using their fusion rules and their braiding logic to identify a possible anyon configuration that satisfies the necessary conditions. Consequently we were able to conclude the anyon model associated with the investigated code. We constructed an example to disprove that the anyon model is the same for all cases, which is the main result of the first part of this thesis.

The second part is centred around using regular MWPM algorithms to develop a decoder. We improved this decoder by pointing out that MWPM always considers both sorts of erros σ_x and σ_z independently. This is one systematic weakness, which restricts its decoding

powers for more realistic error models. Consequently, we enhanced the decoder's capabilities with the power to handle correlated error models. We found that the improved version of the decoder uses significantly fewer qubits for the limit noise $\rightarrow$ 0 to obtain the same logical error rates compared with the first version of the decoder.

VI. ACKNOWLEDGEMENT

Special thanks goes to

- James R. Wootton

This thesis would not have been possible without his support, supervision and inspiring discussions during the whole project.

The author also thanks:

- Markus S. Kesselring

for letting the author explain strange stuff to him as long as the author needed in order to understand it.

[1] J. R. Wootton, "A family of stabilizer codes for D($\mathbb{Z}_2$) anyons and majorana modes," *Journal of Physics A: Mathematical and Theoretical*, vol. 48, p. 215302, may 2015. Available online at `http://arxiv.org/abs/1501.07779`; visited on December 7th 2015.

[2] H. Bombin and M. A. Martin-Delgado, "Topological quantum distillation," *Phys. Rev. Lett.*, vol. 97, oct 2006. Available online at `http://arxiv.org/abs/1501.07779`; visited on December 11th 2015.

[3] J. R. Wootton, "Lecture: Topological quantum computing and quantum information." Website, 2013. Available online at `https://sites.google.com/site/woottonjames/`; visited on December 28th 2013.

[4] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.

[5] R. Shankar, *Principles of Quantum Mechanics, 2nd Edition*. Plenum Press, 2011.

[6] J. Pachos, *Introduction to topological quantum computation*. Cambridge: Cambridge University Press, 2012.

[7] J. R. Wootton, "A simple decoder for topological codes," *Entropy*, vol. 17, pp. 1946–1957, apr 2015. Available online at `http://arxiv.org/abs/1310.2393`; visited on December 10th 2015.

[8] A. Hutter, J. R. Wootton, and D. Loss, "Efficient markov chain monte carlo algorithm for the surface code," *Phys. Rev. A*, vol. 89, feb 2014. Available online at `http://arxiv.org/abs/1302.2669`; visited on November 4th 2015.

[9] J. Edmonds, "Paths, trees, and flowers," *Journal canadien de mathématiques*, vol. 17, pp. 449–467, jan 1965.

[10] V. Kolmogorov, "Blossom v: A new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 43, no. 9, pp. 43–67, 2009. Available online at `http://pub.ist.ac.at/~vnk/software/blossom5-v2.05.src.tar.gz`; visited on December 12th 2015.

Appendix

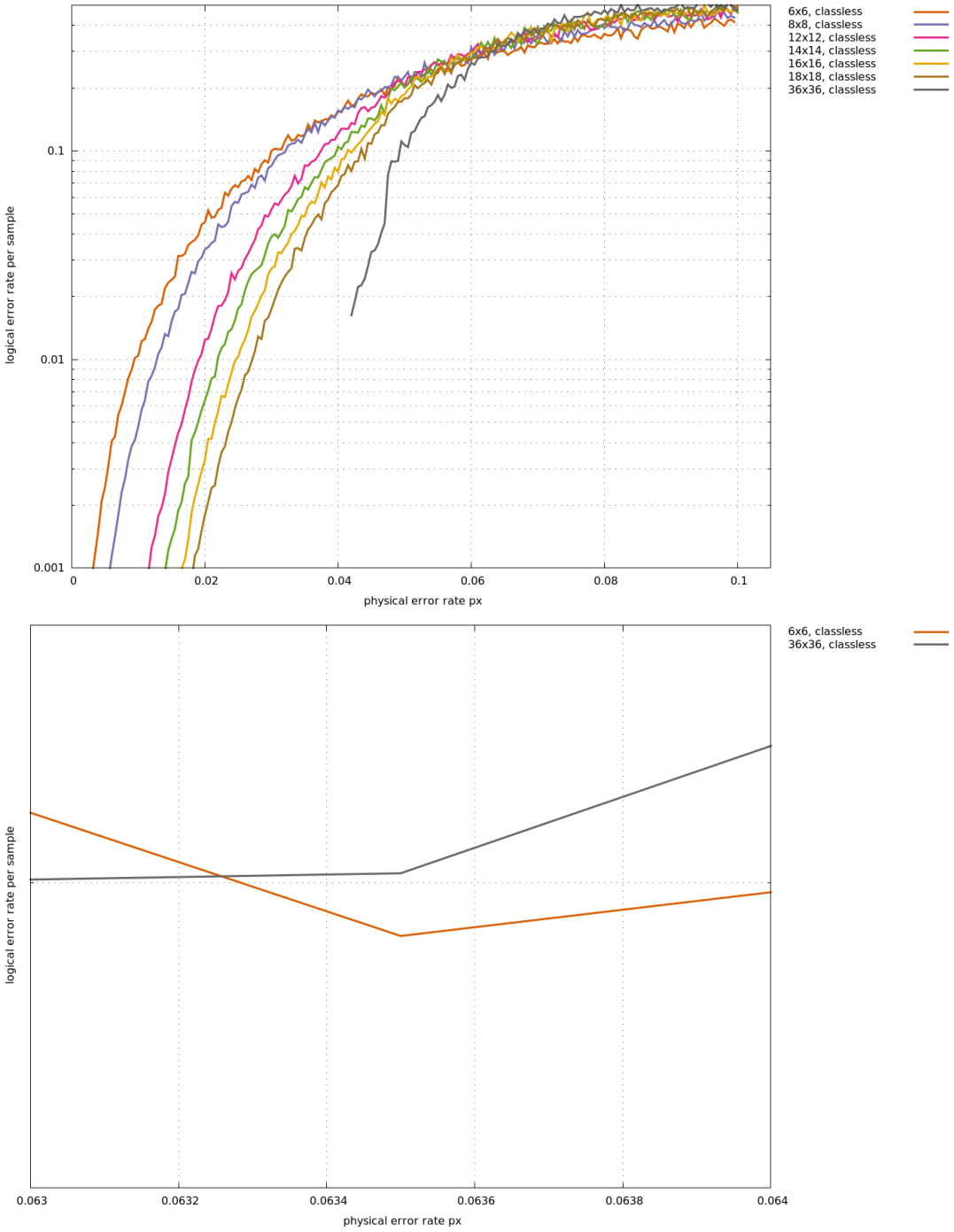

FIG. 21: Treshhold is obtained for: classless decoder, independent errors, logical errors X, two measurement rounds and a probability for measurement errors $p_m = 1\%$

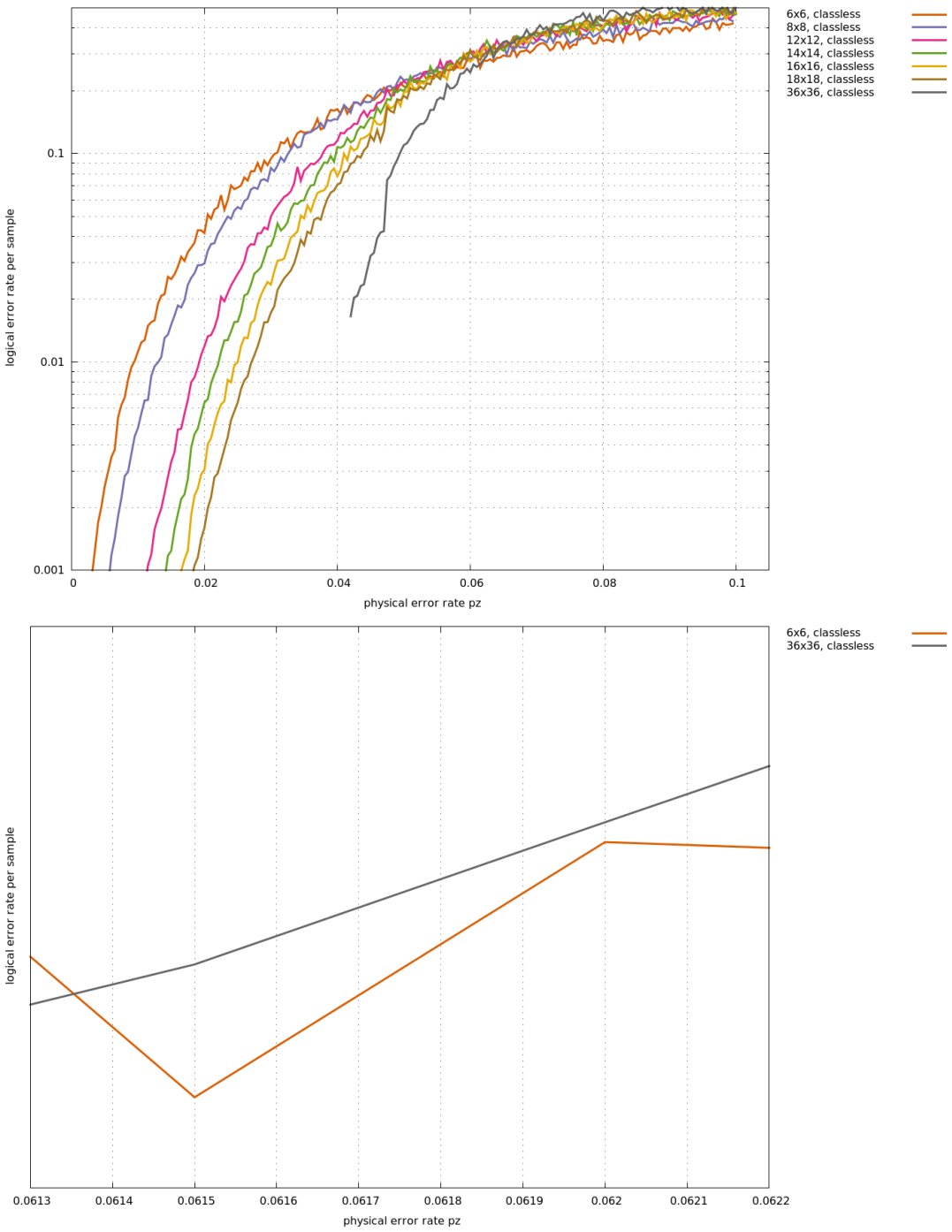

FIG. 22: Treshhold is obtained for: classless decoder, independent errors, logical errors Z, two measurements round and a probability for measurement errors $p_m = 1\%$

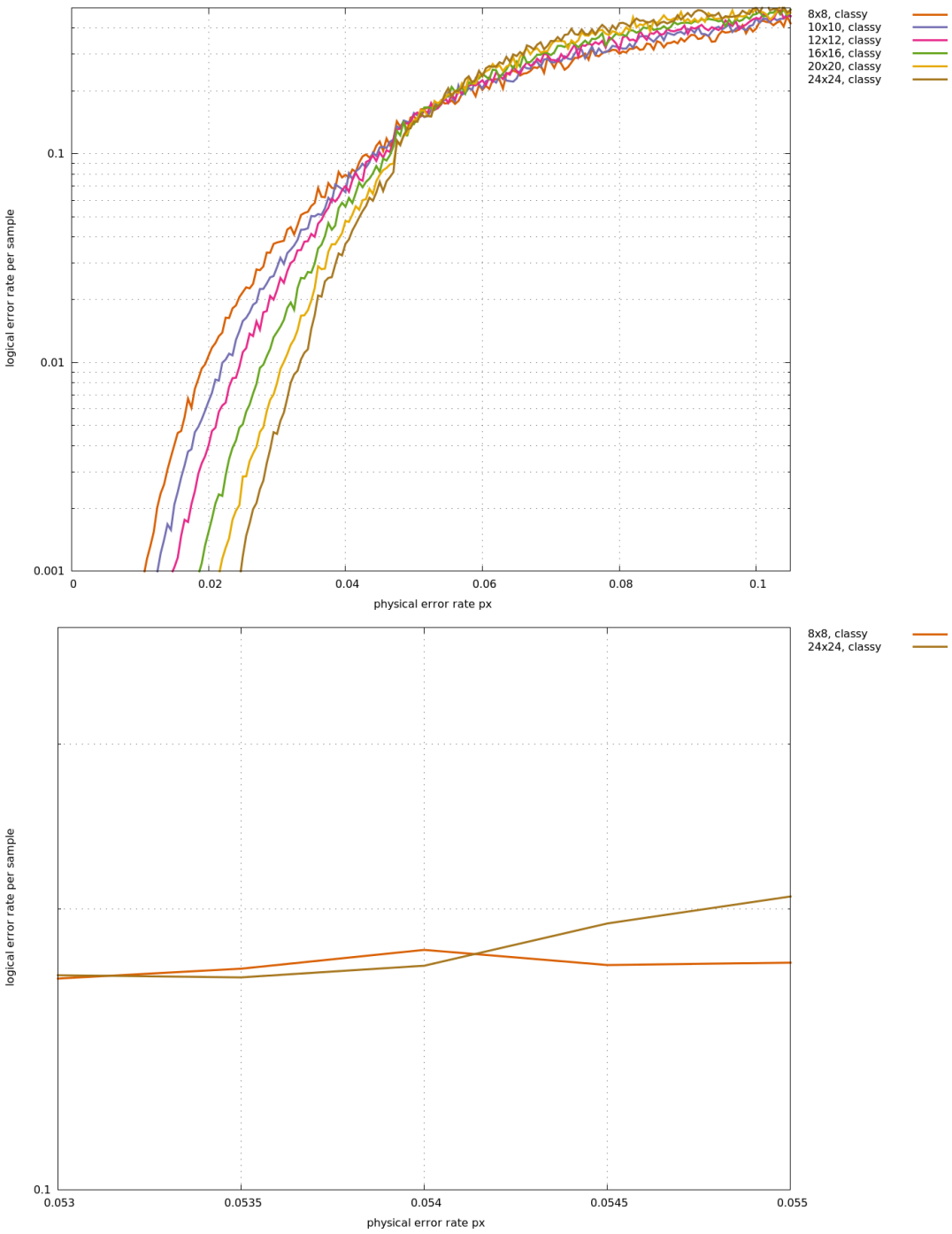

FIG. 23: Treshhold is obtained for: classy decoder, independent errors, logical errors X, two measurement rounds and a probability for measurement errors $p_m = 1\%$

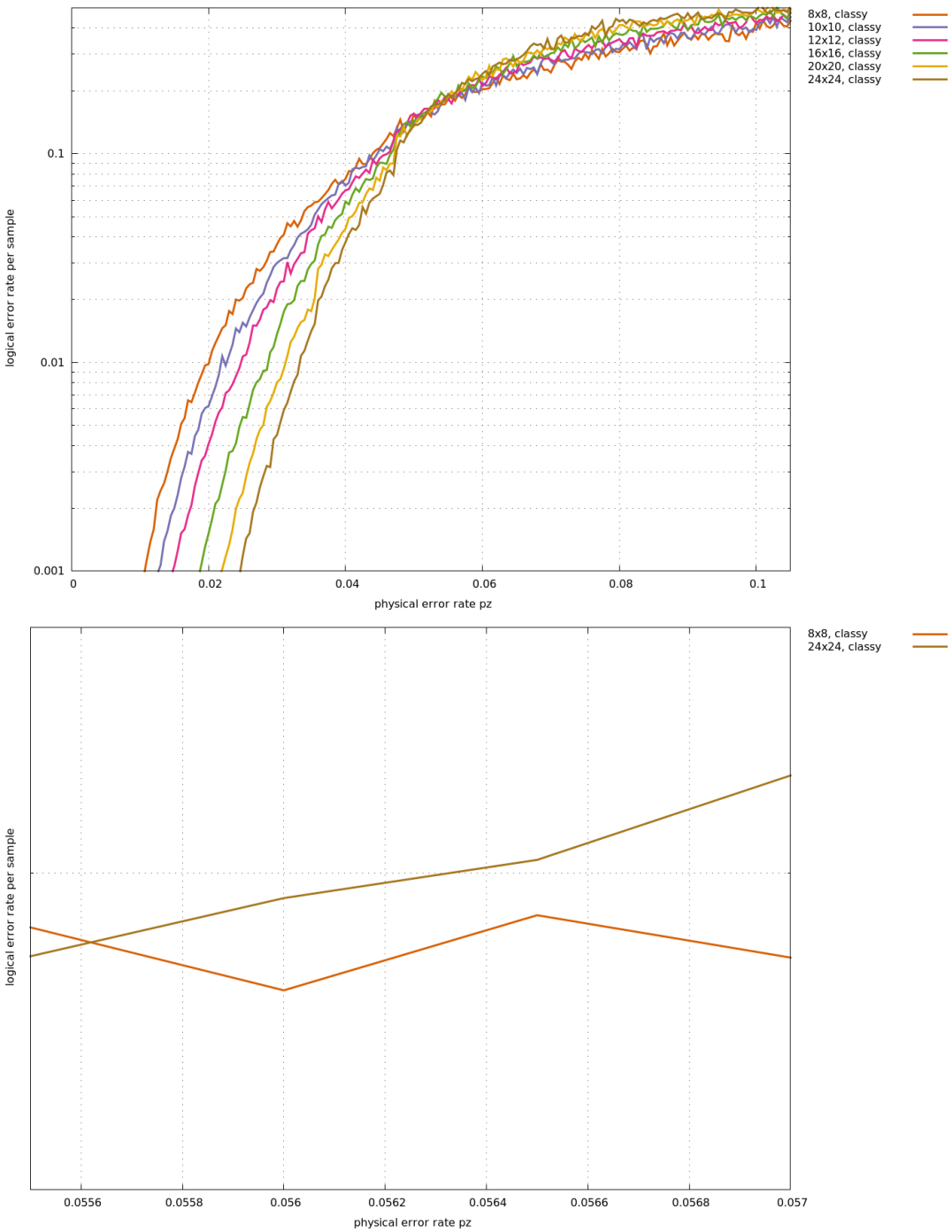

FIG. 24: Treshhold is obtained for: classy decoder, independent errors, logical errors Z, two measurements round and a probability for measurement errors $p_m = 1\%$

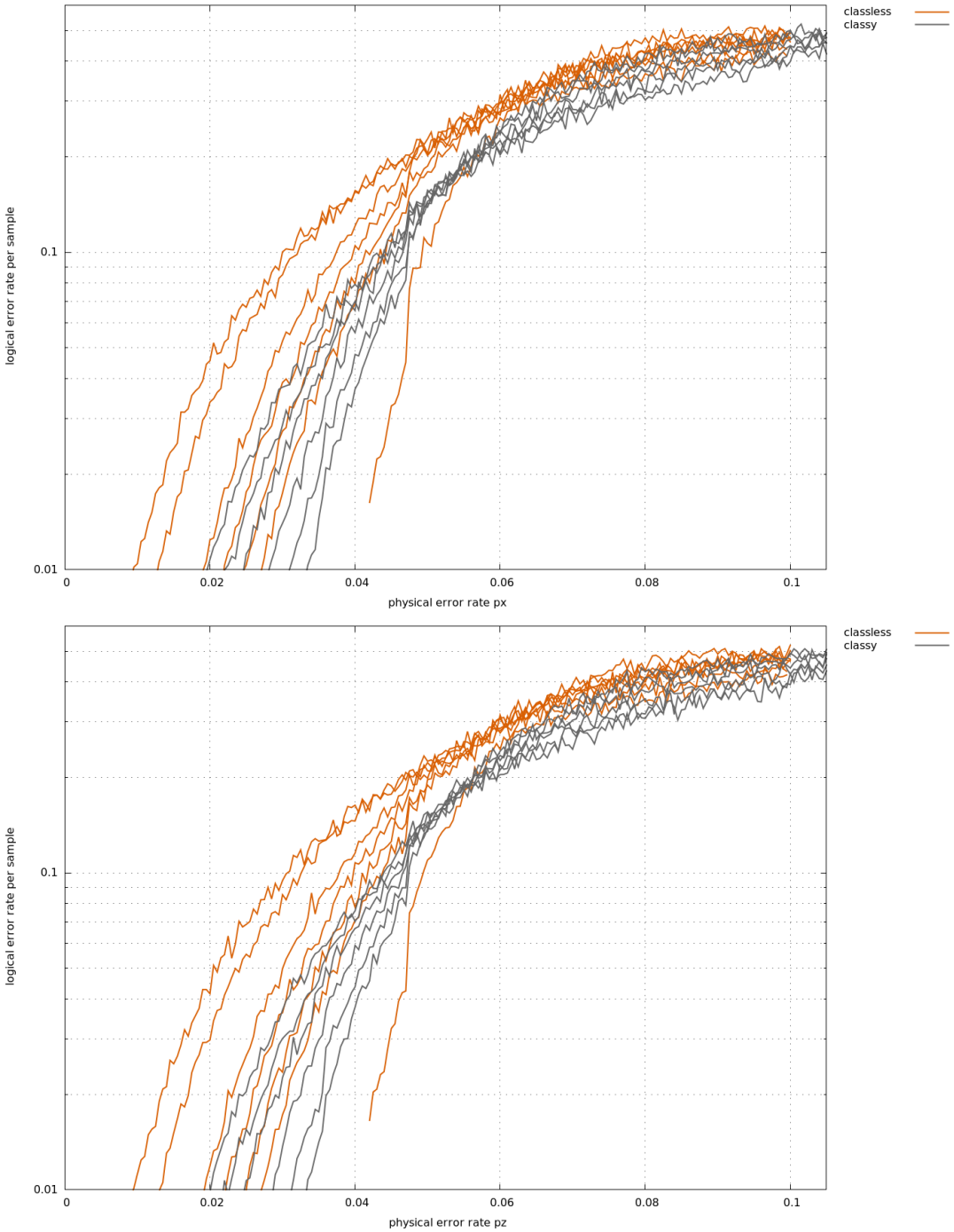

FIG. 25: Treshhold is obtained for both decoder, independent errors, logical errors X/Z, two measurements round and a probability for measurement errors $p_m = 1\%$

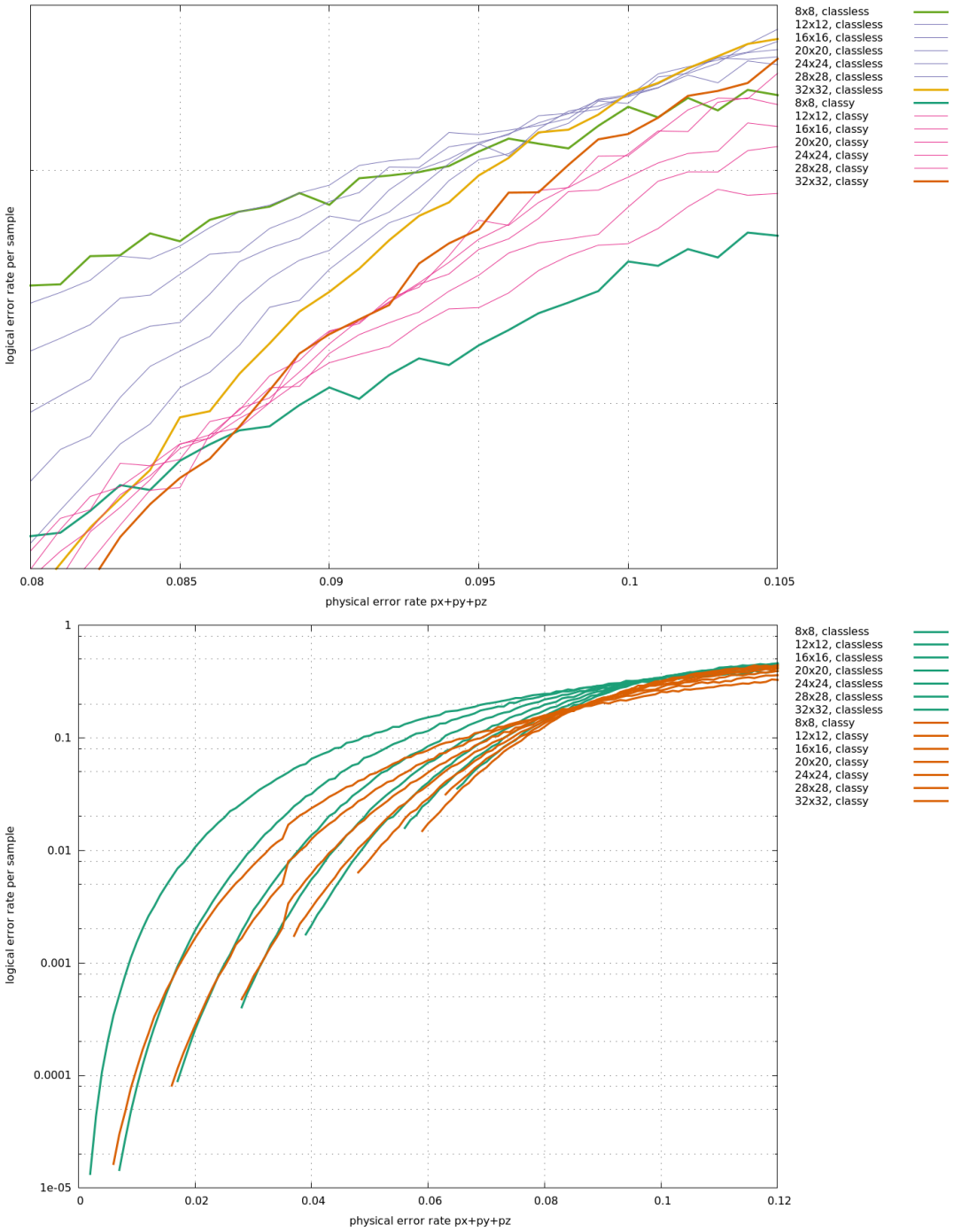

FIG. 26: Treshhold is obtained for both decoders, polarized noise, logical errors X/Z, two measurements round and a probability for measurement errors $p_m = 1\%$