Report

Pedro Parrado Rodríguez

PhD

Contents

1 Warm up

1.1 Lookup tables

A lookup table is the simplest decoder. It is a table in which we have the correction to apply for every syndrome.

To generate the lookup table for the simplest codes, I have explored all possible combinations of up to 5 errors. Then for each error combination, I find the syndrome that it generates, and I add it to the lookuptable if there is no other error configuration for that syndrome with less errors (meaning, we only keep the error configurations with the minimum weight).

Lookup tables are probably the fastest decoders, as they take just one step to decode a syndrome. The problem comes at generating them, as the size of the table grows exponentially with the number of stabilizers (2^N), and both filling that table and storing it becomes impossible after increasing the system size just a bit.

1.2 MonteCarlo simulations with Lookup tables

To check the efficiency of the lookuptables, I have run some MonteCarlo simulations. The simulation works with the following steps:

1. First, we generate an error configuration by throwing errors at each qubit with probability p. For each error appearing in the code, we apply a pauli operator X, Y or Z with probability 1/3.

2. Second, we measure the syndrome for the system with that error configuration.

3. Then, we apply the correction from the lookuptable for that syndrome.

4. Finally, we check the combination of the error and our correction, to see if there is a logical error or not in the final result.

From each of those simulations we get either a 1 in case of logical error, or a 0 in case of a proper correction. We then repeat this simulation several times (of the order of 10^4 at least) to find the probability of logical error, and we compute that probability for different values of p (the probability of an error in a single qubit) to find the behavior of the probability of logical error, and to find the threshold (when $p_{Logical\quad error} = p_{error\quad in\quad a\quad qubit}$).

2 Surface Code

The surface code is an algorithm to encode quantum information in several physical qubits. It is a stabilizer code, in which we have the qubits arranged in the edges of a square lattice, the Z stabilizers in the faces (acting over the 4 qubits on the sides), and the X stabilizers in the sites (acting also over the 4 qubits corresponding to the edges that cross that site).

The surface code corresponds to the family of topological codes. It can be encoded as a toric code, if we are allowed to introduce periodic boundary conditions. In that case, we encode 2 logical qubits, and its logical operators are vertical and horizontal chains of pauli operators. Otherwise, we can use rough and smooth boundaries to use the code without periodic boundary conditions. In that case, we only encode one logical qubit, and the logical operators are also chains of pauli operators, that go from smooth - smooth borders for the X_L and from rough to rough borders for the Z_L.

As the stabilizers are either formed of only X or Z operators, it is a CSS code, so we can focus on just one of the 2 for the rest of the chapter.

2.1 MLE Decoder

In the surface code, an error produces two excitations of stabilizer measurements, so they are easy to detect. However, when we have a chain of errors, that only excites 2 stabilizers, one at the beginning and one at the end of the chain. That makes it more difficult to find the error, when we have several excitations in the code. The good thing is that error chains of the same topological class are equivalent up to stabilizers, so the goal of a decoder would be to find the most probable homology class for a given syndrome. That problem, though, is too hard to solve, but we can still do a very good approximation by finding the Most Likely Error (MLE Decoder).

As the error chains generate pairs of excitations, the way the decoder works is by finding the minimum weight perfect matching of the syndrome. To do that, we use the Blossom algorithm, from a C implementation made by Kolmogorov.

2.2 MonteCarlo simulation

Once we have the decoder working, our goal is to find the threshold of the surface code. This means, finding the critical value of p_c such that, below that value, we can improve the precision of the code as much as we want by increasing the code size.

The way we are going to do this is by Monte-Carlo simulations. By simulating error configurations for different values of p, we can compute an approximation to the probability of logical error, counting the percentage of cases in which after the correction there is a logical error in the code. Having this tool, we can compute the logical error probability around the critical point for different system sizes. The curves for each system size should cross each other at the critical point. Then, we can do a multiparameter fit to find the value of the critical point:

$$p_{LE} = A + B(p - p_c)L^{1/\nu} \tag{1}$$

2.3 Measurement errors Decoder

Up to this point, we have assumed that we can measure the syndromes reliably. However, that is not the case for a real experiment, and there is an error probability for a syndrome measure as well, which we will call q (remember that p is the error probability of a single qubit). Therefore, we will need to repeat the measurements several times in order to improve the probability of success of our decoder.

In general, for a code of size L we will measure the syndromes L times as well. That will transform our 2 dimensional graph into a 3D graph, where the vertical direction corresponds now to time. Vertex correspond to stabilizer measurement values, horizontal links to error in the qubits, and vertical links to errors in a syndrome measurement.

In the 2D version, we added nodes to that graph for the excitations of the stabilizers, and our decoder solved the problem by finding the Min. Weight Perfect matching that linked every node in that graph. We can make that algorithm work for the 3D case as well, if we include a small modification. Here, instead of adding nodes to the graph for the excitations of the stabilizers, we will add them in a different way: the nodes in the 3D graph correspond to the stabilizer measurements that are different from the measurement of the same stabilizer in the previous step.
For example, for a given stabilizer, if we measure in 6 different time-steps the following results: $+1, -1, -1, -1, +1, +1$, we will add 2 nodes to the graph for that stabilizer: one in the second time-step, and another one in the last time-step, because those were the points at which the syndrome changed from the previous step.

Therefore, this new perfect matching in the 3D graph will find the configuration with the minimum amount of errors, both qubit and measurement errors, compatible with the syndrome measurements. However, if the probability of error in the qubits is different to the probability of error in measurements, we have to assign different weights to the vertical and horizontal links. Those weights will be:

$$\log\left(\frac{1-p}{p}\right), \qquad \log\left(\frac{1-q}{q}\right).$$

We can introduce those weights into the MWPM algorithm, so that we find the most probable error configuration.

2.4 MonteCarlo simulation

As in the 2D case, we want to find the value of the critical point by using Monte-Carlo simulations. The simulations will give us approximations of the probability of logical error for different values of p, q and L, which will form different curves for different system sizes. Those curves should cross at the critical point. Therefore, by finding that crossing point, we will find an approximation for the critical point. To simplify, we will only consider X errors, and $p = q$.

3 Color Code

3.1 Rescaling decoder

This decoder is based on dividing the code in small groups of qubits (4 in our case). Then, decode those small groups of qubits, and assing a new qubit for each group. This process will effectively reduce the size of the code. If we apply this method iteratively, we can keep reducing the size of the code until it is small enough to be decoded with a lookuptable. At that point, we can translate the correction in the higher levels of the rescaling procedure to the lower levels, until we have again the entire code, and the correction associated with it.

3.1.1 Outline of the decoder

1. Read the Syndrome from the errors in the qubits.
2. Assign an initial splitting (or initial split probabilities).
3. Update the splittings (or the split probabilities) until convergence.
4. Decode the cells independently according to the splitting from the previous step.
5. Create the new rescaled code, and assign to each logical qubit the probability of a logical error in the cell.
6. Apply the decoder again, until the size of the code is small enough to apply a complete lookuptable.
7. Apply the corrections of the higher levels to the lower ones.

3.1.2 Hard splitting method

This algorithm for splitting is based on changing each individual splitting to a better one asuming the others are constant, until convergence. Therefore, the core of the algorithm is to compute the probability of a given splitting:

$$p(s_0^u|s_1 s_2 s_1' s_2') = \frac{p(s_0^u|s_1 s_2)p(s_0^l|s_1' s_2')}{p(s_0^u|s_1 s_2)p(s_0^l|s_1' s_2') + [1 - p(s_0^u|s_1 s_2)][1 - p(s_0^l|s_1' s_2']}, \tag{2}$$

where the conditional probabilities are:

$$p(s_0^u|s_1 s_2) = \frac{p(s_0^u s_1 s_2)}{p(s_0^u = 1, s_1 s_2) + p(s_0^u = 0, s_1 s_2)}, \tag{3}$$

where the probability $p(s_0^u s_1 s_2)$ is computed as the sum of the probabilities of all the error configurations which are compatible with the syndrome.

So, using equation 2 for a given stabilizer in the boundary of 2 cells, we can choose which splitting is the most probable, given the rest. Now the idea is to repeat this selection process for every slitting several times until we reach convergence, and that brings us to the next problem, which is how to update those splittings in a way that allows us to get to the best splitting.

Before going into the ways of choosing the updates, we are going to define some energy-like function to characterize the likelihood of a splitting. This energy should be minimum for the best splitting, and we define it as:

$$E(\{split\}) = -\sum_{cells} \log p(s_0 s_1 s_2). \tag{4}$$

Which is related to the probability of the error configurations that are compatible with the particular choice of the splitting. We will use this function to characterize the improvement in the choice of a splitting. These are some of those methods:

1. The first method would be to simply update all of the splittings in a fixed order. That can lead to cycles, due to the way of choosing the splittings.

2. The second way would be to update first the horizontal borders, then the verticals, and then the diagonal (the actual order of these 3 does not matter, just the fact that we update them in different steps). This can also lead to cycles, but we can parallelize this method as the updates of these 3 types of borders are independent.

3. The third way is to choose the next border to update at random. This solves the problem of the cycles, but also has the same problem of the other 2 methods, we cannot be sure that we reach the real minimum instead of a local minimum.

All of these methods have a problem that comes from the fact that, in each update of a single split, we always choose the one with the highest probability. That prevents us from exploring the whole space of options, and therefore can lead to the algorithm getting stuck in a local minimum of energy, because changing any single split would increase the energy, even though changing 2 at the same time could improve the energy.

One possible improvement to this method would be to allow a split to swap to a least probable option, in a thermal-montecarlo-like simulation. We could also swap clusters of splits at the same time. A different way of doing it is the Soft-splitting method.

3.1.3 Soft splitting method

This algorithm for splitting, instead of choosing fixed values for a set of splittings and explore the space by flipping splittings in the set, assigns a probability for each splitting. That means, we have now a function $p(s_0^u)$ for each splitting in the code instead of a value of 0 or 1. The key idea is that we now update those probabilities $p(s_0^u)$ according to the probabilities of the error configurations compatible with every possible splitting choice of the neighbouring splittings (s_1, s_2, s_1', s_2'), which means that we are now moving in a continuous way from one splitting choice to the other.

The way to update a particular splitting is by using the following equations:

$$p(s_0^u)_{t+1} = \frac{p(s_0^u)p(s_0^l)}{p(s_0^u)p(s_0^l) + [1 - p(s_0^u)][1 - p(s_0^l)]} \tag{5}$$

where those intermediate probabilities are computed by:

$$p(s_0^u) = \sum \frac{p(s_0^u s_1 s_2)p(s_1)p(s_2)}{p(s_0^u = 1 s_1 s_2) + p(s_0^u = 0 s_1 s_2)}, \tag{6}$$

$$p(s_0^l) = \sum \frac{p(s_0^l s_1' s_2')p(s_1')p(s_2')}{p(s_0^l = 1 s_1' s_2') + p(s_0^l = 0 s_1' s_2')}. \tag{7}$$

The probabilities $p(s_i)$ are the current values of the probability of each splitting, which are also to be updated in the next steps. The probabilities $p(s_0 s_1 s_2)$ are comp uted as before, by the probabilities of every different error configuration which is compatible with that choice of splittings.

As with the soft decoder, we can use different methods to choose the order of the updates of the splittings.

Once we have a good splitting, we can proceed to decode the cells independently of each other, by applying a decoder.

3.1.4 Cell Decoder

The decoding of a single cell of four qubits is simple. We have 3 values for the syndromes in the boundaries with the other cells s_0, s_1 and s_2; and 4 qubits. That means, we have a total of 2^3 possible syndrome measurements, and 2^4 possible error configurations, 2 per syndrome measurement. We can build a complete lookuptable with the possible configurations of errors.

Figure 1: 4 qubit cell, and the logical operator of that cell (shaded cells).

We can use the lookup table to compute the probability of each of the 2 possible corrections, knowing the probability of error of each individual qubit (each qubit contributes with its p_{err}, or $1 - p_{err}$).

After choosing the best option, the error probability of the renormalized cell is the probability of the other option that we have rejected. In other words, the probability of

s_0	s_1	s_2	Correction	Correction + Log. Operator
+	+	+	-	0,2,3
+	+	-	0,1	1,2,3
+	-	+	1,2	0,3
+	-	-	3	0,2
-	+	+	1,3	0,1,2
-	+	-	2	0,3
-	-	+	0	2,3
-	-	-	1	0,1,2,3

applying also the logical operator ($\hat{X}$), which will be the operation that we will apply if we decide to apply a correction in the next step:

$$p_{err} = \frac{p(e+\hat{X})}{p(e)+p(e+\hat{X})} \tag{8}$$

3.1.5 Decoder for the 18 qubit case

After each rescaling, we reduce the number of qubits by a factor of 4. In each of those processes, we simply decode groups of 4 cells and generate a reduced version of the codes. This process goes on until we reach a system size small enough to decode the entire code. That point is the 18 qubit case, and in this section we are going to see a couple of ways of decoding that final state.

The decoding process consists in using a lookup table. For the syndrome that we find in the code, we check in the table the possible error configurations consistent with that error configurations. Then, from there, we decide which correction we should apply. If the probability of error of every qubit was the same, we could have simply one option in the lookuptable for each syndrome, and then apply that one. However, as the rescaling procedure assigns different error probabilities to each cell after the rescaling, we need to take that into account. Therefore, this decoding procedure consists of 2 steps: First, the generation of the lookuptable, which only needs to be made once before using any decoder, and can be stored in a file; and second, the selection of the appropiate correction from the lookuptable for a given syndrome and set of error probabilities of the qubits.

The generation of the lookuptable is simple, but before explaining how it is generated, let us clarify how it is supposed to work. The table is a list of sets: the index of the table corresponds to the syndrome, and the set which is stored in that position of the table is the set of error configurations consistent with that particular syndrome. To put an example, lets say that, from the 9 stabilizers S_i, $i = 1, ...9$, we have measured an excitation on S_7.

We can write that state in binary, by assigning a 0 to nonexcited stabilizers and a 1 to excited ones:

$$S_0S_1S_2S_3S_4S_5S_6S_7S_8 = 000000010$$

We can read that as a number in binary, which corresponds to number 3. Therefore, if we measure that syndrome, we will check the position 3 in our lookuptable. And, in that position of the list, we will find a set of the possible error configurations consistent with that syndrome in particular. To indentify the error configurations, we will also write them in binary (1's for the qubits with errors, and 0's for the rest), such that an error configuration consisting in errors in the qubits 2,3 and 5 would be written as:

$$E = 001101000000000000$$

Now that we know how to read the lookuptable, let us see how to generate it: we generate all 2^18 possible error configurations, by simply generating all binary numbers from 0 to 2^18. Then, for each error configuration, we compute the syndrome that will be obtained from that error, which leads us to a position on the list (our lookup table, which starts being empty). We add the current error configuration to the set in that position in the lookuptable, and we keep exploring error configurations. After having explored all error configurations, we have the entire lookuptable ready to use.

It is important to note that, in the 2^9 (9 is the number of stabilizers in the 18 qubit code) positions in the lookuptable, the distribution of errors in the sets is not even. That is due to the fact that from the 9 stabilizers, only 7 are independent. Therefore, we will have some positions in the table for which we will have no error configurations, because those syndromes are incompatible with the model. And, for the rest of them (we will have exactly 2^7 positions with nonempty sets), we will have 2^11 different error configurations in each set. The reason for that is that we can always apply any combination of the stabilizers and the logical operators to the system and we will end up with the same syndrome. That makes $2^7 \cdot 2^4 = 2^11$ combinations for each syndrome.

Once we have the lookuptable, we need to be able to extract from it the correction that we should apply. The simplest way to do this is to find the error configuration with the maximum error probability. As we have the error probability for each qubit, what we do is to compute for each of the 2^11 error configurations that we find in the lookuptable for the given syndrome the correspondent probability. That probability is the product of a factor of either p_i or $(1-p_i)$ for each qubit if it has an error or not. Then, we simply need to find the maximum of that error probability, and apply that correction. This would be a MLE decoder (maximum likelihood error).

However, our goal with the error correction procedure is not only to return the system to the codespace, but also to correct the logical errors. Therefore, we could try to improve the procedure by computing, instead of the error configuration with the maximum likelyhood, the homology class with the most likelyhood. That means, computing the likelyhood of any case with no logical operators applied, with one of them, 2 of them, etc, and then apply the correction according to which of those scenarios is the most probable.

For this process of error correction, we need a new and improved lookuptable, in which for each syndrome we will have not one, but 2^4 different sets (the different combinations of the 4 logical operators). Each of those sets will correspond to one of the homology classes, and the decoder will have to compute the probability of each of those clases by adding the probabilities of each of those configurations. Then, we will apply the best correction from the best homology class. Let us see how to generate that improved lookuptable:

First of all, we will start with the information already available from the first lookuptable. For each non-empty position in the first lookuptable, we will pick just one of the error configurations (the rest are equivalent, either up to stabilizers or by applying some set of logical operators). That configuration, E_0, will be our starting point to fill the 2^4 sets in that position of the improved lookuptable. By applying the 2^7 possible combinations of stabilizers, we will fill the first set. Then, we will apply one of the 2^4 logical operator combinations (For example, X_{L1}) to E_0, and fill the set by applying again the 2^7 combinations of stabilizers to that new starting combination $\{X_L\}E_0$. After applying all 2^4 combinations of Logical operators, all the sets for that syndrome will be filled, and we can continue with the next non-empty position in the lookup table.

Although theoretically this second method should give a higher probability of success (understood as not having applyed any logical operator after the error correction), if the error probability of all qubits are equal, the monte-carlo simulations reveal an equal probability of success for both methods.

3.2 Details of the simulation

In this section we will specify some details of the code implementation, such as the method for indexing cells, qubits and syndromes.

3.2.1 Programs

The core of the simulation is contained in the file *colorCodeh.py*. It contains most of the functions and variables needed to simulate one color code. To make it work, it also needs the results from 18*qlookuptable.py*, which is the program in charge of generating the lookuptable for 18 qubits (both of the ones described before). That program only needs to be run once, and it will generate the lookuptables for the main program. Those 2 files are enough to simulate the color code and apply the decoder.

The other files are simply scripts to run monteCarlo simulations with the code, and to do controlled experiments with single functions from the main program for debugging purposes.

3.2.2 Coordinate system

We will store the information of the qubits, the stabilizers and the cells in arrays. To find the correspondence of a certain element in the array and the element in the code, we need a way to identify the elements. In order to do that, we have 2 systems of identifying them: the index in the array, and its spatial coordinates x and y.

For the stabilizers, the coordinates x and y correspond directly to the position of the stabilizer, as they are simply dots in the dual representation of the code. For the qubits, we have them represented as triangles, and those triangles are organized in squares. Our coordinate system will identify one qubit with the (x, y) coordinates of the bottom left corner of the square to which it belongs. To distinguish between the 2 qubits inside a square, we introduce a third "dimension" z. For the triangle in the bottom left of the square, we have $z = 0$, and we have $z = 1$ for the other. We identify the cells in a similar way.

To identify the stabilizers that are in the boundary between 2 cells, we have assigned s_0 to the horizontal boundaries, s_1 to the vertical and s_2 to the diagonal ones.

Finally, we need to find the equivalence between the spatial coordinates, and the index in the array. The following table has all that information [1]:

	Size	Index	Coordinates
Qubit	N	$i = 2x + 2Ly + z$	$x = (i/2)\%L$ $y = (i/2)/L$ $z = i\%2$
Stabilizers	N/2	$i = x + Ly$	$x = i\%L$ $y = i/L$
Cells	N/4	$i = x + Ly/2 + z$	$x = i\%L - i\%2$ $y = 2(i/L)$ $z = i\%2$

Table 1: Equivalence between coordinates

Once we have the coordinates of the elements, we need to be able to relate the coordinates of a cell with the coordinates of the qubits inside of it, and the coordinates of the stabilizers in the borders of it. We can find that relation easily in spatial coordinates, and then translate it to indexes using the previous table:

[1] It is important to note that the divisions in the table are understood as integer divisions, without decimals. For example, $7/3 = 2$ in that notation.

	z	=	0		z	=	1	
	q_0	q_1	q_2	q_3	q_0	q_1	q_2	q_3
x	0	0	1	0	1	1	0	1
y	0	0	0	1	1	1	1	0
z	0	1	0	0	1	0	1	1

Table 2: Coordinates of the qubits inside a cell, with respect to the cell coordinates. Remember that the spatial coordinates are referred to the bottom left corner of the square the qubit/cell is part of.

	z	=	0	z	=	1
	S_0	S_1	S_2	S_0	S_1	S_2
x	1	0	1	1	2	1
y	0	1	1	2	1	1

Table 3: Coordinates of the stabilizers in the boundaries of a cell with respect to the coordiantes of the cell.

3.3 Variables in the code

In the code, the variables are contained in 2 classes:

•Cell(i,L):

This class is an utility for the main class ColorCode. It contains the information about the indexes correspondent to the qubits and syndromes in the full code. The input to generate the cells is the index i of the cell in the full code, and the size of the code L. This is the information we can find inside of this structure[2]:

1. Cell[i].q=[q0,q1,q2,q3] contains the indexes(with respecto to the full code) of the four qubits inside cell i.
2. Cell[i].s=[s0,s1,s2] contains the indexes of the 3 syndromes in the boundaries.

In this way, the main code can simply call the decoder of a cell, and it will look for the appropiate values of the probabilities of error in single qubits, and the appropiate values for the syndromes s0,s1,s2, so that it can check the lookuptable.

•ColorCode(m,p):

The class ColorCode contains all the variables and functions needed for a simulation. The inputs to generate it are m, which is related to the system size by $N = 18 \cdot 2^{2m}$; and

[2]We write $Cell[i]$ because, in the code, we will have an array of cells.

p, which is the probability of error of the qubits[3]. The main functions in the class are the one for plotting, and the one that decodes the entire code. The class also contains the information of the error, the correction, the syndrome and the splitting. Let us start by describing the main functions in the class:

1. noise() generates X errors in the qubits with probability $p[i]$ for each qubit i. That probability is stored in a variable inside the class, and all $p[i]$ are initialized with the input value when you generate the code.

2. syndrome() measures the syndrome that corresponds to the actual error configuration.

3. hardDecoder(softsplit=True,plotall=False) generates a correction according to the measured syndrome.The variables of the input allow to choose between the soft-hard splitting methods, and to plot the intermediate steps. The function stores the correction information in the internal variable for the correction values. The way the decoder works is the following: 0. If the code has size m=0, applies the lookuptable decoder for 18 qubits, otherwise:

    (a) Applies the resplitting function (soft or hard) until convergence, or until 45 resplitting steps (first to happen).

    (b) Decodes each of the cells using the lookuptable for 4 qubits.

    (c) Creates a new code, with size m-1, and initializes it with the values of the corner syndromes (after applying the corrections from the cells to the measurements of those syndromes), and with the new probabilities of error of the qubits according to the probability of logical error of each cell.

    (d) Applies the function code.hardDecoder() of the new code. That means this process is applied recursively unitl the new code has size m=0, in which case the process finishes.

    (e) Translates the corrections in the higher levels of encoding to the lower levels, by applying the logical operator of the cells in which there is a correction.

4. plot(lattice=True,qubits=False,syndrome=True,correction=True, cells=False, error=True, splitting=False,coll='k') This functions plots the lattice. The variables in the input are there to activate the plotting of:

    (a) lattice: the lattice of the code, basic for everything.

    (b) qubits: it plots a q over every qubit. It is just for debugging.

[3]This probability can be changed manually after generating the code. In fact, you can assign a different error probability to each qubit in the code.

(c) syndrome: the values of the measurements of the syndromes. Plots a yellow ball over the -1 syndromes.

(d) correction: plots a C over the qubits with a correction.

(e) cells: plots the cells by writing the word CELL over every qubit in a cell, with a different colour per cell.

(f) error: plots an X over the qubits with error.

(g) splitting: plots the current splitting, (u for split[i]=0, d otherwise, and a + or - sign for each of the half splits)

(h) coll: is the colour used to plot the lattice.

The rest of the functions are mostly for the decoder to work. That includes the soft and hard splitting functions, several functions to compute probabilities of splittings, and the decoder for the 18 qubit case. Apart from those, there are other functions made for the debugging and for the testing of the algorithms:

1. energy(): measures the value of the "energy" of the current splitting, by computing the minus logarithm of the sum of the probabilities of the possible error configurations inside the cells assuming the current splitting.

2. fullsplittester(printon=False): this functions computes the energy of each and every possible splitting, and returns the minimum of that energy, and the corresponding splitting. The input variable printon makes it print the progress of the program (as it takes very long for codes of the size of m=1).

Now that we have covered the main functions, we can go through the different variables stored in the code:

1. m, N, and L are all 1D variables in the class. The first one is introduced as input, and the others follow $N = 18 \cdot 2^{2m}$ and $L = \sqrt{N/2}$.

2. p[N] is an array that contains the error probability of each qubit. It is initialized by the input variable p of the class, but can be changed at any moment.

3. e[N] is an array that, for each position i, contains a 1 if the qubit i has an error, or a 0 otherwise.

4. c[N] is similar to $e[N]$, but contains a 1 in the position i if there is a correction to be applied in that qubit. The decoder functions uses this variable to store the current state of the correction.

5. s[L^2] is an array that contains the values of the measured stabilizers. A 1 in the position i means that the stabilizer is excited. 0 otherwise.

6. split[L^2] is an array that contains the information about the splittings. There are 2 possible splittings, 0 or 1, for each stabilizer. The relation between that value and the half stabilizers is shown in table 7. This value is varied directly by the hard resplitting function, and by the value of the probability of each splitting choice which is varied by the soft resplitting function, depending on which one do we use. That probability of splitting is the next value in this list.

7. sp[L^2] is an array that contains, for each stabilizer, the probability of choosing the splitting 0 (or, equivalently, the probability $sp[i] = p(code.split[i] = 0)$).

	$s_i =$	+1	$s_i =$	-1
s_{plit}	s_i^u	s_i^d	s_i^u	s_i^d
0	0	0	1	0
1	1	1	0	1

Table 4: There are always 2 possible splittings for a giving stabilizer in the boundary between 2 cells. We label those 2 possibilities in the variable s_plit, which can have the values 0 or 1. The equivalence between the choice of splitting ($s_plit = 1$ or 2) and the actual values of the half splittings s^u and s^d are displayed in this table, where s_i is the value of the measurement of that stabilizer. The half splitting s^u corresponds to the cell with $z = 0$ and the other half splitting s^d corresponds to the cell with $z = 1$.

8. spt[2][2][2] contains the information from table 7. In other words, it returns the value of the half stabilizer given the syndrome, the split choice and the cell value of z. The first index refers to the syndrome value, and it is usually filled by using code.s[i]. The second is the split choice, which is usually filled by code.split[i]. The final index is the value of z, to know if we are looking for s^u or s^d, and it is usually filled with code.cell[i].z.

9. cells[N/4] contains a list of N/4 cells. Each element is an instance of class Cell, generated by code.cell[i]=Cell(i,L).

10. s0s[$L^2/4$], s1s[$L^2/4$], and s2s[$L^2/4$] are lists that contain the indexes of the stabilizers correspondent to horizontal, vertical and diagonal boundaries between cells, respectively.