PHYS 513: Final Project
Decoding Quantum Error Correcting Codes with a Quantum Annealer

Onkar Apte[1], Ammar Babar[1], Mario Morford-Oberst[1,2],
[1]*University of Southern California, Los Angeles, California 90089, USA,*
[2]*Center for Quantum Information Science and Technology, Los Angeles, USA.*

***Abstract*—We propose a decoder for quantum error correcting codes that converts the classical syndrome decoding problem into a QUBO ising spin chain problem. Authors in [1] developed an algorithm for mapping the MWPM decoding problem into a spin chain problem for the 2D surface code. Our scheme is shown in Fig. 1**

I. Introduction

We propose to extend the work of the authors of [1], where they develop a decoding algorithm for the Surface Code by mapping the decoding problem to an Ising-type optimization problem. In [1], they use a Digital Annealer (DA) to solve the Ising-type problem and show that their construction can decode the Surface Code with better scaling than Simulated Annealing (SA) and than the well-known MWPM decoder. Furthermore, their proposed scheme gives a very competitive code threshold, 9.4%-9.8%. They perform all of their analysis under a simple single physical error rate model, as is often done in the literature. They claim that this construction is more general, however, and that it can be applied to all stabilizer error correction codes. Unfortunately, they do not discuss in any detail how this extension can be performed. We extend their work in two ways. First, we formally define a way to extend the construction of the many-body Ising Hamiltonian (the *decoding* Hamiltonian) for arbitrary stabilizer codes. Second, instead of performing decoding using a DA, we will use D-Wave, a real quantum device, to solve the Ising-type decoding problem. Here, we use our new definition to solve the decoding problem for all classes of stabilizer codes with the aim of demonstrating the universality of this decoding approach.

Our report is structured as follows. In Section II, we generate the motivation for the need for decoding (Sec. II-A1), talk about current state-of-the-art methods for decoding (Sec. II-A2) and briefly discuss the prior work done on building a decoder using QUBO formulation (Sec. II-B). In Section III, we improve on the prior work (Sec. III-A) and discuss the results (Sec. III-B). In the last Section, Section IV, we conclude and discuss what our future plans are and how we can improve on what we have presented. Appendices A and B go over explicit examples, Appendix C discusses the results and Appendix D contains all the code required to reproduce our results.

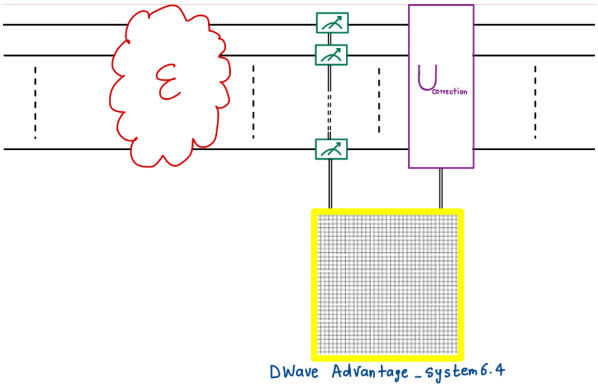

Fig. 1: We have our gate model quantum computer. Noise ε acts on it, and some errors occur. We perform the syndrome measurement and send the syndrome to our decoder, which is the `DWave Advantage_system6.4` quantum annealer. The decoder tells us the most probable error for that syndrome, and then we perform the correction operation $U_{\text{correction}}$ accordingly.

II. Background

A. Syndrome Decoding

You encode k-qubits into n-qubits. Then, some error occurs on your n-qubits. You measure $n-k=r$ different stabilizers and get a r-bit long binary string. This is the syndrome that contains the information about which error occurred. The problem of actually finding out which error has occurred from this r-bit string is the decoding problem.

*1) **Why decoding is an issue?:*** Consider $[[7,1,3]]$ Stean code, for example. It has stabilizers

$$S_1 = IIIXXXX\,, \tag{1}$$
$$S_2 = IXXIIXX\,, \tag{2}$$
$$S_3 = XIXIXIX\,, \tag{3}$$
$$S_4 = IIIZZZZ\,, \tag{4}$$
$$S_5 = IZZIIZZ\,, \tag{5}$$
$$S_6 = ZIZIZIZ\,. \tag{6}$$

So, the parity check matrix H for this code will be (6×14) and is given as:

$$H = \left[\begin{array}{ccccccc|ccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right] = \left[\begin{array}{c} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \end{array}\right]. \quad (7)$$

We have in total $2^r = 2^{n-k} = 2^6 = 64$ different syndromes. Hence, there are 64 different errors that we can correct. Given the r-bit syndrome, we need to figure out which error occurred. We can create a look-up table for that. The way we create a look-up table is just by multiplying the binary-symplectic vector corresponding to the error with the H matrix, and it should result in a 6-bit syndrome. For example, consider error Y_1. Its binary symplectic representation is

$$Y_1 = \left[1\ 0\ 0\ 0\ 0\ 0\ 0 \mid 1\ 0\ 0\ 0\ 0\ 0\ 0\right]. \quad (8)$$

Hence, the syndrome for this error is given by

$$\vec{s_{Y_1}} = H \cdot Y_1, \quad (9)$$

$$= \left[0\ 0\ 1\ 0\ 0\ 1\right]. \quad (10)$$

Similarly, in this manner, we start from the lowest weight error, perform this multiplication one by one and create a look-up table which contains 2^6 syndromes and indicates which error each syndrome corresponds to. Even though look-up is $O(1)$, creating this lookup table is a non-trivial task.

As r increases, you can see that the number of syndromes increases exponentially, and then generating the look-up table becomes an unviable option very soon.

Consider $[[228, 12, 18]]$ QECC for example. Here, $r = 228 - 12 = 216$. Hence, the look-up table will contain 2^{216} different entries, and to create each entry, you would need to perform matrix-vector multiplication of size 216×24. Assume that somehow you can do this matrix-vector multiplication magically in 10^{-15} seconds. Even in that case, you would need 10^{45} years to construct this look-up table, which is completely impractical. And even if, somehow, some magical wizard gave you this table, you would need 10^{50} Terrabytes of data storage, which is also impractical. This is the reason why decoding is an issue, and we need to employ some clever decoding methods.

*2) **Current state-of-the-art methods**:* As can be gleaned from the previous section, developing decoders is as much an art as it is a science. As a general rule, there is no one-size-fits-all for different families of code. In the interest of brevity (and relevance), we summarize the performance of different decoders available for quantum low-density parity-check (qLDPC) codes. These codes, owing to their ability to correct a significant number of errors greater than the theoretical distance of the code and low-weight stabilizer measurements that can be measured in parallel with shallow circuits [2], are widely regarded as the best practical candidates for building resilient quantum processors. It is important to note that qLDPC codes are a family of codes with many different realizations, all of which share the key property of having geometrically local stabilizers.

In the table below, we share the performance of 3 widely regarded & used decoders for topological codes [3] [4]:

Decoder type	Asymptotic performance
Perfect decoder	$O(e^n)$
MWPM	$O(n^3)$
BP+OSD	$O(n^3)$
Union-Find	$O(\alpha(n)n)$

Here, $\alpha(n)$ is the inverse of Ackermann's function that grows very slowly (i.e. for all practical purposes, it can be treated as a constant) [5].

It is important to note that the asymptotics only show a limited picture. While MWPM ((Minimum Weight Perfect Matching) and BP+OSD (Belief Propagation + Ordered Statistics Decoding) have the same asymptotic performance, in practice, BP+OSD performs better for qLDPC codes since it excels at handling the degeneracy in the code (where MWPM might get stuck) [4]. Furthermore, while Union-Find's asymptotic performance is impressive, a significant amount of work needs to be done to make it practically competitive, since it is a very new decoder [5].

Each of the decoders discussed above is being worked on by huge teams of some of the brightest minds in the field of error correction. We hope that this brief discussion highlights how challenging it is to craft decoders for quantum codes and the utility of a universal, fast decoder.

*B. **Prior work on using annealing for decoding surface code***

In [1], a Digital Annealer (DA) is used to solve the decoding problem for the surface code. The authors give a formulation to define a many-body Ising-type Hamiltonian for the purpose of decoding the surface code undergoing Z errors. Finding the ground state of this Hamiltonian is then equivalent to having found the most likely error to have occurred based on the syndrome. In their paper, they analyze their DA decoder's performance by defining a noise model and comparing the performance of the decoder to some of the most well-known and best-performing decoders for the surface code. In order to justify their decoder's abilities, they look at two metrics. First, they look at how many annealing samples need to be taken to find the ground state of the Hamiltonian as a function of the physical error rate. Second, they look at the logical error rate of the code as a function of the physical error rate. These two metrics provide valuable information about the performance of the decoder. The first metric looks at how quickly their decoder can generate an

answer to the decoding problem, and the second metric looks at how accurate this answer is.

As mentioned earlier, many decoders have been proposed. However, they all face different challenges in being practical answers to the decoding problem. For instance, the MWPM decoder is a promising decoder for the surface code, but the complexity of this decoding algorithm would make it difficult for the classical hardware that is inside the fridge with the quantum computer to be able to run the algorithm. As a result, this option would require improvements in classical computing to make the MWPM algorithm a viable option for practical use. The DA decoder is advertised as an all-in-one solution that already has existing hardware to support the underlying decoding algorithm. However, it is also unclear how easily this architecture would integrate with a gate-based quantum computer since it cannot be placed inside of the dilution refrigerator (DR).

To get a better sense of how the DA decoder works, let's get more familiar with the context. Annealers are computing devices that can be used to solve optimization problems. Specifically, they can be used to find the ground state of Ising Hamiltonians. Informally, they do this by initializing in the ground state of a known Hamiltonian. As the computation progresses, internal "dials" are turned such that the physical system being simulated by the annealer is changed from being the initial physical system to that of the target physical system (i.e. the system for which we wish to find the ground state). This happens over finite time, and so there is a possibility that the system does not find itself in the ground state of the target Hamiltonian. If this computation occurs in the infinite time limit, also known as the adiabatic limit, then the system will always occupy the ground state [6]. Since we live in a world where computation must take finite time, we can perform multiple annealing cycles and post-select the outcome by always selecting the outcome with the least energy. Assuming we had at least one successful trial, we will have found the ground state of the target Hamiltonian.

We might now ask ourselves how we can solve the decoding problem using an annealing device. The answer is quite simple. If we can construct a Hamiltonian that has a ground state that gives the answer to the decoding problem, then we can use an annealing device to decode quantum error correction codes. The authors of [1] did exactly that by defining the construction of a many-body Ising-Hamiltonian for the surface code undergoing Z-type errors, which has a ground state that is the answer to the decoding problem. The Ising-type Hamiltonian defined in [1] is given as

$$H = -J \sum_{v}^{N_v} b_v \prod_{i \in \delta_v}^{4} \sigma_i - h \sum_{i}^{N_d} \sigma_i \,, \tag{11}$$

where
- J and h are some constants.[1]
- b_v is the syndrome corresponding to X-type stabilizer B_v.
- δ_v is the set of qubits on which stabilizer B_v acts.
- N_d and N_v are the number of data qubits and number of X-type stabilizers, respectively.
- σ_i is the i^{th} spin variable.

In order for the DA to solve this problem, the authors of [1] first needed to convert this (up to) four-body Hamiltonian into a QUBO problem, which only contains two-body terms. This conversion process involves four steps. First, the σ_i terms are converted into x_i terms using the following equation

$$x_i = \frac{(1 - \sigma_i)}{2} \,. \tag{12}$$

Second, the fourth-order terms in the Hamiltonian generated by the previous substitution will be used to define further substitution, creating z_k terms. The z_k terms are defined as

$$z_k = x_i x_j (0 \leq k < N_m) \,, \tag{13}$$

where N_m is the number of z_k terms necessary to convert four-body x terms into two body z terms. There is a subtlety here that needs to be closely followed. The original many-body Ising Hamiltonian can be split into mutually exclusive terms corresponding to each stabilizer measurement outcome, b_v. The z substitutions defined in the second step are only made in each respective term corresponding to each stabilizer measurement, b_v, after the substitution to x variables is made. Details about this can be more easily understood when looking at the example in Appendix A. Third, in order for the above equality to hold, a penalty Hamiltonian is required such that the ground state of the resulting Hamiltonian is preserved. Here, the penalty Hamiltonian is defined as

$$H_{\text{penalty}} = \alpha \sum_{m}^{N_m} [x_i x_j - 2 z_m (x_i + x_j) + 3 z_m] \,, \tag{14}$$

where α is a hyperparameter to control the strength of the penalty term[2]. The fourth step is to generate the full QUBO Hamiltonian by summing the newly converted Hamiltonian H (with x and z terms) together with the penalty, Hamiltonian H_{penalty}, and converting x_i and z_k into only y_i terms using the following conversion principle

$$y_i = \begin{cases} x_i \,, (0 \leq i < N_d) \\ z_i \,, (N_d \leq i < N_d + N_m) \end{cases} \,. \tag{15}$$

Using this final conversion, we get our full QUBO Hamiltonian H_{decoder} defined as

$$\begin{aligned} H_{\text{QUBO\_decoder}} &= H_{\text{QUBO}} + H_{\text{penalty}} \\ &= -\frac{1}{2} \sum_{i,j}^{N} W_{i,j} y_i y_j - \sum_{i}^{N} V_i y_i + c \,, \end{aligned} \tag{16}$$

where $N = N_d + N_m$, $W_{i,j}$ is an off-diagonal coefficient of the QUBO matrix, V_i is a diagonal coefficient of the QUBO matrix, and c is just a constant. We have thus constructed the QUBO formulation of the problem from the many-body

[1] J and h need to be choosen such that they satisfy $J > \left\lfloor \frac{d-1}{2} \right\rfloor \frac{h}{2}$, where d is the code distance. This ensures that there is always a sufficient energy gap between the ground state and the first excited state.

[2] In [1], α was set to $8J$.

Ising Hamiltonian, which can now be fed to the DA for decoding. The solution of the QUBO formulation will give a bitstring which indicates where errors occurred (if any), and the corrective step will then be to apply X gates to the qubits which correspond to the index of 1s in the solution bit string.

We believe that it is valuable to see if the implementation of a decoding algorithm based on Ising-type optimization on a quantum annealer gives any advantages over implementing it on a DA. Solving Ising-type problems using an annealer has proven to be successful historically, which is what gives us this motivation.

III. Our Work

A. *Building on Fujisaki et al. [1]'s work*

The formulation of the *decoder* Hamiltonian as it is given in [1], Eq. 11, is incomplete. As we discussed earlier, their formulation only accounts for X errors effecting the code, but this does not tell the full story. Realistically, quantum devices will experience all types of Pauli noise, so it is necessary to consider the general case where these errors occur. Furthermore, this formulation does not express in any detail how to extend the formulation to general stabilizer codes. We extend the formulation given in [1] in order to define a construction that makes it clear how to implement the Annealing Decoder for arbitrary stabilizer codes undergoing universal Pauli noise. We define the general formulation for H_{decoder} as

$$H_{\text{decoder}} = -J \sum_{i=1}^{N_s} \left(s_i \prod_{j \in \hat{S}_{i_z}} \sigma_j \prod_{k \in \hat{S}_{i_x}} \sigma_{k+N_d} \right) - h \sum_{i=1}^{2N_d} \sigma_i \,, \quad (17)$$

where
- J and h are the hyperparameters from footnote 1
- N_s is the total number of stabilizers for the code.
- $s_i \in \{+1, -1\}$ is the measurement outcome of stabilizer $\hat{S}_i$.
- $\hat{S}_{i_z}$ is the Z component of $\hat{S}_i$ in symplectic representation.
- $\hat{S}_{i_x}$ is the X component of $\hat{S}_i$ in symplectic representation.
- N_d is the number of data qubits in the code.

This H_{decoder} will be a $2^{2N_d} \times 2^{2N_d}$ matrix. All binary vectors of length $2N_d$ will be eigenstates of this Hamiltonian. However, only one will correspond to the lowest energy, which will tell us what the most probable error is that occurred depending on the syndrome $\vec{s} \equiv (s_1, s_2, \cdots s_i)$. Let $|\psi_g\rangle$ be the ground state of H_{decoder} for a given $\vec{s}$.

$$|\psi_g\rangle = \bigotimes_{i=1}^{2N_d} |b_i\rangle \,, \quad b_i \in \{0, 1\} \,, \quad (18)$$

$$= |b_1 b_2 \cdots b_{N_d} b_{N_d+1} \cdots b_{2N_d-1} b_{2N_d}\rangle \,. \quad (19)$$

Let the error guessed by the decoder be E. The way we get E from the ground state is as follows:

$$E = \bigotimes_{i=0}^{N_d} Z^{b_i} X^{b_{N_d+i}} \,. \quad (20)$$

Please see Appendix B for an explicit example.

B. *Results*

In order to comprehensively explore how this extended formulation works, we used our new formulation for a variety of error-correcting codes. These codes belong to various stabilizer code families and show that the new formulation works for a diverse set of code families. The families we consider are 1D linear block codes, non-CSS codes, symmetric CSS codes, asymmetric CSS codes, and 2D topological codes. Our testing was conducted in two parts. First, we tested using classical devices, and second, we tested using D-Wave's quantum annealer.

*1) **Testing on a Classical Machine**:* Classical testing was performed to show proof of concept prior to attempting to use quantum devices to solve the decoding problem. Here, we test our formulation of the decoder Hamiltonian, given in Eq.(17), and check it's validity by constructing the explicit $2^{2n} \times 2^{2n}$ Hamiltonian matrix for every syndrome generated by each correctable error for the given code. We then use classical software to diagonalize the Hamiltonian and check if the ground state corresponds to the correct error.

We have successfully shown this for all correctable errors for the following codes:
- $[[3, 1, 3]]$ Bitflip code
- $[[5, 1, 5]]$ Bitflip code
- $[[5, 1, 3]]$ Perfect code
- $[[7, 1, 3]]$ Color code
- $[[9, 1, 3]]$ Shor code
- $[[9, 1, 3]]$ Surface code

This set of QEC codes, even though small in the number of qubits, does encompass different families and gives the proof-of-concept working for 1D linear block codes, symmetric CSS codes, asymmetric CSS codes, non-CSS codes and topological codes such as surface codes and colour codes. Readers can find code snippets performing this task in the Appendix D.

*2) **Testing on DWave Annealer**:* Testing of our scheme was performed on D-Wave's `Advantage_System6.4` quantum annealer. In order to use D-Wave to solve the decoding problem, we first generate the decoder Hamiltonian using Eq.(17), and then convert it into a QUBO problem using Eqs.(12)-(16). After the conversion is complete, we feed the QUBO decoder Hamiltonian to D-Wave to solve using annealing shots. The resulting data is then interpreted using Eqs.(18)-(20). In order to perform experiments with many different codes and large code families, we worked to develop a code library that automates the generation of Eq.(17) and the conversion to the QUBO problem, Eqs.(12)-(16). The latter proved to be quite involved and is not yet robust. As a result, we generated the QUBO decoder Hamiltonian for the $[[5, 1, 5]]$ bit flip code by hand for the purpose of showing that our scheme works. We continued work on the code base to increase its robustness, but it is still in progress. As a result, we currently do not have further tests using D-Wave for larger codes.

*a) **Syndrome decoding for** $[[5, 1, 5]]$ **bitflip code on D-Wave annealer**:* Our D-Wave tests involving the $[[5, 1, 5]]$ code consisted of constructing the decoder Hamiltonian using

Eq. (17) and then constructing the respective QUBO decoder Hamiltonian using Eqs. (18)-(20) for each each correctable bit flip error. The $[[5,1,5]]$ code can correct any single or two-qubit bitflip error, which means there are a total of 15 correctable errors this code can handle. For each correctable error, we constructed the respective QUBO decoder Hamiltonians and fed them to D-Wave. We looked at the results for each Hamiltonian and were able to conclude that the correct correction was found for each syndrome, indicating that we successfully decoded a quantum error correction code with a quantum annealing device. Please refer to the appendices C and D for the results and the code that implemented this experiment, respectively.

IV. Conclusion & Future Scope

Our work shows that the syndrome decoding problem can be mapped to QUBO s.t. the ground state of the QUBO Hamiltonian corresponds to the correct error that generated the syndrome. We demonstrated, by mapping our QUBO problem to the `DWave Advantage_system6.4` system, that our scheme can correct all weight $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors for an $[[n,k,d]]$ CSS code, and weight $\left\lfloor \frac{d-1}{2} \right\rfloor$ X and Z errors for an $[[n,k,d]]$ non-CSS code.

We claim our construction serves as a universal decoder for CSS stabilizer codes since we do not appeal to any properties or structure of a code except for requiring the code to have a CSS stabilizer representation.

Since annealers are very good at giving a decent answer very fast, they are promising candidates for decoding errors on quantum processors in the regime of >100k+ qubits. Additionally, with advances in photonic interconnect technology (actively being developed for quantum device applications), we could hope to see both annealers and QPUs integrated into a single system, potentially eliminating the need for notorious noisy classical channels that pose a significant engineering hurdle.

While decoding with annealers seems promising theoretically, due to the runtime of an annealer (order of microseconds), it makes it practically useless for superconducting qubit architectures. DWave did recently announce a fast anneal method (order of nanoseconds), but we do not know of its accuracy running our algorithm and would need to benchmark it.

There are a number of outstanding questions and challenges that need to be overcome before this scheme can be practically relevant. For one, we have not established a theoretical (asymptotic) bound for our decoder. It is not immediately obvious how the number of shots required to find the ground state of QUBO Hamiltonian will scale for larger codes. A new insight is required to extend our decoder to correct all errors (not just X and Z) for non-CSS codes. Additionally, while it seems plausible that our scheme can scale favourably to tackle larger codes, it is not obvious how the QUBO conversion and mapping blows up the number of annealing qubits required to run our algorithm. Of course, further optimizations of the decoder for specific codes would be extremely useful for practical considerations, and it would also be worthwhile to explore how techniques from learning and noise mitigation can be leveraged to tailor the QUBO Hamiltonian for a particular device. It would be really interesting to see if this can be done in real-time (with the help of the annealer) as the noise propagates through the QPU.

V. Contribution

All authors have contributed equally in terms of research, efforts, and implementation, except programming. For coding, OA handled classical checking of the validity of our proposed decoder formulation. MO handled performing the D-Wave runs, and AB handled writing the code to automate everything.

References

[1] Jun Fujisaki et al. "Practical and scalable decoder for topological quantum error correction with an Ising machine". In: *Phys. Rev. Res.* 4 (4 Nov. 2022), p. 043086. DOI: 10.1103/PhysRevResearch.4.043086. URL: https://link.aps.org/doi/10.1103/PhysRevResearch.4.043086.

[2] *Quantum low-density parity-check (QLDPC) code*. 2022. URL: https://errorcorrectionzoo.org/c/qldpc.

[3] Naomi Nickerson Nicolas Delfosse. *Almost-linear time decoding algorithm for topological codes*. https://qutech.nl/wp-content/uploads/2018/02/w5-Naomi-Nickerson-wednesday_nickerson.pdf. 2018.

[4] Pavel Panteleev and Gleb Kalachev. "Degenerate Quantum LDPC Codes With Good Finite Length Performance". In: *Quantum* 5 (Nov. 2021), p. 585. ISSN: 2521-327X. DOI: 10.22331/q-2021-11-22-585. URL: https://doi.org/10.22331/q-2021-11-22-585.

[5] Nicolas Delfosse and Naomi H. Nickerson. "Almost-linear time decoding algorithm for topological codes". In: *Quantum* 5 (Dec. 2021), p. 595. ISSN: 2521-327X. DOI: 10.22331/q-2021-12-02-595. URL: https://doi.org/10.22331/q-2021-12-02-595.

[6] Tameem Albash and Daniel A. Lidar. "Adiabatic quantum computation". In: *Reviews of Modern Physics* 90.1 (Jan. 2018). ISSN: 1539-0756. DOI: 10.1103/revmodphys.90.015002. URL: http://dx.doi.org/10.1103/RevModPhys.90.015002.

[7] *GitHub Repo for the program to perform the annealing given stabilizers and a syndrome*. 2024. URL: https://github.com/ababar-usc/annealing_decoder.

APPENDIX-A: EXPLICIT EXAMPLE FOR [[5, 1, 5]] BITFLIP REPETITION CODE

Let's choose the following stabilizer generators for the [[5, 1, 5]] Bitflip repetition Code:

$$S_1 = Z_1Z_2Z_3Z_4\,, \tag{21}$$
$$S_2 = Z_2Z_3\,, \tag{22}$$
$$S_3 = Z_3Z_4\,, \tag{23}$$
$$S_4 = Z_2Z_3Z_4Z_5\,. \tag{24}$$

Let the syndrome for this code be $\vec{s} = (s_1, s_2, s_3, s_4)$, where s_i are the measurement outcomes +1 or -1 on measuring stabilizer S_i. Now, we can construct the many-body decoder Hamiltonian as follows, following the Eq.(17):

$$H = -J(s_1\sigma_1\sigma_2\sigma_3\sigma_4 + s_2\sigma_2\sigma_3 + s_3\sigma_3\sigma_4 + s_4\sigma_2\sigma_3\sigma_4\sigma_5) - h(\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4)\,. \tag{25}$$

Now, we substitute σ_i with binary variables x_i. From Eq.(12), we get that

$$\sigma_i = 1 - 2x_i\,. \tag{26}$$

So now, our Eq.(25) becomes

$$\begin{aligned} H = &-J\big(s_1(1-2x_1)(1-2x_2)(1-2x_3)(1-2x_4) + s_2(1-2x_2)(1-2x_3) + s_3(1-2x_3)(1-2x_4) \\ &+ s_4(1-2x_2)(1-2x_3)(1-2x_4)(1-2x_5)\big) - h((1-2x_1) + (1-2x_2) + (1-2x_3) + (1-2x_4) + (1-2x_5))\,. \end{aligned} \tag{27}$$

Expanding out everything, we get

$$\begin{aligned} H = &-J\Big(s_1\big(1 - 2x_1 - 2x_2 - 2x_3 - 2x_4 + 4x_1x_2 + 4x_1x_3 + 4x_1x_4 + 4x_2x_3 + 4x_2x_4 + 4x_3x_4 - 8x_1x_2x_3 - 8x_1x_2x_4 - 8x_2x_3x_4 - 8x_1x_3x_4 \\ &+ 16x_1x_2x_3x_4\big) + s_2\big(1 - 2x_2 - 2x_3 + 4x_2x_3\big) + s_3\big(1 - 2x_3 - 2x_4 + 4x_3x_4\big) + s_4\big(1 - 2x_2 - 2x_3 - 2x_4 - 2x_5 + 4x_2x_3 + 4x_2x_4 + 4x_2x_5 \\ &+ 4x_3x_4 + 4x_3x_5 + 4x_4x_5 - 8x_2x_3x_4 - 8x_2x_3x_5 - 8x_3x_4x_5 - 8x_2x_4x_5 + 16x_2x_3x_4x_5\big)\Big) - h(5 - 2x_1 - 2x_2 - 2x_3 - 2x_4 - 2x_5)\,. \end{aligned} \tag{28}$$

Now, we make substitutions dictated by 4-body terms as shown in Eq.(13). In our case, substitutions are:

$$z_1 = x_1x_2\,, \tag{29}$$
$$z_2 = x_3x_4\,, \tag{30}$$
$$z_3 = x_2x_3\,, \tag{31}$$
$$z_4 = x_4x_5\,. \tag{32}$$

We make replacements as shown above in Eq.(28). We get

$$\begin{aligned} H = &-J\Big(s_1\big(1 - 2x_1 - 2x_2 - 2x_3 - 2x_4 + 4x_1x_2 + 4x_1x_3 + 4x_1x_4 + 4x_2x_3 + 4x_2x_4 + 4x_3x_4 - 8z_1x_3 - 8z_1x_4 - 8x_2z_2 - 8x_1z_2 + 16z_1z_2\big) \\ &+ s_2\big(1 - 2x_2 - 2x_3 + 4x_2x_3\big) + s_3\big(1 - 2x_3 - 2x_4 + 4x_3x_4\big) + s_4\big(1 - 2x_2 - 2x_3 - 2x_4 - 2x_5 + 4x_2x_3 + 4x_2x_4 + 4x_2x_5 + 4x_3x_4 + 4x_3x_5 \\ &+ 4x_4x_5 - 8z_3x_4 - 8z_3x_5 - 8x_3z_4 - 8x_2z_4 + 16z_3z_4\big)\Big) - h(5 - 2x_1 - 2x_2 - 2x_3 - 2x_4 - 2x_5)\,. \end{aligned} \tag{33}$$

And we have the penalty term, as defined in Eq.(14), which will be as follows:

$$H_{\text{penalty}} = \alpha\big(x_1x_2 - 2z_1(x_1 + x_2) + 3z_1x_3x_4 - 2z_2(x_3 + x_4) + 3z_2x_2x_3 - 2z_3(x_2 + x_3) + 3z_3x_4x_5 - 2z_4(x_4 + x_5) + 3z_4\big)\,. \tag{34}$$

So, our Decoder Hamiltonian in the QUBO form is

$$H_{\text{QUBO\_decoder}} = H + H_{\text{penalty}}\,. \tag{35}$$

At this point, we have a problem with the QUBO format. Now, we make the last round of substitutions and replace everything with variables y_i to make things uniform. This step is not necessary. We replace according to Eq.(15). Hence we have

$$
\begin{aligned}
y_1 &= x_1\,, & y_6 &= z_1\,,\\
y_2 &= x_2\,, & y_7 &= z_2\,,\\
y_3 &= x_3\,, & y_8 &= z_3\,,\\
y_4 &= x_4\,, & y_9 &= z_4\,,\\
y_5 &= x_5\,. &&
\end{aligned}
$$

So now, we have:

$$
\begin{aligned}
H_{\text{decoder}} = -J\Big(& s_1\big(1-2y_1-2y_2-2y_3-2y_4+4y_1y_2+4y_1y_3+4y_1y_4+4y_2y_3+4y_2y_4+4y_3y_4-8y_6y_3-8y_6y_4-8x_2y_7-8y_1y_7\\
&+16y_6y_7\big)+s_2\big(1-2y_2-2y_3+4y_2y_3\big)+s_3\big(1-2y_3-2y_4+4y_3y_4\big)+s_4\big(1-2y_2-2y_3-2y_4-2y_5+4y_2y_3+4y_2y_4\\
&+4y_2x_5+4y_3y_4+4y_3y_5+4y_4y_5-8y_8y_4-8y_9y_5-8x_3y_8-8y_2y_9+16y_8y_9\big)\Big)-h(5-2y_1-2y_2-2y_3-2y_4-2y_5)\\
&+\alpha\Big(y_1y_2-2y_6(y_1+y_2)+3y_6y_3y_4-2y_7(y_3+y_4)+3y_7y_2y_3-2y_8(y_2+y_3)+3y_8y_4y_5-2y_9(y_4+y_5)+3y_9\Big)\,.
\end{aligned}
\tag{36}
$$

Now, expanding out everything, we get the coefficients for the terms as follows:

$$
\begin{aligned}
y_1 &= 2Js_1+2h\,, && (37)\\
y_2 &= 2Js_1+2Js_2+2Js_4+2h\,, && (38)\\
y_3 &= 2Js_1+2Js_2+2Js_3+2Js_4+2h\,, && (39)\\
y_4 &= 2Js_1+2Js_3+2Js_4+2h\,, && (40)\\
y_5 &= 2Js_4+2h\,, && (41)\\
y_6 &= 3\alpha\,, && (42)\\
y_7 &= 3\alpha\,, && (43)\\
y_8 &= 3\alpha\,, && (44)\\
y_9 &= 3\alpha\,, && (45)\\
y_1y_2 &= -4Js_1+\alpha\,, && (46)\\
y_1y_3 &= -4Js_1\,, && (47)\\
y_1y_4 &= -4Js_1\,, && (48)\\
y_1y_7 &= 8Js_1\,, && (49)\\
y_2y_3 &= -4Js_1-4Js_2-4Js_4+\alpha\,, && (50)\\
y_2y_4 &= -4Js_1-4Js_4\,, && (51)\\
y_2y_5 &= -4Js_4\,, && (52)\\
y_2y_7 &= 8Js_1\,, && (53)\\
y_2y_9 &= 8Js_4\,, && (54)\\
y_3y_4 &= -4Js_1-4Js_3-4Js_4+\alpha\,, && (55)\\
y_3y_5 &= -4Js_4\,, && (56)\\
y_3y_9 &= 8Js_4\,, && (57)\\
y_4y_5 &= -4Js_4+\alpha\,, && (58)\\
y_6y_1 &= -2\alpha\,, && (59)\\
y_6y_2 &= -2\alpha\,, && (60)\\
y_6y_3 &= 8Js_1\,, && (61)\\
y_6y_4 &= 8Js_1\,, && (62)\\
y_6y_7 &= -16Js_1\,, && (63)\\
y_7y_3 &= -2\alpha\,, && (64)\\
y_7y_4 &= -2\alpha\,, && (65)\\
y_8y_2 &= -2\alpha\,, && (66)\\
y_8y_3 &= -2\alpha\,, && (67)\\
y_8y_4 &= 8Js_4\,, && (68)\\
y_8y_5 &= 8Js_4\,, && (69)\\
y_8y_9 &= -16Js_4\,, && (70)\\
y_9y_4 &= -2\alpha\,, && (71)\\
y_9y_5 &= -2\alpha\,. && (72)
\end{aligned}
$$

You can then solve this QUBO problem via any means to obtain the answer to syndrome decoding. The program to do this on a DWave machine is given in Appendix D.

APPENDIX-B: EXPLICIT H_{DECODER} GENERATION FOR [[5, 1, 3]] PERFECT CODE

Let's choose the following stabilizer generators for the [[5, 1, 3]] Perfect Code:

$$S_1 = XZZXI\,, \tag{73}$$
$$S_2 = IXZZX\,, \tag{74}$$
$$S_3 = XIXZZ\,, \tag{75}$$
$$S_4 = ZXIXZ\,. \tag{76}$$

Let the syndrome for this code be $\vec{s} = (s_1, s_2, s_3, s_4)$, where s_i are the measurement outcomes +1 or -1 on measuring stabilizer S_i . Now, the σ_i for decoder ising Hamiltonian will be according to Eq.(17).

$$\begin{aligned} Z_1 &= \sigma_1\,, & X_1 &= \sigma_6\,,\\ Z_2 &= \sigma_2\,, & X_2 &= \sigma_7\,,\\ Z_3 &= \sigma_3\,, & X_3 &= \sigma_8\,,\\ Z_4 &= \sigma_4\,, & X_4 &= \sigma_9\,,\\ Z_5 &= \sigma_5\,, & X_5 &= \sigma_{10}\,. \end{aligned}$$

Hence, now, the decoder Ising Hamiltonian will be as follows:

$$H = -J\Big(s_1\sigma_6\sigma_2\sigma_3\sigma_9 + s_2\sigma_7\sigma_3\sigma_4\sigma_{10} + s_3\sigma_6\sigma_8\sigma_4\sigma_6 + s_4\sigma_1\sigma_7\sigma_9\sigma_5\Big) - h\big(\sigma_1 + \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4 + \sigma_5 + \sigma_6 + \sigma_7 + \sigma_8 + \sigma_9 + \sigma_{10}\big)\,. \tag{77}$$

The ground state of this Hamiltonian will be as follows

$$|\psi\rangle_{\text{ground}} = |\underbrace{b_1b_2b_3b_4b_5}_{x_j}\,\underbrace{b_6b_7b_8b_9b_{10}}_{z_j}\rangle\,, \quad b_i \in \{0, 1\}\,. \tag{78}$$

We would always get some bitstring $\vec{b}$, as the ground state. This bitstring is nothing but the binary symplectic vector representation of the Pauli error that has occurred. Say

$$(x_1x_2x_3x_4x_5|z_1z_2z_3z_4z_5) := (b_1b_2b_3b_4b_5|b_6b_7b_8b_9b_{10}) \equiv \vec{b}\,. \tag{79}$$

Let the error that has occurred be E. This E is given as follows from the $\vec{b}$:

$$E(\vec{b}) = Z^{z_1}X^{x_1} \otimes Z^{z_2}X^{x_2} \otimes Z^{z_3}X^{x_3} \otimes Z^{z_4}X^{x_4} \otimes Z^{z_5}X^{x_5}\,. \tag{80}$$

For example, if $\vec{b}$ = (01000|00100), then

$$E = Z^0X^0 \otimes Z^1X^0 \otimes Z^0X^1 \otimes Z^0X^0 \otimes Z^0X^0\,, \tag{81}$$
$$= IZXII\,, \tag{82}$$
$$= Z_2X_3\,. \tag{83}$$

For example, if $\vec{b}$ = (00001|00001), then

$$E = Z^0X^0 \otimes Z^0X^0 \otimes Z^0X^0 \otimes Z^0X^0 \otimes Z^1X^1\,, \tag{84}$$
$$= Z_5X_5\,, \tag{85}$$
$$\approx Y_5\,. \tag{86}$$

And then, accordingly, you perform the correction operation.

APPENDIX-C: RESULTS

Syndrome decoding for $[[5,1,5]]$ ***bitflip code on D-Wave annealer***

Below, we show histograms generated for each decoding problem for the $[[5,1,5]]$ code. Each syndrome generates a unique decoding problem to be solved by the quantum annealer, and the histogram depicts how many of the total trials concluded in the respective energy levels of the decoder Hamiltonian. As we can see, not every trial ended in the ground state. This is precisely why we have to run many repeated experiments. Each experiment depicted below involved 5000 shots. The beauty of this approach is that we only need one of the many experiments to actually conclude in the ground state, since the ground state always corresponds to the solution of the decoding problem. The leftmost result, highlighted in light blue, is the lowest energy state of the experiment. In each of these cases, the ground state did indeed correspond to the correct correction, and we did manage to find the ground state. Note that as a result of converting the problem to a QUBO problem, we will get result vectors of much higher dimension than $2n$, which are the auxiliary variables. The auxiliary variables are irrelevant to interpreting the results with respect to the decoding problem since their values are dependent on the values of the first $2n$ terms. For this reason, we truncate the QUBO result vector to only include the first $2n$ terms. We then use Eqs.(18)-(20) to determine the necessary correction.

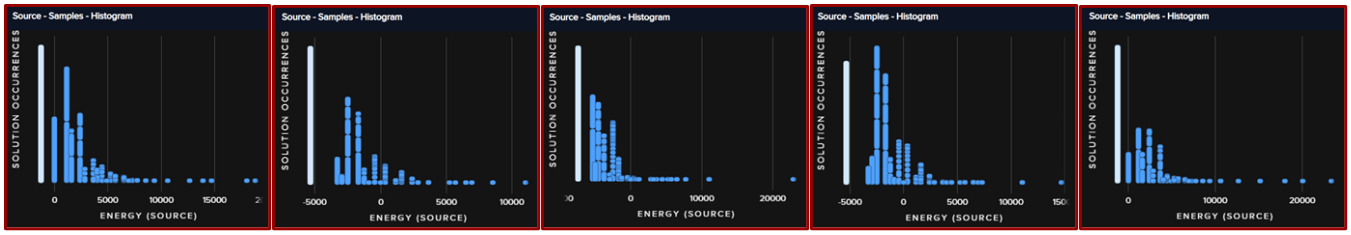

Fig. 2: Histograms from the DWave runs for decoding the syndrome corresponding to all X_i type errors

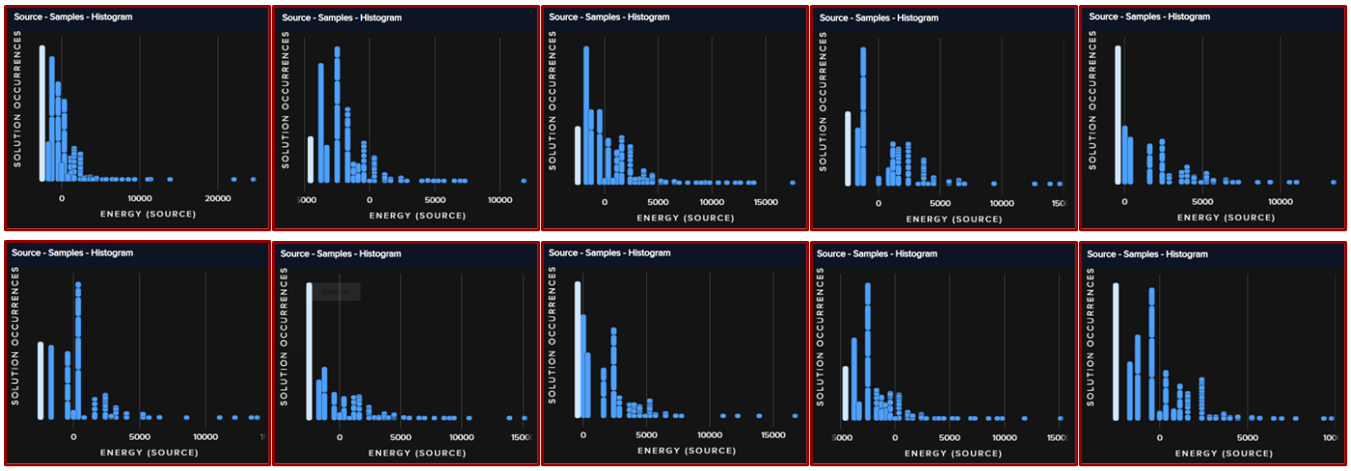

Fig. 3: Histograms from the DWave runs for decoding the syndrome corresponding to all X_iX_j type errors

APPENDIX-D: CODE

In this section, we give the code that we used to get all the results, including the classical testing for the Hamiltonian expression that we described in Sec. III-B1 and the actual annealing runs, which we talked about in III-B2 and showed all the results in Appendix C.

Classical Testing

This Appendix contains all the code for classically testing the correctness of the H_{decoder} given in Eq.(17). The following is a common piece of code, which should be included at the beginning before a piece of code for a specific QECC.

```
#Import required libraries
import matplotlib.pyplot as plt
import qutip as qt
from qutip.qip.operations import hadamard_transform, phasegate
import numpy as np
import scipy as sp

#Define basic operators
I,X,Y,Z = qt.qeye(2),qt.sigmax(),qt.sigmay(),qt.sigmaz()

#Defining a function, which given an input Pauli String - e.g. = XXI
#- returns the operator corresponding to that string - i.e., returns operator X  X  I
def str2op(string):
    op_dict = {'I':I,'X':X,'Y':Y,'Z':Z}
    assert type(string) == str or type(string) == list
    return qt.tensor([op_dict[p] for p in string])

#Setting hyperparameters
j = 1024
h = 400
```

[[3, 1, 3]] ***Bitflip code:***

```
#Set Syndrome bits +1 or -1
s1, s2 =

#Constucting Hamiltonian
Term1 = -j*(s2*str2op('IZZ')+ s1*str2op('ZZI'))
Term2 = -h*(str2op('ZII') + str2op('IZI') + str2op('IIZ'))
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(3))
        print(i)
```

[[5, 1, 5]] ***Bitflip code:***

```
#Set Syndrome bits +1 or -1
s1, s2, s3, s4 =

#Constucting Hamiltonian
Term1 = -j*(s1*str2op('ZZZZI')+
            s2*str2op('IZZII')+
            s3*str2op('IIZZI')+
            s4*str2op('IZZZZ')
            )
Term2 = -h*(str2op('ZIIII')
            +str2op('IZIII')
            +str2op('IIZII')
            +str2op('IIIZI')
            +str2op('IIIIZ')
            )
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(5))
        print(i)
```

[[5, 1, 3]] ***Perfect code***:

```
#Set Syndrome bits +1 or -1
s1, s2, s3, s4 =

#Constucting Hamiltonian
Term1 = -j*(s1*str2op('IZZIIZIIZI')+
            s2*str2op('IIZZIIZIIZ')+
            s3*str2op('IIIZZZIZII')+
            s4*str2op('ZIIIZIZIZI')
            )
Term2 = -h*(str2op('ZIIIIIIIII')
            +str2op('IZIIIIIIII')
            +str2op('IIZIIIIIII')
            +str2op('IIIZIIIIII')
            +str2op('IIIIZIIIII')
            +str2op('IIIIIZIIII')
            +str2op('IIIIIIZIII')
            +str2op('IIIIIIIZII')
            +str2op('IIIIIIIIZI')
             +str2op('IIIIIIIIIZ')
             )
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(10))
        print(i)
```

[[7, 1, 3]] ***Color code***:

```
#Set Syndrome bits +1 or -1
s1, s2, s3 =

#Constucting Hamiltonian
Term1 = -j*(s1*str2op('IIIZZZZ')+
            s2*str2op('IZZIIZZ')+
            s3*str2op('ZIZIZIZ')
            )
Term2 = -h*(str2op('ZIIIIII')
            +str2op('IZIIIII')
            +str2op('IIZIIII')
            +str2op('IIIZIII')
            +str2op('IIIIZII')
            +str2op('IIIIIZI')
            +str2op('IIIIIIZ')
            )
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(7))
        print(i)
```

[[9, 1, 3]] ***Shor code:***

```
#Set Syndrome bits +1 or -1
s1, s2, s3, s4, s5, s6 =

#Constucting Hamiltonian
Term1 = -j*(s1*str2op('ZZIIIIIII')+
            s2*str2op('IIIZZIIII')+
            s3*str2op('IIIIIIZZI')+
            s4*str2op('ZIZIIIIII')+
            s5*str2op('IIIZIZIII')+
            s6*str2op('IIIIIIZIZ')
            )
Term2 = -h*(str2op('ZIIIIIIII')
            +str2op('IZIIIIIII')
            +str2op('IIZIIIIII')
            +str2op('IIIZIIIII')
            +str2op('IIIIZIIII')
            +str2op('IIIIIZIII')
            +str2op('IIIIIIZII')
            +str2op('IIIIIIIZI')
            +str2op('IIIIIIIIZ')
            )
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(9))
        print(i)
```

[[9, 1, 3]] ***Surface code:***

```
#Set Syndrome bits +1 or -1
s1, s2, s3, s4 =

#Constucting Hamiltonian
Term1 = -j*(s1*str2op('ZZIIIIIII')+
            s2*str2op('IZZIZZIII')+
            s3*str2op('IIIZZIZZI')+
            s4*str2op('IIIIIIIZZ')
            )
Term2 = -h*(str2op('ZIIIIIIII')
            +str2op('IZIIIIIII')
            +str2op('IIZIIIIII')
            +str2op('IIIZIIIII')
            +str2op('IIIIZIIII')
            +str2op('IIIIIZIII')
            +str2op('IIIIIIZII')
            +str2op('IIIIIIIZI')
            +str2op('IIIIIIIIZ')
            )
H =  Term1 + Term2

#Computing Eigenvalues and Eigenvectors
eigenvalues, eigenstates = H.eigenstates()

lowest_eigenvalue = eigenvalues[0]
lowest_eigenstate = eigenstates[0]

lowest_eigenstate = np.array(lowest_eigenstate)

#Priting out which qubit has the error for the given syndrome
for i in range(len(lowest_eigenstate)):
    if lowest_eigenstate[i] != 0+0j:
        print(bin(i)[2:].zfill(9))
        print(i)
```

Running QUBO on D-Wave

This section of the Appendix contains the code used to run the decoding problem on D-Wave, as described in section III-B2. This code can be easily used for any decoding problem after the QUBO matrix is defined in a dictionary object. Simply fill in the definition for the necessary QUBO matrix, and D-Wave will provide you with the correct answer to the decoding problem.

```
import dwave.inspector
from dwave.system import DWaveSampler, EmbeddingComposite
import dimod

sampler = EmbeddingComposite(DWaveSampler())

# Your QUBO dictionary is defined here
# each non-zero entry is defined in the dictionary
Q = {}

sampleset = sampler.sample_qubo(Q, num_reads=5000)

# Look at the experimental results
print(sampleset)

# Create Histogram of results
dwave.inspector.show(sampleset)
```

Automated QUBO Generation and DWave Execution [7]

Tool built using `Python` and `SymPy` (a Python package for doing symbolic mathematics). We create a wrapper class for managing `SymPy` expressions that are used to represent stabilizers. These stabilizers are passed to a QUBO Hamiltonian generation class, which creates an Ising Hamiltonian, performs the necessary substitutions and returns a QUBO Hamiltonian equation. This equation, then, is fed to a method that substitutes for J, h and α and returns a dictionary of parameters for performing a DWave run. The attached code also contains methods for testing the $[[5, 1, 5]]$ bit flip code and $[[5, 1, 3]]$ perfect code on the DWave annealer.

```
from sympy import *
from functools import reduce
import re

# TODO: import multithreading/processing (spread out the work for subs/parsing)

# TODO: should this even be a class or should we absorb it into QUBO??
# TODO: write a descriptive comment on what the ising hamiltonian would look like
# H = - J ( syndrome * stab) - h ( sigma_i)
class IsingHamiltonian:
    # NB: tracking J, h symbols rn, abstract them away to keep things simple??
    def __init__(self, gen_stabs, sigmas):
        self.gen_stabs = gen_stabs
        self.syndromes = symbols(' '.join(
            [f's{i}' for i in range(1, len(gen_stabs) + 1)]), commutative=False)
        self.sigmas = sigmas
        self.J = symbols('J')
        self.h = symbols('h')
        self.J_terms = []
        self.h_terms = []

    # TODO: generalize this to n-body terms and rename appropriately
    def gen_four_body_ham_expr(self):
        H1 = -self.J * reduce(lambda x, y: x + y,
            [syn * stab for syn, stab in zip(self.syndromes, self.gen_stabs)])
        H2 = -self.h * reduce(lambda x, y: x + y, self.sigmas)
        return H1 + H2

# TODO: maybe find a better name??
class SympyExpression:
    # static methods
    def sort_expr_by_weight(expr):
        if (len(expr.as_ordered_terms()) == 1):
            return expr
        return sorted(expr.args, key=lambda term: len(term.args))

    def extract_vars_from_term_ordered(term):
        # TODO: sanity check that no summands
        # custom sort boilder-plate in case we have specific sorting requirements
        # ordered_symbols_str = str(term).split('*')
        # return sorted(
        #     term.free_symbols, key=lambda x: ordered_symbols_str.index(str(x)))

        ordered_terms = term.as_ordered_factors()
        # strip integer coefficients
        return list(filter(lambda x: not isinstance(x, Number), ordered_terms))

    # class methods
    def __init__(self, term):
```

```
        self.original_term = term
        self.current_term = term
        self.sorted_summands = SympyExpression.sort_expr_by_weight(term)
        # TODO: store them in the order they appear in the expression
        self.symbols = term.free_symbols
        self.symbols_str = [str(symbol) for symbol in self.symbols]
        # keep history of substitutions made
        # (useful for debugging nested subs)
        self.term_history = [term]
        self.substitution_history = []
        # list of 2-body terms that need to be substituted
        self.term_needing_substitution = []
        self.terms_without_substitution = []
        # will be passed from the parent (QUBOHamiltonian)
        self.substitution_symbols = []

    def find_necessary_substitutions(self, terms_needing_substitution):
        subs = set()
        for term in terms_needing_substitution:
            sub1 = (term[0], term[1])
            # sub for first two terms if none of them are in subs
            if len(term) == 3:
                sub2 = (term[1], term[2])
                if not any(sub in subs for sub in [sub1, sub2]):
                    subs.add(sub1)
            # add both sub1 and sub2 for 4 terms
            if len(term) == 4:
                sub2 = (term[2], term[3])
                subs.update([sub1, sub2])
        return subs

    # TODO: filter subs that are not necessary for robustness
    # subs: {symbol_expr: substitution_symbol_expr, ...}
    def perform_substitution(self, subs):
        # TODO: all tracking stuff
        self.current_term = expand(self.current_term.subs(subs))
        self.sorted_summands = SympyExpression.sort_expr_by_weight(self.current_term)
        return self.current_term

    def perform_subs_on_w_3_or_more_summands(self, subs):
        curr_term = 0
        for term in self.sorted_summands:
            vars_in_term = SympyExpression.extract_vars_from_term_ordered(term)
            if len(vars_in_term) > 2:
                curr_term += term.subs(subs)
            else:
                curr_term += term
        self.current_term = curr_term
        self.sorted_summands = SympyExpression.sort_expr_by_weight(self.current_term)
        return self.current_term

    def extract_substitutions(self):
        # TODO: make sure this assumption holds i.e. all symbols of same type
        # else we'll need to extract substitution candidates more carefully
        sub_candidates = []
        for term in reversed(self.sorted_summands):
            vars_in_term = SympyExpression.extract_vars_from_term_ordered(term)
```

```
            if len(vars_in_term) > 2:
                sub_candidates.append(vars_in_term)

        return self.find_necessary_substitutions(sub_candidates)

    # TODO: decide if we should keep
    def find_degree_of_expr(expr):
        # Fully expand to get the terms in the expression
        terms = expand(expr).args

        max_terms_multiplied = 0
        max_term_example = None

        for term in terms:
            # check if the term is a multiplication
            if isinstance(term, Mul):
                # get the number of factors in the multiplication
                num_factors = len(term.args)
                # update the maximum number of terms multiplied
                if num_factors > max_terms_multiplied:
                    max_terms_multiplied = num_factors
                    max_term_example = term

        print(max_terms_multiplied, max_term_example)

class QUBOHamiltonianBuilder:
    BINARY_VAR_SYMBOL = 'x'
    Z_VAR_SYMBOL = 'z'
    Y_VAR_SYMBOL = 'y'
    # TODO: have another init that takes in an ising ham
    # TODO: maybe params should be a dict??
    def __init__(self, gen_stabs, sigmas, debug=False):
        self.sigmas = sigmas
        self.stabs = gen_stabs
        self.Ising_H = Ising_H = IsingHamiltonian(gen_stabs, sigmas)
        self.h_terms = sigmas
        self.J_terms = gen_stabs
        self.syndromes = Ising_H.syndromes
        # substitution stuff
        self.binary_vars = []
        self.binary_var_subs = {}
        self.z_vars = [] # no way of knowing num z vars until we parse the expr
        self.z_subs_for_each_stab = []
        self.y_vars = [] # #(y_vars) = #(binary_vars) + #(z_vars)
        self.y_subs = {}
        # penalty ham stuff
        self.penalty_terms = [0]*len(gen_stabs)
        self.QUBO_H = None
        # debug stuff
        self.debug = False

    def perform_binary_var_subs(self):
        # create binary variables for each ising spin
        # TODO: wrap them as SympyExpression objects??
        # NB: current pattern: make symbols in parent => distribute to children
        # to perform subs
        self.binary_vars = list(symbols(' '.join(
```

```
            [f'{self.BINARY_VAR_SYMBOL + str(i)}'
             for i in range(1, len(self.sigmas)+1)]),
             commutative=False))

        # replace each ising spin with binary variable
        binary_vars_subs = [1 - 2 * bin_var for bin_var in self.binary_vars]
        self.h_terms = [SympyExpression(bin_var_sub)
            for bin_var_sub in binary_vars_subs]
        self.binary_var_subs = subs = dict(zip(self.sigmas, binary_vars_subs))
        # TODO: spreadout the work onto different cores
        self.J_terms = [stab.perform_substitution(subs) for stab in self.stabs]

    # we only substitute for terms with 3 or 4 x symbols
    # TODO: handle overlapping subs e.g. x1x2 in S1 = x1x2x3x4, S2 = x1x2x3x5 ??
    def perform_z_subs(self):
        subs_to_perform = [stab.extract_substitutions() for stab in self.stabs]
        # count number of subs to perform
        num_z_subs = sum([len(subs) for subs in subs_to_perform])
        self.z_vars = list(symbols(' '.join(
            [f'{self.Z_VAR_SYMBOL + str(i)}'
             for i in range(1, num_z_subs + 1)]),
             commutative=False))
        # pair z_vars with subs
        z_subs_all_stabs = []
        curr_z_term_idx = 0
        for (idx,subs) in enumerate(subs_to_perform):
            z_subs_all_stabs.append({})
            for sub in subs:
                to_substitute = sub[0] * sub[1]
                z_subs_all_stabs[idx][to_substitute] = self.z_vars[curr_z_term_idx]
                curr_z_term_idx += 1
        self.z_subs_for_each_stab = z_subs_all_stabs
        self.J_terms = [self.stabs[idx].perform_subs_on_w_3_or_more_summands(z_subs)
                        for (idx, z_subs) in enumerate(z_subs_all_stabs)]

    def generate_penalty_hamiltonian(self):
        # each stabilizer is going to have a penalty term associated with it
        for (idx, subs) in enumerate(self.z_subs_for_each_stab):
            penalty_term = 0
            for (sub, z_term) in subs.items():
                [sub1, sub2] = sub.as_ordered_factors()
                penalty_term += sub - (2 * z_term * (sub1 + sub2)) + (3 * z_term)

            self.penalty_terms[idx] = SympyExpression(expand(penalty_term))

    def perform_y_subs(self):
        self.y_vars = symbols(' '.join([
            f'{self.Y_VAR_SYMBOL + str(i)}'
            for i in range(1, len(self.binary_vars) + len(self.z_vars) + 1)]),
                        commutative=False)
        self.y_subs = dict(zip(self.binary_vars + self.z_vars, self.y_vars))
        # go into each h, J and penalty term and replace with y_i
        # h terms
        y_subs_for_h_terms = dict(
        zip(self.binary_vars, self.y_vars[:len(self.binary_vars)]))
        self.h_terms = [term.perform_substitution(y_subs_for_h_terms)
                        for term in self.h_terms]
```

```
        # J terms and penalty terms
        J_int = []
        P_int = []
        for (idx, stab) in enumerate(self.stabs):
            # TODO: update stab and access property instead of accessing via sympy
            symbols_in_stab = stab.current_term.free_symbols

            y_subs_for_stab = {k: v for k, v in self.y_subs.items()
                if k in symbols_in_stab}

            J_int.append(stab.perform_substitution(y_subs_for_stab))
            P_int.append(
            self.penalty_terms[idx].perform_substitution(y_subs_for_stab))

        self.J_terms = J_int
        self.penalty_terms = P_int

    # TODO: flesh out with debug stuff
    def gen_qubo_ham(self):
        self.perform_binary_var_subs()
        # debug statement: print H after binary var subs
        self.perform_z_subs()
        for term in self.J_terms:
            print(term)
        # debug statement: print H after z subs
        # print_only_2_body_terms(H_after_z_subs)
        # print(check_if_hamiltonian_2_body(H_after_z_subs))

        # # create penalty hamiltonian
        self.generate_penalty_hamiltonian()

        self.perform_y_subs()
        J_terms_summed = 0
        for (idx, term) in enumerate(self.J_terms):
            J_terms_summed += -1 * term * self.syndromes[idx]
        QUBO_H = (
        symbols('h') * -1 * sum(self.h_terms) +
        symbols('J') * J_terms_summed +
        symbols('a') * sum(self.penalty_terms))
        return expand(QUBO_H)
        # print("**********************")
        # qubo_H = qubo_H.subs(dict(zip(binary_vars + z_terms, y_vars)))
        # return qubo_H

    def check_if_hamiltonian_2_body(H):
        for term in expand(H).args:
            relevant_symbols = [symbol for symbol in term.free_symbols
            if re.match(r'z|x', str(symbol))]
            if len(relevant_symbols) > 2:
                print(relevant_symbols, term)
                return False
        return True

def get_ordered_y_symbols_in_term(term):
    ordered_y_str = [symbol for symbol in str(term).split('*')
        if 'y' in symbol]
```

```
    y_symbols_in_term = [symbol for symbol in term.free_symbols if 'y' in str(symbol)]
    y_symbols = sorted(y_symbols_in_term, key=lambda x: ordered_y_str.index(str(x)))
    return y_symbols

def gen_dwave_param_key(y_symbols):
    if len(y_symbols) == 0:
        return None
    if len(y_symbols) == 1:
        return tuple([str(y_symbols[0]), str(y_symbols[0])])
    if len(y_symbols) == 2:
        return tuple([str(y_symbols[0]), str(y_symbols[1])])
    if len(y_symbols) > 2:
        return ValueError("Qubo ham with more than 2 y terms")

def convert_to_dwave_params(qubo_H, J, h, alpha, syndromes):
    dwave_params = {}
    symbolic_params = {}
    for term in expand(qubo_H).args:
        ordered_y_symbols = get_ordered_y_symbols_in_term(term)
        key = gen_dwave_param_key(ordered_y_symbols)

        # NB: we don't care about terms with no y symbols
        if key is None:
            continue

        # make substitutions
        value_int = term.subs(dict(zip(ordered_y_symbols,
                                       [1]*len(ordered_y_symbols))))
        # keep symbolic representation for debugging purposes
        if key in symbolic_params:
            symbolic_params[key] += value_int
        else:
            symbolic_params[key] = value_int

        # substitute for J, h, alpha
        value = value_int.subs({symbols('J'): J,
                                symbols('h'): h,
                                symbols('a'): alpha})
        # substitute for syndromes
        syndrome_symbols_in_term = [symbol for symbol in value.free_symbols
            if 's' in str(symbol)]
        syndromes_indexes = [int(re.search(r'\d+', str(s)).group())
            for s in syndrome_symbols_in_term]
        syndromes_in_term = [syndromes[i-1] for i in syndromes_indexes]
        value = value.subs(dict(zip(syndrome_symbols_in_term, syndromes_in_term)))
        if key in dwave_params:
            dwave_params[key] += value
        else:
            dwave_params[key] = value
    return dwave_params

def bit_flip_test():
    Z_ops = [Z1, Z2, Z3, Z4, Z5] = (
        symbols('Z1 Z2 Z3 Z4 Z5', commutative=False)
    )
    stabs = [
        SympyExpression(Z1*Z2*Z3*Z4),
```

```
        SympyExpression(Z2*Z3),
        SympyExpression(Z3*Z4),
        SympyExpression(Z2*Z3*Z4*Z5)
    ]
    qubo = QUBOHamiltonianBuilder(stabs, Z_ops)
    qubo_ham = qubo.gen_qubo_ham()
    J=1024
    a = 8*J
    h = 400
    syndromes = [-1,1,1,1]
    d_wave_params_auto = convert_to_dwave_params(qubo_ham, J, h, a, syndromes)


def perfect_code_test():
    Z_ops = [Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, Z9, Z10] = (
        symbols('Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10', commutative=False)
    )

    stabs = [
        SympyExpression(Z2*Z3*Z6*Z9),
        SympyExpression(Z3*Z4*Z7*Z10),
        SympyExpression(Z4*Z5*Z6*Z8),
        SympyExpression(Z1*Z5*Z7*Z9)
    ]
    qubo = QUBOHamiltonianBuilder(stabs, Z_ops)
    qubo_ham = qubo.gen_qubo_ham()
    J=1024
    a = 8*J
    h = 400
    syndromes = [1,1,1,1]
    d_wave_params_auto = convert_to_dwave_params(qubo_ham, J, h, a, syndromes)


perfect_code_test()
```