UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Uku Kangur

Computing logical X and Z gates for stabilizer codes

Master's Thesis (30 ECTS)

Supervisor(s): Francisco Javier Gil Vidal, MSc
Dirk Oliver Theis, PhD

Tartu 2022

Computing logical X and Z gates for stabilizer codes

Abstract:
Quantum error correction is an inseparable part of making quantum computing viable for real-life use cases. The field heavily relies on experimentation, thus it is important that useful research software in the area exists and is openly available. This thesis introduces an algorithmic method for calculating logical X and Z operators given a set of Pauli group elements. The aim of the thesis is to analyse and implement this method in the Python programming language. Results and performance of the implemented program are given through experiments.

Keywords:
Quantum Computing, Quantum Error Correction, Stabilizer Theory, Pauli Groups

CERCS:
P170 Computer science, numerical analysis, systems, control

Stabilisaatorkoodide loogiliste X ja Z operatsioonide arvutamine

Lühikokkuvõte:

Kvantvigadeparandus on lahutamatu osa kvantarvutite kasutuskõlblikuks muutmisel. Kvantarvutuse valdkond tugineb suuresti füüsikalistele katsetele, mistõttu on oluline, et teadlastel olemas vajalik avalikult kättesaadav teadustarkvara nende läbiviimiseks. Antud lõputöö tutvustab algoritmilist meetodit loogiliste X- ja Z-operaatorite leidmiseks kasutades sisendina Pauli rühma elemente. Lõputöö eesmärk on analüüsida ja rakendada seda meetodit Pythoni programmeerimiskeeles. Rakendatud programmi tulemused ja efektiivsus antakse edasi läbi eksperimentaalsete katsete.

Võtmesõnad:

Kvantarvutus, Kvantvigadeparandus, Stabilisaatori Teooria, Pauli Rühmad

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

// Acknowledgments

I take this section to thank everyone who supported me throughout my master's journey.

First I would like to thank my supervisors Francisco Javier Gil Vidal and prof. Dirk Oliver Theis. Their relentless support and encouragement was upmost important while writing this thesis. Dirk's lectures on linear algebra and Javier's lectures on quantum error correction helped tremendously to better understand quantum informational concepts and proofs.

I would like to thank Evgenii Dolzhkov, and Galina Pass, who first as course mates and later as teaching assistants helped me better understand quantum algorithms and quantum algorithm design. I would like to also thank dr. Veiko Palge for giving me the opportunity to teach to others through quantum computing seminars. Their feedback was significant throughout my master's.

I would like to thank my friends and collegues Henry Narits, Tejas Anil Shah, Alejandro Villoria, Anabel Ovide Gonzalez, Handy Kurniawan, Markus Punnar and Oliver Vainumäe for lenghty discussions and study groups on quantum computing. Their support also helped me develop on a personal level.

Lastly, I would like to thank my BSc thesis supervisor Vitaly Skachek, who gave me insights into academical work and introduced me to the field of error-correction, which I now hold dear to my heart.

Contents

Notations **6**

1 Introduction **7**

2 Background **8**

2.1 Group Theory 8

2.2 Pauli Groups 10

2.3 Stabilizer theory 13

2.4 Pauli group homomorphism and bilinear form 17

2.5 Calculating logical X and Z 24

3 Implementation **30**

3.1 Input parsing 30

3.2 Finding maximum commuting list of operators 32

3.3 Solving of systems of linear equations over $\mathbb{F}_2$ 33

3.4 Calculating logical Z and X operators 36

3.5 Complexity and performance 37

4 Conclusion **41**

References **43**

Appendix **44**

I. Source code 44

II. Licence 45

Notations

$\mathcal{H}$	Hilbert space
$(G, \cdot)$	Group G with operation $\cdot$
e	Identity element
a^{-1}	Inverse of a
$\cong$	Group isomorphism
$\lvert G \rvert$	Order of G
$[G : H]$	Index of H in G
$H \triangleleft G$	H is normal subgroup of G
$\mathbb{F}_2$	Galois field of size 2
$\lvert\psi\rangle$	Ket representation of qubit (column vector)
$\langle\psi\rvert$	Bra representation of qubit (row vector)
$\langle\psi\vert\phi\rangle$	Inner product of ψ and ϕ
$\lvert 010\rangle$	Tensor product $\lvert 0\rangle \otimes \lvert 1\rangle \otimes \lvert 0\rangle$
$\mathcal{P}_n$	Pauli group acting on n qubits
XYZ	Tensor product $X \otimes Y \otimes Z$
X_1	X operation acting on the first qubit
I	Identity matrix
$I^{\otimes n}$	N-fold tensor product of I
P	Projector
$\lvert\overline{0}\rangle$	Ket representation of logical qubit
$\overline{X}$	Logical operation X
$\vec{v}$	Vector v
H	Check matrix
Λ	Simplectic inner product matrix

1 Introduction

Quantum computation has proven to give an exponential speed-up in-regards to many classical computational problems. This has given rise to many new fields such as quantum algorithmics and quantum cryptography (i.e. Shor [Sho94] and Grover [Gro96]). Even though quantum algorithms perform well in theory, practical applications are strongly susceptible to computational errors due to the environment (temperature, radiation, light, etc.). The critics of quantum computing also see this as the main downside compared to their classical counterparts [Aar13]. It was only in 1995, that Peter Shor [Sho95] first showed that quantum errors can be corrected by constructing the first quantum error correcting code. This discovery proved, that by using quantum error correcting code, we can make quantum computation scaleable enough to run the beforementioned algorithms.

Nikolas P. Breuckmann has stated in his 2019 talk at UCL [Bre19] that bridging the gap between theory and experimental work has been the main research problem in quantum error correction during the last 20 years. We are only now starting to see some overlap between the two. Therefore it is important that researchers in the field have openly-available software to aid their research in finding good quantum error correction codes. While there exists (Python based) software for testing and experimenting with quantum codes such as QuaEC [Gra12] and qecsim [Tuc20], these often are specialized tools for certain use-cases and don't cover a lot of demand. The QuaEC package lets its users manipulate Pauli and Clifford operators (and their symplectic representations), but does not support finding good logical operators given a starting set of Pauli group elements. In the package qecsim logical operators are expected as a human-given input of the quantum error code you are using, thus it also does not support finding them. Therefore there is still a gap in quantum error correction regarding open-source research software.

The aim of this thesis was to write computer software (in Python) with the following functionality: An input list of multi-Pauli generators is processed to return nice choices for the logical Z and X operators of the corresponding logical qubit. The thesis provides the mathematical background, complexity and implementation of the algorithm. This software could be used for daily research in the area of quantum error correction. All of the code created is available in the Appendix I. Source Code. The thesis is composed by the following two chapters:

- Chapter 2 gives an overview of the required mathematical theory needed to understand both the problem and also the algorithm itself. It introduces the primary notions used from group theory and stabilizer theory.
- Chapter 3 provides the analysis of implementation and complexity of the algorithm. In addition it also shows benchmarking results (given a growing number of qubits).

2 Background

In this section we cover the mathematical basics and concepts needed to understand how we can utilize Pauli group homomorphisms to find commuting Pauli group elements. The aim of the chapter is to familirize the reader with the mathematical background of the algorithm implemented. The group theory relies on the algebra book by Lang [Lan05]. The stabilizer theory introduced is based on papers by Gottesmann [Got96], [Got97]. The quantum error correction theory introduced here is mainly based on books by Mermin [Mer07] and Nielsen & Chuang [NC10].

2.1 Group Theory

Definition 2.1.1. *A group is a set G with a binary operation $\cdot$ with the following requirements:*

- *Closure:* $\forall a, b \in G; \quad a \cdot b \in G$
- *Associativity:* $\forall a, b \in G; \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- *Identity:* $\exists e \in G, \forall a \in G; \quad ea = a$
- *Inverse:* $\forall a \in G, \exists b \in G; \quad ba = e$

Definition 2.1.2. *An abelian group $(G, \cdot)$ is a group where the operation is commutative:* $\forall a, b \in G; \quad a \cdot b = b \cdot a$.

In what follows G will always denote a finite group. The number of elements of G is called its order, denoted by $|G|$.

Definition 2.1.3. *We say that S is a subgroup of G if $(S, \cdot)$ is a group, where $S \subset G$.*

Definition 2.1.4. *Given $g \in G$, the left coset of H by g is the set*

$$gH = \{gh : h \in H\} \subset G$$

The right coset of H by $g \in G$ is the set

$$Hg = \{hg : h \in H\} \subset G$$

For two different elements $g, g' \in G$, we have either $gH = g'H$, or $gH \cap g'H = \emptyset$. The same for right co-sets.

The number of left cosets of H in G is equal to the number of right cosets of H in G and is called the index of H in G, noted as $[G : H]$. Any two left (right) cosets have the same number of elements, so $|H||_{|G|}$.

Lemma 2.1.5. *If $|G| = n \in \mathbb{N}$, $H \subset G$, and $|H| = m$, then $m|n$ (m divides n).*

Definition 2.1.6. *An equivalence relation on a set G is a binary relation $\sim$ with the following properties:*

- *Reflexivity:* $\forall a \in G; \quad a \sim a$
- *Symmetry:* $\forall a, b \in G; \quad a \sim b \implies b \sim a$
- *Transitivity:* $\forall$*a,b,c* $\in G$; *if $a \sim b$ and $b \sim c$, then $a \sim c$*

Definition 2.1.7. *The equivalence class of $a \in G$ under $\sim$ is $[a] = \{b \in G : b \sim a\}$.*

Definition 2.1.8. *The quotient set G/H is the set of all equivalence classes in G with respect to an equivalence relation H.*

Definition 2.1.9. *We say that $H \subseteq G$ is a normal subgroup of G of $\forall x \in G, \forall h \in H, xhx^{-1} \in H$. If $H \subseteq G$ is normal, we denote it as $H \triangleleft G$.*

Lemma 2.1.10. *If G is Abelian, then every subgroup $H \subseteq G$ is normal.*

Theorem 2.1.11. *If $H \triangleleft G$, then the quotient set of G by H is a group, also called the quotient group $(G/H, \cdot)$. The operation in G/H is defined in a natural way from the operation "$\cdot$" in G, as follows:*

$$(g_1H) \cdot (g_2H) = (g_1g_2)H$$

We have $(gH)^{-1} = g^{-1}H$.

Proof Let $g_1 \sim k_1$, which means that $\exists h_1 \in H$ such that $k_1 = g_1 \cdot h_1$. Similarly, let $g_2 \sim k_2$, with $k_2 = g_2 \cdot h_2$. We want to prove that $(g_1 \cdot g_2) \sim (k_1 \cdot k_2)$, i. e. $\exists h \in H$ such that $k_1 \cdot k_2 = (g_1 \cdot g_2) \cdot h$. Now

$k_1 \cdot k_2 = (g_1 \cdot h_1) \cdot (g_2 \cdot h_2) = [g_1 \cdot (g_2 \cdot g_2^{-1}) \cdot h_1] \cdot (g_2 \cdot h_2) = (g_1 \cdot g_2) \cdot [(g_2^{-1} \cdot h_1 \cdot g_2) \cdot h_2]$.

Because $H \triangleleft G$,

$$g_2^{-1} \cdot h_1 \cdot g_2 \in H \Rightarrow h = (g_2^{-1} \cdot h_1 \cdot g_2) \cdot h_2 \in H$$

so

$$k_1 \cdot k_2 = (g_1 \cdot g_2) \cdot h.$$

■

Definition 2.1.12. *A commutative ring is a set R with two operations $+$ and $\cdot$ (addition and multiplication respectively) with the following properties:*

- *Associativity:* $a + (b + c) = (a + b) + c$ *and* $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- *Commutativity:* $a + b = b + a$ *and* $a \cdot b = b \cdot a$
- *Additive identity:* $\exists 0 \in R$ *so that* $a + 0 = a$
- *Multiplicative identity:* $\exists 1 \neq 0 \in R$ *so that* $1 \cdot a = a$
- *Additive inverses:* $\forall a \in R, \exists - a \in R$ *satisfying* $a + (-a) = 0$
- *Distributivity of multiplication over addition:* $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Definition 2.1.13. *A* ***field*** *is a ring K such that every nonzero element has a multiplicative inverse. That is, $\forall a \in K$ with $a \neq 0$, $\exists a^{-1} \in K$ so that $a \cdot a^{-1} = 1$*

2.2 Pauli Groups

Definition 2.2.1. *A linear (vector) space over a field $\mathbb{F}$ is a set V with a binary vector addition operation "+" and a multiplication by scalars "$\cdot$" in the field, that satisfy the following conditions:*

- *Associativity of vector addition:* $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$
- *Commutativity of vector addition:* $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
- *identity of vector addition:* $\forall \vec{v} \in V, \exists \vec{0} \in V$ *such that* $\vec{v} + \vec{0} = \vec{v}$
- *inverse of vector addition:* $\forall \vec{v} \in V, \exists - \vec{v} \in V$ *such that* $\vec{v} + (-\vec{v}) = e$
- *Compatibility of scalar multiplication with field multiplication:* $a \cdot (b \cdot \vec{v}) = (a \cdot b) \cdot \vec{v}$
- *identity of scalar multiplication:* $\forall \vec{v} \in V, \exists e \in V$ *such that* $\vec{v} \cdot e = \vec{v}$
- *Distributivity of scalar multiplication with respect to vector addition:* $a(\vec{u} + \vec{v}) = a\vec{u} + a\vec{v}$
- *Distributivity of scalar multiplication with respect to field addition:* $(a + b)\vec{v} = a\vec{v} + b\vec{v}$

Later in the thesis, we will be doing operations over the two element field $\mathbb{F}_2$.

Definition 2.2.2. *A Hilbert space $\mathcal{H}$ is a real or complex vector space with the inner product operation. Let there be two n-length real vectors $\vec{a}, \vec{b}$. The inner product in a real Hilbert space is defined as the dot product:*

$$\langle \vec{a} | \vec{b} \rangle = a_1 b_1 + \ldots a_n b_n$$

Now let there be two n-length complex vectors $\vec{a}, b$. The inner product in a complex Hilbert space is defined as:

$$\langle\vec{a}|\vec{b}\rangle = a_1^* b_1 + \dots a_n^* b_n$$

where a_i^* represents the complex conjugate of a_i.

The inner product of a Hilbert space also satisfies the following properties:

- *Inner product is conjugate symmetric:*

$$\langle\vec{a}|\vec{b}\rangle = (\langle\vec{b}|\vec{a}\rangle)^*$$

- *Inner product is anti-linear in its first argument:*

$$\langle\alpha\vec{a}_1 + \beta\vec{a}_2|\vec{b}\rangle = \alpha^*\langle\vec{a}_1|\vec{b}\rangle + \beta^*\langle\vec{a}_2|\vec{b}\rangle$$

- *Inner product of an element with itself is positive definite:*

$$\langle\vec{a}|\vec{a}\rangle > 0 \quad \textit{if}\ \vec{a} \neq \vec{0}$$

$$\langle\vec{a}|\vec{a}\rangle = 0 \quad \textit{if}\ \vec{a} = \vec{0}$$

Definition 2.2.3. *We call a function $f : \mathbb{C}^n \to \mathbb{R}$ a norm if it satisfies the following conditions:*

- *Positivity: $f(\vec{x}) \geq 0, f(\vec{x}) = 0 \iff x = 0$*
- *Homogeneity: $f(\alpha\vec{x}) = |\alpha| f(\vec{x}), \forall\alpha \in \mathbb{C}$*
- *Triangle inequality: $f(\vec{x} + \vec{y}) \leq f(\vec{x}) + f(\vec{y})$*

If the norm is given on a vector space, we often denote it as $f(\vec{x}) = ||\vec{x}||$.

Definition 2.2.4. *We call a non-zero vector $\vec{x} \in \mathbb{C}^n$ a unit vector if its length is 1:*

$$||\vec{x}|| = 1$$

Definition 2.2.5. *Let $\vec{x} \in \mathbb{C}^n$; the norm of $\vec{x}$ is defined as $||\vec{x}| \models \sqrt{\langle\vec{x}|\vec{x}\rangle}$. A vector $\vec{u} \in \mathbb{C}^n$ is a unit vector if $||\vec{u}|| = 1$. Any nonzero vector $\vec{x} \in \mathbb{C}^n$ can be normalized (i.e., replaced by its corresponding unit vector $\vec{u}_{\vec{x}}$) by letting $\vec{u}_{\vec{x}} = \frac{\vec{x}}{||\vec{x}||}$.*

Definition 2.2.6. *We call two vectors $\vec{a}, \vec{b} \in \mathcal{H}$ orthogonal, if their inner product is zero:*

$$\langle\vec{a}|\vec{b}\rangle = 0;$$

orthogonal unit vectors are called orthonormal.

In quantum computing, we use qubits, which are the quantum equivalents to bits. We define

$$0 \text{ as } |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$1 \text{ as } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

We also define the vector space V of all linear combinations of $|0\rangle, |1\rangle$ with complex coefficients. This is a Hilbert space $\mathcal{H}$, with the inner product

$$\langle (a_1, a_2)|(b_1, b_2)\rangle = a_1^* b_1 + a_2^* b_2$$

It is important to note that $|0\rangle$ and $|1\rangle$ form an orthonormal basis, which we also call the computational basis. Qubits are normalised linear combinations of $|0\rangle, |1\rangle$, meaning for some $x, y \in \mathbb{C}$ the qubit is defined as $x|0\rangle + y|1\rangle$, where

$$|x|^2 + |y|^2 = 1$$

Definition 2.2.7. *We call Pauli matrices the following matrices*

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The Pauli matrices all square to the identity matrix:

$$I^2 = X^2 = Y^2 = Z^2 = I$$

Pauli matrices (other than the identity) anti-commute with each other, as is readily verified by direct computation.

These matrices implement the following operations on our qubit vector representations:

- identity: $I|0\rangle = |0\rangle, I|1\rangle = |1\rangle$
- bit-flip: $X|0\rangle = |1\rangle, X|1\rangle = |0\rangle$
- phase-flip: $Z|0\rangle = |0\rangle, Z|1\rangle = -|1\rangle$
- the combination of bit and phase flips: $Y|0\rangle = i|1\rangle, Y|1\rangle = -i|0\rangle$

Let us define a "real" $Y = XZ$ operation. From now on, we will be using this to define concepts.

Definition 2.2.8. *We define the* 1*-qubit Pauli group as*

$$\mathcal{P}_1 = \{\pm I, \pm X, \pm Y, \pm Z, \pm iI, \pm iX, \pm iY, \pm iZ\}$$

The group operation is matrix multiplication. Therefore we have:

$$XY = iZ$$

$$YZ = iX$$

$$ZX = iY$$

But also since we know that Pauli matrices (except for the identity) anti-commute, then:

$$YX = -XY$$

$$XZ = -ZX$$

$$ZY = -YZ$$

Definition 2.2.9. *We define the* n*-qubit Pauli group as*

$$\mathcal{P}_n = \{cG_1 \otimes G_2 \otimes \ldots \otimes G_n\},$$

where each $G_i \in \{I, X, Z, Y\}$ *and* $c \in \{\pm 1, \pm i\}$

2.3 Stabilizer theory

Definition 2.3.1. *A vector* $|\psi\rangle$ *is stabilized by a Pauli Group element* $g \in \mathcal{P}_n$ *if*

$$g|\psi\rangle = |\psi\rangle$$

Definition 2.3.2. *A non-void subset* $S \subset \mathcal{P}_n$ *is called the stabilizer of a subset* V_S *if the elements of* S *stabilize all of the vectors in* V_S *(leaves them invariant).*

Lemma 2.3.3. S *is a subgroup of* $\mathcal{P}_n$*.*

Proof. We know that $I^{\otimes n} \in \mathcal{P}_n$ for every n. We also know that the identity stabilizes every vector (as long as the dimensions match) i.e. $I|\psi\rangle = |\psi\rangle$. Therefore the identity must always belong to the subset $S \subset \mathcal{P}_n$. This also proves that our subset S can not be empty.

Let us define two n-length Pauli group elements $a = a_1 \otimes \ldots \otimes a_n$ and $b = b_1 \otimes \ldots \otimes b_n$ a, where $a, b \in \mathcal{P}_n$ and a_i, b_i are Pauli matrices.

We know that all Pauli matrices square to the identity (meaning they are their own inverses), meaning $a_i a_i^{-1} = a_i^2 = I$. Therefore every Pauli group element (tensor of Pauli matrices) must also be its own inverse as $a \cdot a = a_1 a_1 \otimes \ldots \otimes a_n a_n = I_1 \otimes \ldots \otimes I_n$. Thus if $a \in S$, then also $a^{-1} \in S$.

Let there be $a \in S$ and $b = a^{-1}$. Let the product of them be $ab = c$. We can now see that $a \cdot b = ab = c$. Since S is closed under inversion, then $b \in S$ and thus $a \cdot b = aa^{-1} = a^2 = I \in S$.

Since S is a subset of the group $\mathcal{P}_n$, then S also inherits the associativity of elements.

Since S is a subset $S \subset \mathcal{P}_n$ and S satisfies the properties of a group, therefore S is a subgroup of $\mathcal{P}_n$.

Lemma 2.3.4. *If a subgroup $S \subset \mathcal{P}_n$ stabilizes some non-trivial subspace V_S then $-I \notin S$ and S is an abelian subgroup*

Proof. Lets first show that $-I \notin S$ for non-trivial subspace. We know that a state $|\psi\rangle$ is stabilized by $-I$, when $-I|\psi\rangle = |\psi\rangle$. We can now say that $-I|\psi\rangle = -|\psi\rangle = |\psi\rangle$. This can only be true if $|\psi\rangle$ is the zero vector.

Let us now show that S must be abelian to stabilize a non-trivial subspace. Let us imagine, we have two anti-commuting elements $g_1, g_2 \in S$. Since they belong to the stabilizer, they both stabilize an arbitrary vector $|\psi\rangle \in V_S$, where V_S is the subspace stabilized by S:

$$g_1|\psi\rangle = |\psi\rangle$$

$$g_2|\psi\rangle = |\psi\rangle$$

Since g_1 and g_2 anti-commute, then

$$g_1 g_2 = -g_2 g_1$$

Now we can see that

$$|\psi\rangle = g_1 g_2 |\psi\rangle = -g_2 g_1 |\psi\rangle = -|\psi\rangle$$

As shown in the previous part, $|\psi\rangle = -|\psi\rangle$ only if our vector $|\psi\rangle$ is trivial.

Therefore we have shown that if a subgroup $S \subset \mathcal{P}_n$ stabilizes a non-trivial subspace then $-I \notin S$ and S is an abelian subgroup. ■

Lemma 2.3.5. *Let $V_S \subset V$ be a subset stabilized by some subgroup $S \subset \mathcal{P}_n$, then V_S is a subspace of V.*

Proof. Let us have an element $g \in S$. Let us first show that it is closed under addition. Let us have two vectors $|\psi\rangle, |\phi\rangle \in V_S$. We know that by definition

$$g|\psi\rangle = |\psi\rangle$$

$$g|\phi\rangle = |\phi\rangle$$

Let us define their sum as

$$|x\rangle = |\psi\rangle + |\phi\rangle$$

We can now see that due to linearity

$$g|x\rangle = g(|\psi\rangle + |\phi\rangle) = g|\psi\rangle + g|\phi\rangle = |\psi\rangle + |\phi\rangle = |x\rangle$$

Let us then show that it is closed under multiplication by scalars. Let us have a vector $|\psi\rangle \in V_S$ and a scalar $\alpha \in \mathbb{C}$. We can now see that due to linearity

$$g\lambda|\psi\rangle = \lambda g|\psi\rangle = \lambda|\psi\rangle$$

Let us show that the zero vector exists in the subset. This is trivial since

$$g\vec{0} = \vec{0}$$

Thus we have shown that V_S is a subspace of V. ■

To do computations in our error-correction system, we use the vectors in our stabilized subspace V_S to define a new set of qubits (called logical qubits or information qubits). These logical qubits are a linear combination of the regular physical qubits. Due to the nature of the stabilizer theory, it is easy to detect if any errors occur with the logical qubits, as they would be forced into another subspace and thus would no longer be stabilized by every element of the stabilizer. From here on out, we base our calculations primarily on these logical qubits.

Definition 2.3.6. *We note n as the number of physical qubits and k the number of logical qubits. An $[n, k]$ stabilizer code $C(S)$ is the non-trivial vector space V_S stabilized by $S \subset \mathcal{P}_n$, such that $I \notin S$ and S has $n - k$ independent and commuting generators, $S =< g_1, ..., g_{n-k} >$.*

Definition 2.3.7. *The logical basis states are any 2^k orthonormal vectors in $C(S)$ given a stabilizer with $n-k$ generators. We denote logical basis states with an overline i.e $|\overline{0}\rangle$.*

Let us have two orthonormal basis states $|\overline{0}\rangle$ and $|\overline{1}\rangle$. Operations that are done with logical qubits are called *logical operations*. Thus we can define logical $\overline{X}$ and $\overline{Z}$ that are applied on our logical qubit as follows:

- logical bit-flip: $\overline{X}|\overline{0}\rangle = |\overline{1}\rangle, \overline{X}|\overline{1}\rangle = |\overline{0}\rangle$
- logical phase-flip: $\overline{Z}|\overline{0}\rangle = |\overline{0}\rangle, \overline{Z}|\overline{1}\rangle = -|\overline{1}\rangle$

As our stabilizer with $n-k$ generators stabilizes our logical basis states (meaning these states are $+1$ eigenvectors of the generators), then we can define them through the projectors of our stabilizer generators. Given an $n-$qubit Hermitian operator $g \in S$, whose square is the identity $g^2 = I$, the projection operator onto the subspace of states with $+1$ eigenvector is defined as

$$P_g = \frac{I+g}{2}$$

Therefore we can define our logical $|\overline{0}\rangle$ as the normalized product of all stabilizer generator $g_1, \dots, g_{n-k} \in S$ projectors times the $n-$qubit tensor of $|0\rangle$. In symbols:

$$|\overline{0}\rangle = \frac{1}{\sqrt{2^{n-k}}}(I+g_1)\cdot \ldots \cdot (I+g_{n-k}) \cdot |0\rangle_n$$

Similarly, we can define the logical $|\overline{1}\rangle$

$$|\overline{1}\rangle = \overline{X}|\overline{0}\rangle = \frac{1}{\sqrt{2^{n-k}}}(I+g_1)\cdot \ldots \cdot (I+g_{n-k}) \cdot |1\rangle_n$$

When we get errors during computations into our qubits, we call these corruptions. In this thesis we only deal with single gate X, Z and Y errors.

Definition 2.3.8. *The general form of a corrupted n-qubit state is defined as*

$$|e\rangle|\psi\rangle \rightarrow \left(|d\rangle 1 + \sum_{i=1}^{7}[|a_i\rangle]X_i + |b_i\rangle]Y_i + |c_i\rangle]Z_i\right)|\psi\rangle$$

where $|e\rangle$ is an "environment" state, summing up all quantum information different from $|\psi\rangle$.

To detect errors with our stabilizer codes we use measurements to see with which generators of the stabilizer the error anti-commuted with. Therefore by measuring we project the state into one of the $1 + 3n$ orthogonal 2-dimensional subspaces. 1 subspace is the uncorrupted one (subspace V_S stabilized by S) and $3n$ 2-dimensional subspaces for each of the 1-qubit errors. Thus the 2^n-dimensional space spanned by all of the states of n qubits must be large enough to contain $1 + 3n$ orthogonal subspaces. Thus giving us the limit to our n-qubit code space

$$2^{n-1} \geq 3n + 1$$

The smallest n satisfying this inequality is $n = 5$, for which the equality holds.

2.4 Pauli group homomorphism and bilinear form

Definition 2.4.1. *Let us have two groups $(X, \cdot_x)$ and $(Y, \cdot_y)$. A function $f : X \to Y$ is a (homo)morphism if $\forall x_1, x_2 \in X$:*

$$f(x_1 \cdot_x x_2) = f(x_1) \cdot_y f(x_2)$$

Definition 2.4.2. *Let X, Y be groups, $f : X \to Y$ an homomorphism. The image of f is the set*

$$\mathrm{im}(f) = \{f(x) : x \in X\}.$$

Lemma 2.4.3. $\mathrm{im}(f)$ *is a subgroup of Y.*

Proof. We know that the image is never empty, because the group homomorphism must preserve the identity $f(e_x) = e_y$. Let $h_1, h_2 \in (f)$. Then there exists $g_1, g_2 \in G$, such that $f(g_1) = h_1$ and $f(g_2) = h_2$. We need to show that (f) is closed under multiplication and inversion. We can see that (f) is closed under multiplication, since

$$h_1 h_2 = f(g_1) f(g_2) = f(g_1 g_2)$$

We can see that (f) is closed under inversion, since

$$h^{-1} = f(g)^{-1} = f(g^{-1})$$

Therefore (f) is a subgroup of H. ■

Definition 2.4.4. *Let X, Y be groups, $f : X \to Y$ an homomorphism. The kernel of f is the set*

$$\ker(f) = \{x \in X : f(x) = e_y\}.$$

Lemma 2.4.5. $\ker(f) \triangleleft X$

Proof. $\ker(f) \neq \emptyset$, since $e_X \in \ker(f)$. Indeed, $\forall x \in X$, $f(x) = f(e_X \cdot_X x) = f(e_x) \cdot_Y f(x) \Leftrightarrow f(e_X) = e_Y$, and $f(e_x) = e_y$. To see normality, let $h \in \ker(f)$ and $x \in X$. Then

$$f(xhx^{-1}) = f(x)f(h)f(x^{-1}) = f(x)f(x)^{-1} = e_y \Rightarrow xhx^{-1} \in \ker(f).$$

■

Theorem 2.4.6 (First isomorphism theorem)**.** *Let X, Y be groups, $f : X \to Y$ an homomorphism. Then* $\text{im}(f) \cong G/ker(f)$. ■

Definition 2.4.7. *Let X, Y be vector spaces over the same field $\mathbb{F}$. A function $f : X \to Y$ is a linear map if for any two vectors $\vec{v}, \vec{u} \in X$ and for any scalar λ, the following conditions hold:*

- $f(\vec{v} + \vec{u}) = f(\vec{v}) + f(\vec{u})$
- $f(\lambda\vec{v}) = \lambda f(\vec{v})$

Definition 2.4.8. *Let X, Y, Z be vector spaces over the same field $\mathbb{F}$. $f : X \times Y \to Z$ is bilinear if it is linear on X, Y separately.*

We start by defining a group homomorphism f from the Pauli group $\mathcal{P}_n$ on n-qubits to $\mathbb{Z}_2^{2n}$ with $\ker(f) =< iI >= \{iI, -I, -iI, I\}$. This mapping maps each Pauli group element to its corresponding binary vector.

Let us give an example on the 2 dimensional Pauli group $\mathcal{P}_1$. Let $\mathbb{Z}_2^2$ be the $\mathbb{Z}_2$-vector space $\mathbb{Z}_2^2 = \{(a, b) : a, b \in \mathbb{Z}_2\}$. Addition is defined component-wise, modulo 2. Scalar multiplication is defined as $0 \cdot \vec{v} = (0, 0) = \vec{0}, 1 \cdot \vec{v} = \vec{v} \quad \forall \vec{v} \in \mathbb{Z}_2^2$. $(\mathbb{Z}_2^2, +)$ is an abelian group of order 4. We are interested in a map $f : \mathcal{P}_n \to \mathbb{Z}_2^2$ defined by $f(\pm I) = f(\pm iI) = \vec{0}$. We can now give a table of this 2-dimensional homomorphism

$\mathcal{P}_1$	$\mathbb{Z}_2^2$
I	00
X	10
Z	01
Y	11

Table 1. homomorphism $\mathcal{P}_1 \to \mathbb{Z}_2^2$

The location of 1's (left, right, neither, both) play an important role in the construction of our mapping:

- For the Pauli I, we have no 1's on either side
- For the Pauli X, we have a 1 on the left side, and a 0 on the right
- For the Pauli Z, we have a 1 on the right side, and a 0 on the left
- For the Pauli Y, we have a 1 on both of the sides

We can now extend this homomorphism to Pauli group $\mathcal{P}_n$ on n qubits, $\forall n \in \mathbb{N}$. Let us showcase an example using the generators used in the 7-qubit Steane code [Ste96]:

$$g_1 = IIIXXXX = X_4X_5X_6X_7 \quad g_4 = IIIZZZZ = Z_4Z_5Z_6Z_7$$

$$g_2 = IXXIIXX = X_2X_3X_6X_7 \quad g_5 = IZZIIZZ = Z_2Z_3Z_6Z_7$$

$$g_3 = XIXIXIX = X_1X_3X_5X_7 \quad g_6 = ZIZIZIZ = Z_1Z_3Z_5Z_7$$

As our generators here have 7 elements, using the homomorphism we get representations with $2 \times 7 = 14$ elements. We can rewrite these using the homomorphism as

$$f(g_1) = 0001111|0000000 \quad f(g_4) = 0000000|0001111$$

$$f(g_2) = 0110011|0000000 \quad f(g_5) = 0000000|0110011$$

$$f(g_3) = 1010101|0000000 \quad f(g_6) = 0000000|1010101$$

We can use these binary vector rows to construct a matrix. We call this the *check matrix* *H*. The check matrix for the 7-qubit code would look as follows:

$$H = \left[\begin{array}{ccccccc|ccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right]$$

We use this homomorphism, because it enables us to use linear algebra to check relations between our Pauli group elements, which would otherwise be difficult to compute. For our case, this enables us to:

- Do calculation over $\mathbb{Z}_2$
- Check if any two Pauli group elements are independent
- Check if any two Pauli group elements commute

We also show this in the proofs below.

Lemma 2.4.9. *Let $g_1 \dots g_n$ be the generators of the stabilizer group S. Then the operators $g_1 \dots g_n$ are independent if and only if $f(g_1) \dots f(g_n)$ are linearly independent.*

Proof. Let us prove by contrapositive. Assume $f(g_1) \dots f(g_n)$ are linearly dependent, then there are coefficients $\alpha_1 \dots \alpha_{n-1} \in \mathbb{Z}_2$ (where some α are non-zero) so that:

$$\sum_{i=1}^{n} \alpha_i g_i = 0$$

We do not care about the arrangement of our equations and since not all α are 0, then we can say

$$f(g_n) = \alpha_1 f(g_1) + \dots + \alpha_{n-1} f(g_{n-1})$$

We know that due to our homomorphism $f(g) + f(g') = f(gg')$. We also know that $g_i^0 = I$ and $g_i^1 = g_i$ for all $i = 1, \dots, n$. Therefore we can say that

$$g_n = g_1^{\alpha_1} \times \dots \times g_{n-1}^{\alpha_{n-1}} \text{ modulo an element in } \ker(f/S).$$

As we previously stated that $\ker(f) =< iI >= \{iI, -iI, I, -I\}$, all elements in S are real and $-I \notin S$, then $\ker(f/S) = I$ and thus

$$g_n = g_1^{\alpha_1} \times \dots \times g_{n-1}^{\alpha_{n-1}},$$

which shows that $g_1, \dots, g_n$ are linearly dependent. ■

To show how we can check if any two Pauli group elements commute, let us introduce a $2n \times 2n$ matrix Λ, such that

$$\Lambda := \begin{bmatrix} 0_n & I_n \\ I_n & 0_n \end{bmatrix}$$

Lemma 2.4.10. *Two Pauli group elements g and g' commute if and only if*

$$f(g)\Lambda f(g')^T = 0$$

Proof. Let us have two Pauli group elements $g, g' \in \mathcal{P}_n$. We use our homomorphism to get $f(g) = \begin{bmatrix} a_1 & \dots & a_n & | & a_{n+1} & \dots & a_{2n} \end{bmatrix}$ and $f(g') = \begin{bmatrix} b_1 & \dots & b_n & | & b_{n+1} & \dots & b_{2n} \end{bmatrix}$, where $a_i, b_i \in \mathbb{F}_2$. By definition if g and g' commute, then

$$f(g)\Lambda f(g')^T = \begin{bmatrix} a_1 & \dots & a_n & a_{n+1} & \dots & a_{2n} \end{bmatrix} \begin{bmatrix} 0 & I_n \\ I_n & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ \dots \\ b_n \\ b_{n+1} \\ \dots \\ b_{2n} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{n+1} & \dots & a_{2n} & a_1 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \dots \\ b_n \\ b_{n+1} \\ \dots \\ b_{2n} \end{bmatrix} = a_{n+1}b_1 + \dots + a_n b_{2n} = 0$$

From this we can see that we multiply the "right side" elements of $f(g)$ with the corresponding "left side" elements of $f(g')$ and we multiply the "left side" elements of $f(g)$ with the corresponding "right side" elements of $f(g')$.

We let us now look at all of the possible combinations of single qubit Pauli Group element homomorphisms, where we multiply the right bit with the left bit and vice-versa. Let us start with anti-commuting Pauli element pairs:

$$f(X)\Lambda f(Z)^T = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1$$

$$f(Z)\Lambda f(X)^T = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$$

$$f(X)\Lambda f(Y)^T = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1$$

$$f(Y)\Lambda f(X)^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$$

$$f(Z)\Lambda f(Y)^T = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1$$

$$f(Y)\Lambda f(Z)^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1$$

As we can see, the anticommuting single qubit Pauli element pairs always give us a 1.

Now let us look at commuting Pauli element pairs:

$$f(X)\Lambda f(I)^T = \begin{bmatrix}0 & 1\end{bmatrix} \begin{bmatrix}0 \\ 0\end{bmatrix} = 0$$

$$f(I)\Lambda f(X)^T = \begin{bmatrix}0 & 0\end{bmatrix} \begin{bmatrix}1 \\ 0\end{bmatrix} = 0$$

$$f(Z)\Lambda f(I)^T = \begin{bmatrix}1 & 0\end{bmatrix} \begin{bmatrix}0 \\ 0\end{bmatrix} = 0$$

$$f(I)\Lambda f(Z)^T = \begin{bmatrix}0 & 0\end{bmatrix} \begin{bmatrix}0 \\ 1\end{bmatrix} = 0$$

$$f(I)\Lambda f(Y)^T = \begin{bmatrix}0 & 0\end{bmatrix} \begin{bmatrix}1 \\ 1\end{bmatrix} = 0$$

$$f(Y)\Lambda f(I)^T = \begin{bmatrix}1 & 1\end{bmatrix} \begin{bmatrix}0 \\ 0\end{bmatrix} = 0$$

$$f(X)\Lambda f(X)^T = \begin{bmatrix}0 & 1\end{bmatrix} \begin{bmatrix}1 \\ 0\end{bmatrix} = 0$$

$$f(Z)\Lambda f(Z)^T = \begin{bmatrix}1 & 0\end{bmatrix} \begin{bmatrix}0 \\ 1\end{bmatrix} = 0$$

$$f(Y)\Lambda f(Y)^T = \begin{bmatrix}1 & 1\end{bmatrix} \begin{bmatrix}1 \\ 1\end{bmatrix} = 0$$

As we can see, the commuting single qubit Pauli element pairs always give us a 0.

We can now extend this to our n-qubit Pauli group homomorphism. The n-qubit Pauli group element consists of tensors of single qubit Pauli group elements. When we use our homomorphism, the binary vector representation retains the order of these single elements (meaning we have the first bit of the single qubit Pauli element on the "left side" and the second bit of the single qubit Pauli element on the "right side", and continue this for all n single qubit Pauli elements). Thus if we yet again look at our equation

$$f(g)\Lambda f(g')^T = a_{n+1}b_1 + \ldots + a_n b_{2n} = 0$$

We can now understand that this is a sum of the single qubit Pauli Group element bilinear forms. And since we operate over $\mathbb{Z}_2$ if we have an even number of anticommutativite pairs (or none) at any of the n locations, then our Pauli group elements commute.

Let us give a 2 qubit example of this bilinear form. Let's say we have two Pauli group elements

$$g = IX = X_2 \quad g' = ZY = Z_1Y_2$$

We first map them using our homomorphism

$$f(g) = 01|00 \qquad f(g') = 01|11$$

Now we can see that

$$f(g)\Lambda(f(g'))^T = \begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}0 & 0 & 1 & 0\\0 & 0 & 0 & 1\\1 & 0 & 0 & 0\\0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}0\\1\\1\\1\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}0\\1\\1\\1\end{bmatrix} = 1 \neq 0$$

Therefore our g and g' do not commute.

Let us also check this using our Pauli group elements. We know that $Y = XZ$ and thus $XY = X(XZ) = (XX)Z = IZ = Z$. Therefore we can see that

$$(I \otimes X)(Z \otimes Y) = IZ \otimes XY = Z \otimes Z$$

We also can see that $YX = (XZ)X = -(XX)Z = -IZ = -Z$. Therefore we can see that

$$(Z \otimes Y)(I \otimes X) = ZI \otimes YX = Z \otimes (-Z) = -(Z \otimes Z)$$

Therefore we have checked that indeed g and g' do not commute.

Now let us define new Pauli group elements using the same variable names

$$g = ZX = Z_1X_2 \quad g' = YY = Y_1Y_2$$

which after mapping give us

$$f(g) = 01|10 \qquad f(g') = 11|11$$

Now we can see that

$$f(g)\Lambda(f(g'))^T = \begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix} \begin{bmatrix}0 & 0 & 1 & 0\\0 & 0 & 0 & 1\\1 & 0 & 0 & 0\\0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}1\\1\\1\\1\end{bmatrix} = 0$$

and thus these Pauli group elements g and g' commute.

Let us also check this using our Pauli group elements. We know that $Y = XZ$ and thus $ZY = Z(XZ) = -X(ZZ) = -XI = -X$. We also saw from the previous example that $XY = Z$. Therefore we can see that

$$(Z \otimes X)(Y \otimes Y) = ZY \otimes XY = -X \otimes Z = -(X \otimes Z)$$

We also can see that $YZ = (XZ)Z = X(ZZ) = XI = X$. We also saw from the previous example that $YX = -Z$ Therefore we can see that

$$(Y \otimes Y)(Z \otimes X) = YZ \otimes YX = X \otimes -Z = -(X \otimes Z)$$

Therefore we have checked that indeed g and g' do commute.

If we are given one Pauli group element, we can also use this to find all of the Pauli group elements, that commute with our given element. For example let's find all of the commuting Pauli group elements for

$$g = YX = Y_1 X_2$$

First we find the corresponding binary mapping

$$f(g_1) = 11|10$$

Let us also define a vector $(f(g'))^T = [\alpha, \beta, \gamma, \delta]$, where $\alpha, \beta, \gamma, \delta \in \mathbb{Z}_2$. This vector represents all the possible commuting elements. Now we can use the before-mentioned equation

$$f(g)\Lambda(f(g'))^T = \begin{bmatrix}1 & 1 & 1 & 0\end{bmatrix} \begin{bmatrix}0 & 0 & 1 & 0\\0 & 0 & 0 & 1\\1 & 0 & 0 & 0\\0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}\alpha\\\beta\\\gamma\\\delta\end{bmatrix} = \begin{bmatrix}1 & 0 & 1 & 1\end{bmatrix} \begin{bmatrix}\alpha\\\beta\\\gamma\\\delta\end{bmatrix} = \alpha+\gamma+\delta = 0$$

Based on this we can compose a table for all the possible values of g'

2.5 Calculating logical X and Z

Logical $\overline{X}$ and $\overline{Z}$ operations are the quantum operations that apply X and Z on the encoded logical qubit. To construct our logical $\overline{X}$ and $\overline{Z}$ gates it is easy to use the before-mentioned check matrix. As an example lets use the 7-qubit Steane code [Ste96] check matrix. To compose the check matrix, we map our Pauli group generators to bitstrings and use these as rows of the matrix

α	β	γ	δ		$\mathcal{P}_n$
0	0	0	0		II
0	0	1	1		ZZ
1	0	0	1		XZ
1	0	1	0	$\mapsto$	YI
0	1	0	0		IX
0	1	1	1		ZY
1	1	0	1		XY
1	1	1	0		YX

Table 2. Commuting Pauli group elements of $g = Y_1 X_2$

$$H = [G_1|G_2] = \begin{bmatrix} 0&0&0&1&1&1&1&|&0&0&0&0&0&0&0 \\ 0&1&1&0&0&1&1&|&0&0&0&0&0&0&0 \\ 1&0&1&0&1&0&1&|&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&|&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&|&0&1&1&0&0&1&1 \\ 0&0&0&0&0&0&0&|&1&0&1&0&1&0&1 \end{bmatrix}$$

We must transform the check matrix in to something that Nielsen & Chuang call the *standard form.* Note that $r = 3$ is the rank of G_1 and G_2, $n = 7$ is the number of physical qubits and $k = 1$ is the number of logical encoded qubits.

$$\begin{array}{rl} r & \{ \\ n-k-r & \{ \end{array} \begin{array}{cccccccc} \overbrace{I}^{r} & \overbrace{A_1}^{n-k-r} & \overbrace{A_2}^{k} & | & \overbrace{B}^{r} & \overbrace{0}^{n-k-r} & \overbrace{C}^{k} \\ 0 & 0 & 0 & | & D & I & E \end{array}$$

Since our Steane code check matrix consists of linearly independent rows, thus we can do row and column transformations so that we get a row-echelon form where needed. It is important to note that we can only switch the columns either on the left side (G_1) or on the right side (G_2), but not between the sides. This corresponds to switching the locations of qubits - therefore if we switch columns in G_1, we have to switch the same rows in G_2 and vice-versa. It is important to keep track of qubit location swaps, as our final logical operations have to be given in the original qubit placement format.

Let us format our matrix similarly as shown above

$$H = \left[\begin{array}{ccccccccccccccccccc}
0 & 0 & 0 & | & 1 & 1 & 1 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
0 & 1 & 1 & | & 0 & 0 & 1 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
1 & 0 & 1 & | & 0 & 1 & 0 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
- & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 0 & 0 & 0 & | & 1 & 1 & 1 & | & 1 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 0 & 1 & 1 & | & 0 & 0 & 1 & | & 1 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 1 & 0 & 1 & | & 0 & 1 & 0 & | & 1
\end{array}\right]$$

Now we want to change the top left submatrix into the identity matrix. We can do this with the following operations:

$$col_1 \Longleftrightarrow col_4$$

$$col_3 \Longleftrightarrow col_4$$

After the operations we get the following matrix

$$H = \left[\begin{array}{ccccccccccccccccccc}
1 & 0 & 0 & | & 0 & 1 & 1 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
0 & 1 & 0 & | & 1 & 0 & 1 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
0 & 0 & 1 & | & 1 & 1 & 0 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
- & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 1 & 0 & 0 & | & 0 & 1 & 1 & | & 1 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 0 & 1 & 0 & | & 1 & 0 & 1 & | & 1 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 0 & 0 & 1 & | & 1 & 1 & 0 & | & 1
\end{array}\right]$$

Now we want to change the bottom fifth submatrix into the identity matrix. We can do this with the following operations:

$$col_4 \Longleftrightarrow col_7$$

$$row_5 \rightarrow row_5 + row_4$$

$$row_6 \rightarrow row_6 + row_4$$

$$row_4 \rightarrow row_4 + row_5$$

$$row_4 \rightarrow row_4 + row_6$$

After the operations we get the following matrix

$$H = \left[\begin{array}{ccccccccccccccccccc}
1 & 0 & 0 & | & 1 & 1 & 1 & | & 0 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
0 & 1 & 0 & | & 1 & 0 & 1 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
0 & 0 & 1 & | & 1 & 1 & 0 & | & 1 & | & 0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 \\
- & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 1 & 1 & 1 & | & 1 & 0 & 0 & | & 0 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 1 & 1 & 0 & | & 0 & 1 & 0 & | & 1 \\
0 & 0 & 0 & | & 0 & 0 & 0 & | & 0 & | & 1 & 0 & 1 & | & 0 & 0 & 1 & | & 1
\end{array}\right]$$

Given the standard form it is easy to get the logical $\overline{Z}$ operation. We have to pick k operators, that are independent with each other and the generators of our stabilizer. The operators also have to commute with each other and the generators of our stabilizer. As we are looking at cases, where we only have one logical qubit, then $k = 1$. We can write the check matrix for the encoded $\overline{Z}$ operators as $H_z = [F1F2F3|F4F5F6]$, where the number of columns of each matrix F_i follow the structure our the standard form (so, the first matrix has r rows, second $n - k - r$, and so on. We choose these matrices so that $H_Z = \begin{bmatrix} 0 & 0 & 0 & | & A_2^T & 0 & I \end{bmatrix}$. In a similar fashion we can define $\overline{X}$ as $H_X = \begin{bmatrix} 0 & E^T & I & | & C^T & 0 & 0 \end{bmatrix}$

It is also easy to see that these $\overline{Z}$ operators are independent from the generators. The first r rows of H only have X operator mappings and these do not coincide with Z operator mappings (X operators have 1's on the left whereas Z operators have 1's on the right). The remaining $n - k - r$ rows of H are independent since the $n - k - r \times n - k - r$ identity matrix is present for those generators and they do not have any corresponding terms.

As for our Steane code check matrix $A_2^T = (0, 1, 1)$, then

$$H_Z = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & | & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

which corresponds to $\overline{Z} = Z_2 Z_3 Z_7$. After re-switching the positions of qubits into the right positions, we get that $\overline{Z} = Z_1 Z_2 Z_3$.

Similarly we know that $E^T = (0, 1, 1)$ and $C^T = (0, 0, 0)$, thus

$$H_X = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & | & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

which correspons to $\overline{X} = X_5 X_6 X_7$. After re-switching the positions of qubits into the right positions, we get that $\overline{X} = X_3 X_5 X_6$.

Since the stabilizers and logical operators are independent and commute, we can multiply them to get better forms for our logical operators (as the subspace stays the same). For example if we multiply $\overline{Z} = Z_1 Z_2 Z_3$ by the generator $g_4 = Z_4 Z_5 Z_6 Z_7$, we get a better form of $\overline{Z} = Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7$ as this is the same as applying the Z operation on every physical qubit. The same applies to $\overline{X}$.

Let us check the correctness of our logical $\overline{Z}$ and $\overline{X}$. To do that we must define our $|\overline{0}\rangle$. As we stated in the stabilizer theory chapter we can do this using the stabilizer generator

projectors. We do not care about adding projectors of g_4, g_5, g_6, since they consist of only Z operators and thus leave $|0\rangle_7$ invariant. Therefore the Steane code logical qubits are defined as

$$|\overline{0}\rangle = \frac{1}{\sqrt{8}}(1+g_1)(1+g_2)(1+g_3)|0\rangle_7 = \frac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle)$$

$$|\overline{1}\rangle = \frac{1}{\sqrt{8}}(1+g_1)(1+g_2)(1+g_3)|1\rangle_7 = \frac{1}{\sqrt{8}}(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle)$$

Now we can check if our calculated logical operators behave as expected. Let us first check our logical $\overline{Z}$.

$$\overline{Z}|\overline{0}\rangle = \frac{1}{\sqrt{8}}Z_1Z_2Z_3(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle) = \frac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle) = |\overline{0}\rangle$$

$$\overline{Z}|\overline{1}\rangle = \frac{1}{\sqrt{8}}Z_1Z_2Z_3(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle) = \frac{1}{\sqrt{8}}(-|1111111\rangle - |0101010\rangle - |1001100\rangle - |0011001\rangle - |1110000\rangle - |0100101\rangle - |1000011\rangle - |0010110\rangle) = -|\overline{1}\rangle$$

We can see that the logical $\overline{Z}$ behaves as expected. Now let us check our logical $\overline{X}$.

$$\overline{X}|\overline{0}\rangle = \frac{1}{\sqrt{8}}X_3X_5X_6(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle) = \frac{1}{\sqrt{8}}(|0010110\rangle + |1000011\rangle + |0100101\rangle + |1110000\rangle + |0011001\rangle + |1001100\rangle + |0101010\rangle + |1111111\rangle) = \frac{1}{\sqrt{8}}(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle) = |\overline{1}\rangle$$

$$\overline{X}|\overline{1}\rangle = \frac{1}{\sqrt{8}}X_3X_5X_6(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle) = \frac{1}{\sqrt{8}}X_3X_5X_6(|1101001\rangle + |0111100\rangle + |1011010\rangle +$$

$|0001111\rangle + |1100110\rangle + |0110011\rangle + |1010101\rangle + |0000000\rangle) = \frac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle) = |\overline{0}\rangle$

Therefore we have shown how to both find the logical X and Z gates given a set of commuting multi Pauli generators and checked that they work (in this case using the Steane code as an example).

3 Implementation

The software, based on Python, takes in a list of Pauli group elements and gives back a list of possible logical Z and X operators. We implement this algorithm in the following steps:

- Input parsing
- Finding maximum commuting list of operators
- Solving of systems of linear equations over $\mathbb{F}_2$
- Calculating logical Z and X operators

After explaining the implementation of each part, we introduce the complexity and performance of our algorithm. We will test the algorithm with the 5 qubit [LMPZ96], 6 qubit [SWO$^+$08] 7 qubit (Steane code) [Ste96], 9 qubit (Shor code) [Sho95], 11 qubit [Got97], 17 qubit and 19 qubit [CB18] error correction codes.

3.1 Input parsing

The input is read off a .txt file, in the following format

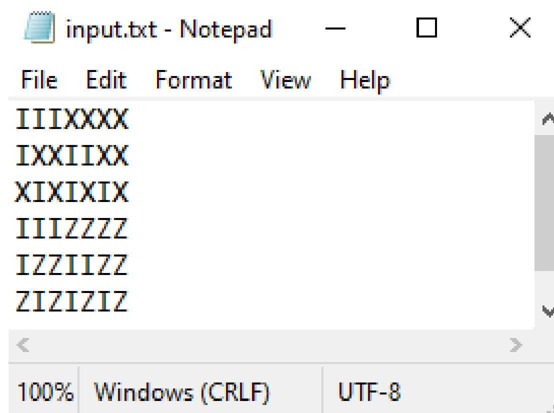

Figure 1. Example input of the Steane code generators

On each line we have a Pauli group element which is written out with single qubit Pauli operators. As a first step, these elements are in collected into a list.

We need to use our homomorphism to transform these Pauli group elements into their corresponding binary bitstrings. To do this we split each Pauli group element into single

qubit Pauli operators. Then we start adding the left bits of these operators into one list and the right bits into another list. This is better visualised by the figure 2 below.

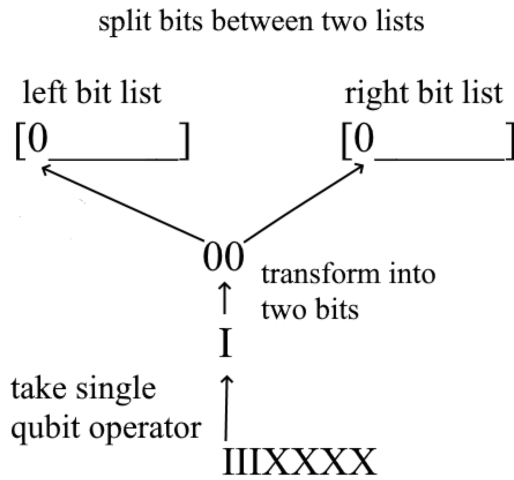

Figure 2. The splitting of single Pauli operator bitstrings

Then we merge these two lists into one whole list

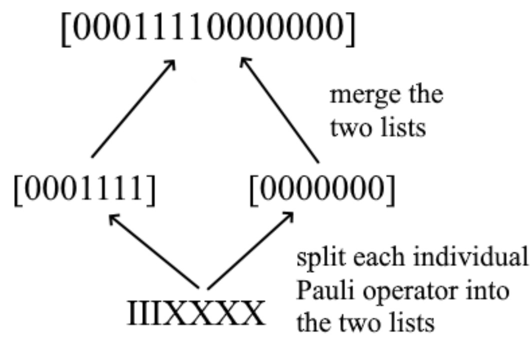

Figure 3. The merging of left and right substrings

This is therefore done for each Pauli group element given. What we are left with is all of the binary representations of our given Pauli group elements.

Given that n is the number of physical qubits, k is the number of information qubits (in our case 1). To do this we need to do $(n-k)\cdot n\cdot 2+1=2n^2-2n+1$ operational

steps. In the case where we are given more generators than $n-1$, then this process takes $2qn+1$ steps, where q is the amount of generators.

3.2 Finding maximum commuting list of operators

After parsing each of the Pauli group elements and transforming them into their binary representations, we can start to check if these elements commute. We check commutativity of two elements in our set using the lambda matrix defined in previous chapters. We create the lambda matrix by initializing a $2n \times 2n$ matrix of zeroes and then manually adding the ones where necessary. As there are $2n$ ones in the lambda matrix, then creating the lambda matrix takes $1+2n$ steps. Alternatively one could create an $n \times n$ identity and zero matrices and merge them (but that would take about the same amount of steps). We also have to check if every one of our input Pauli group elements are of the same size, since we can not do vector-matrix multiplication if their dimensions do not match.

$$0_{2n\times 2n} = \begin{bmatrix} 0_n & 0_n \\ 0_n & 0_n \end{bmatrix} \xrightarrow{\text{adding of ones}} \begin{bmatrix} 0_n & I_n \\ I_n & 0_n \end{bmatrix} = \Lambda_{2n\times 2n}$$

Our aim is to find the largest set of commuting Pauli group elements given our initial set. The easiest (though not computationally efficient) way to do this is to check with what generators every single generator commutes with (that is see if the result of the bilinear form is 0). This means we will have q lists of commuting elements, where q represents the amout of Pauli elements in our initial input. We also know that $q \geq n-1$. A mock example can be seen below for 6 generators (let us imagine that this is a 5 qubit system):

$$\begin{aligned}
g_1 &\text{ commutes with elements } [g_1, g_2, g_3, g_4, g_6] \\
g_2 &\text{ commutes with elements } [g_1, g_2, g_3, g_6] \\
g_3 &\text{ commutes with elements } [g_1, g_2, g_3, g_4, g_6] \\
g_4 &\text{ commutes with elements } [g_1, g_3, g_6] \\
g_5 &\text{ commutes with elements } [g_5] \\
g_6 &\text{ commutes with elements } [g_1, g_2, g_3, g_4, g_6]
\end{aligned}$$

After we have created all of the commuting lists for each element, we check if the elements within these lists also commute with each other. We start with the largest lists. If the elements do not commute, then we take the next largest list. If the elements do commute, then we return the list as it is the maximal commuting set. In the case where the size of the list itself is smaller than $n-1$, then we stop the program, because our

stabilizer must include $n-k$ elements to stabilize the subspace. So using the example given above, we would start by checking lists associated with g_1, g_3 and g_6. But as these do not include elements that all commute with each other, then we would move on to the list associated with g_4. As this list does compose of all commuting elements, then we return the list. As the length of the list is $n-1=5-1=4$, then it is suitable for our error correcting code.

To create our initial list of commuting sets, it takes us, in the worst case $2q^2$ steps. In the best case it takes us $2(n-1)^2$ steps. Finding the largest commuting subset from our list takes in the worst case q^3 steps and in the best case $(n-1)^2$ steps. Knowing this we can see that the whole process in worst case takes $1+2n+2q^2+q^3$ steps and in best case $1+2n+2(n-1)^2+(n-1)^2 = 3n^2-4n+4$ steps.

3.3 Solving of systems of linear equations over $\mathbb{F}_2$

We remember from subchapter 2.5, that we use our bitstrings to compose a check matrix. This check matrix had in turn be changed into the standard form using row and column transformations.

$$\begin{array}{rc|ccc|ccc} & & \overbrace{}^{r} & \overbrace{}^{n-k-r} & \overbrace{}^{k} & \overbrace{}^{r} & \overbrace{}^{n-k-r} & \overbrace{}^{k} \\ r & \{ & I & A_1 & A_2 \quad | & B & 0 & C \\ n-k-r & \{ & 0 & 0 & 0 \quad | & D & I & E \end{array}$$

We see that our check matrix splits into a left submatrix and right submatrix. We compose these left and right submatrices by using our left and right bit lists, we created in our previous step. Now we can start to get the identity into both of the submatrices using Gaussian elimination.

Since we work over a binary field, we only need to use addition (or multiplication) to do row and column transformations inside our matrix. The addition of two binary vectors is easily done by using the XOR operation. Lets imagine we have two vectors $x = \begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix}$ and $y = \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix}$. We can calculate the xor of two binary vectors as

$$x \oplus y = \begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix} \oplus \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix} =$$
$$= \begin{bmatrix}1\oplus 1 & 0\oplus 0 & 0\oplus 1 & 1\oplus 0 & 0\oplus 0\end{bmatrix} = \begin{bmatrix}0 & 0 & 1 & 1 & 0\end{bmatrix}$$

This is sufficient if we have two 1's in one column and we want to eliminate one of them (as is common in Gaussian elimination). In the case where our columns are more dense (meaning there are several 1's), we can use the outer product to do operations on all rows

where within one column 1's coincide. We will show this using an example matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

As we want to eliminate all of the 1's in the pivot (red) column. The trivial way to do this would be to do XOR operations with the pivot row and the bottom two rows. Instead we can take the pivot column and row.

$$p_{col} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad p_{row} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Since we want to avoid XORing our pivot row, we will make our pivot in the pivot column zero. We are left with

$$p_{col} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad p_{row} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Now we can take the outer product of those two (denoted here using $\otimes$)

$$p_{col} \otimes p_{row} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} = p_{outer}$$

As a final step we take the XOR of the outer product with our original matrix M

$$M \oplus p_{outer} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \oplus \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

This method is good to use when we do not want to traverse through the whole matrix several times. This can be seen better in the next step of this matrix. We choose the new pivot (blue)

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

As we do not care what resides above and left of the pivot. We can easily take the submatrix of the large matrix, where the pivot element is the upper left element.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Now we repeat the process that was explained above (only that we change the actual bits in our original matrix). This reduces the computation time with each iteration as the last iteration only has a 2×2 submatrix (if our matrix is square). After we have reached row-echelon form, we must eliminate all ones that are above the diagonal as well to reach the identity matrix form.

It is important to note that we can only use this way of row transformations if we have a non-zero pivot. In case we do have a zero pivot, we first try to get a non-zero pivot by looking for a suitable row to switch our pivot row with. If there are no suitable rows to switch, we check if there are any suitable columns to switch. If a row/column with a non-zero pivot element is found, then we proceed with the switch and then do the calculation explained beforehand.

We start with the left $n-1 \times n-1$ submatrix. It is trivial to see that the complexity of our first Gaussian elimination is $O(r^3)$, where r is the rank of the left submatrix. The right submatrix behaves a bit different from the left one. Remember that our standard form was

$$\begin{array}{rl} r & \{ \\ n-k-r & \{ \end{array} \begin{array}{cccccc} \overbrace{I}^{r} & \overbrace{A_1}^{n-k-r} & \overbrace{A_2}^{k} & | & \overbrace{B}^{r} & \overbrace{0}^{n-k-r} & \overbrace{C}^{k} \\ 0 & 0 & 0 & | & D & I & E \end{array}$$

In the case where $n-k=r$ (as in the five qubit error correcting code), we can see the standard form as

$$r \ \{ \ \overbrace{I}^{r} \ \ \overbrace{A_2}^{k} \ | \ \overbrace{B}^{r} \ \ \overbrace{C}^{k}$$

This means that in the cases where $n-k=r$ we actually do not need to do Gaussian elimination in the right submatrix. Therefore the best case complexity for solving the right submatrix is $O(1)$ and worst case $O((n-1-r)^3)$. We have to not with both the left and right submatrices, that if we do an operation on the right submatrix, we must follow the same operation on the right submatrix. Which means that in reality the amount of steps necessary to get our result is doubled in the end.

3.4 Calculating logical Z and X operators

Once we have the desired standard form, we can calculate our logical X and Z binary vector representations. The equations for the logical X and Z are

$$H_Z = \begin{bmatrix} 0 & 0 & 0 & | & A_2^T & 0 & I \end{bmatrix}$$
$$H_X = \begin{bmatrix} 0 & E^T & I & | & C^T & 0 & 0 \end{bmatrix}$$

The length of these vectors in the equations follow the same structure as the lenghts of the submatrices in the standard form (so $r, n-k-r, k$ for both sides). Therefore we can trivially conclude that constructing these vectors takes $2n$ steps for both logical $\overline{Z}$ and $\overline{X}$ operations. This way we get two possible versions of our logical $\overline{Z}$ and $\overline{X}$ operations.

However we are also interested in nice alternatives for our initial logical operators. By nice, we mean logical operators that do not have a mix of X, Y or Z single gate operators. We achive this by taking linear combinations of our initial logical operators and out stabilizer generators.

We start by intializing an empty set that holds distinct elements. We have three nested for-loops, where we iterate through our $n-1$ generators (in their binary forms). We choose to do this three times, so that we do not raise the complexity for this over $O(n-1)^3$. In each loop we first add the logical operator into our set and then multiply it by a generator. Each multiplied result is also added into the set. By the end we are left with a set with distinct logical X and Z operators which are our initial logical operators that have been multiplied by $0, 1, 2$ or 3 generators. It is important to note that this list is not a list of all of the possible logical $\overline{Z}$ and $\overline{X}$ operations, but rather a selection of some (searching for all of the possible combinations would be computationally inefficient).

After we have the set of our possible generators we sort out those that are not nice (that are not a mix of X, Y and Z). To do this we first convert our set of possible logical operators back to their Pauli group format (text strings in Python). Then we check each of the operator for a mix of letters. We only need to find the first two differing letters in each string. Those that have a mix are removed from the set. This filtering takes $(n-1)^3 \cdot n$ steps. The final solution of nice logical X and Z operators are written into two files, which takes as many steps as there are solutions for the logical X and Z operators (worse case is $(n-1)^3$ steps).

3.5 Complexity and performance

This subschapter introduces the complexity and shows the performance of the algorithm (noted as QEC algorithm) given a growing set of inputs. We analyse our algorithm by the 4 phases it goes through.

The first phase - input parsing, took $2qn + 1$ steps. We consider the case, where the amount of input Pauli group elements $q > n - 1$. This is because one of the stages looks for the suitable maximal commuting set for which the input can be larget than $n-1$.

The second phase - finding maximum commuting operator list, took in the worst case $1 + 2n + 2q^2 + q^3$ steps and in the best case $3n^2 - 4n + 4$ steps. We will use the best case when analysing the performance results here, since our test inputs are also proven to give a definite answer with $n - 1$ generators.

The third phase - solving Gaussian elimination (twice), has the complexity $O(r^3) + O((n - 1 - r)^3) \approx O(r^3)$. In CSS (Calderbank-Shor-Steane) codes [NC10], the generators only compose of "X and I" or "Z and I" operators. This is why in many CSS codes, the rank of both submatrices is half the amount of generators (so $r = (n - 1)/2$) since there are as many generators composing of X-s as there are generators composing of Z-s. This is why we approximate the complexity to be around $O(r^3)$ for this phase. Consider here that we use the worst case complexity for solving the right submatrix, since in most inputs, we do not encounter the edge-case where $n - 1 = r$.

The fourth and final phase - finding the nice set of commuting Pauli group generators is the most computationally inefficient. The finding of possible operators takes $(n - 1)^3$ steps. Filtering the result takes $(n-1)^3 \cdot n$ steps. Writing the result to the output file takes at worst case $(n-1)^3$ steps. This means the total amount of steps for this process can be as high as $n^3-3n^2+3n-1+n^4-3n^3+3n^2-n+n^3-3n^2+3n-1 = n^4-3n^2+5n-2$ steps.

The inefficiency of the last phase of the algorithm thus can be expected to take the most of the running time. In total, considering all phases of the algorithm, the whole complexity is $O(n^4)$, which is highly inefficient, when we consider that IBM aims to build a quantum computer with over 1000 qubits by 2023 [ACJG21]. Therefore we can assume that future developments of the algorithm must find better ways of finding these "nice" formats for logical operations that don't rely on multiplying the generators with the initial logical operation.

We ran the algorithm with sets of Pauli group operators, where the set included atleast $n-1$ Pauli group elements that commuted with each other. We tested with sizes of $5, 6, 7, 9, 11, 17$ and 19 qubits. The simulations we ran can bee also seen on Figures 4 and 5. When analysing the performance of each individual phase (subpart), we split our graphs into two - linear scale and log scale. We did this to better highlight the differences in complexities and overall performance of the phases.

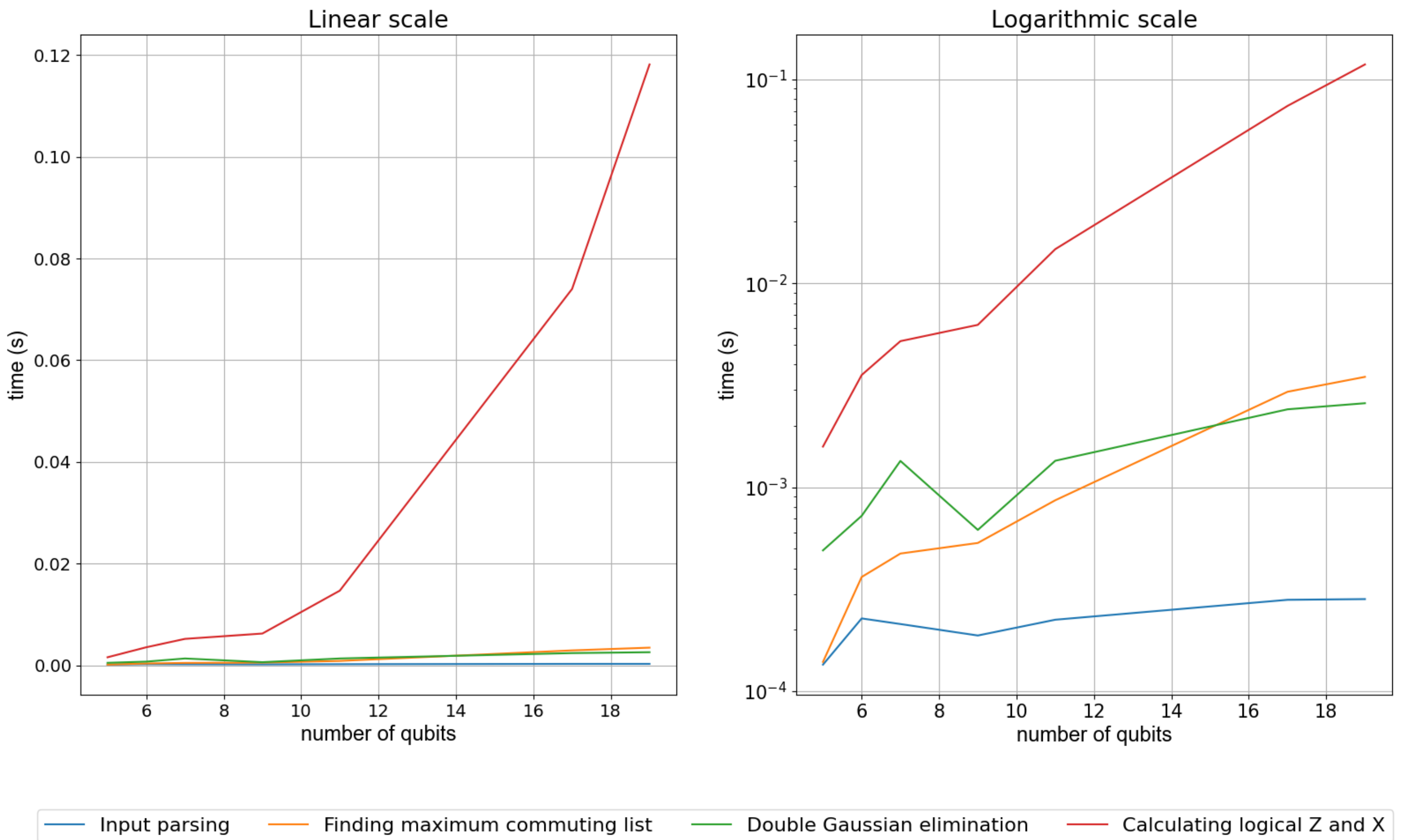

Figure 4. Time performance of different subparts of the QEC algorithm

The linear scale graph clearly highlights the point made previously, that the algorithm highly depends on the last part of our algorithm, where we generate the set of nice logical oprators. While this process takes quite a lot of time, most of the other processes perform similarly (which is expected given that their complexities are also similar). With smaller sets (as with which we experimented with) the running time is trivial (under a second),

but we have to keep in mind that with inputs with more than 1000 qubits, this running time can extend to tens of minutes, if not hours. While in single cases it does not appear that bad, we have to keep in mind, that in real quantum systems these calculations often have to be done after each quantum operation to detect and correct potential disturbences.

The logarithmic scale brings a more comprehensive view of the performance related differences between our phases. Here we can see that input parsing takes the least time (which makes sense considering that the complexity is approximately $O(n^2)$. Finding the maximum commuting list and Gaussian elimination both tae approximately the same amount of time (since their complexity is about $O(n^3)$). And the final part takes the most as calculating the logical operator sets has the complexity of about $O(n^4)$.

It is important to note that the running time of calculating the logical Z and X operators is also raised due to the fact that the results have to be written into a .txt files after calculating. Steps related to accessing and writing the files takes more time than the average calculation steps.

We can also see a small anomaly in the results - the 9 qubit input running time is unexpectedly smaller than expected (especially the Gaussian elimination parts). This is due to the fact that the Shor code check matrix is quite sparse and thus the algorithm solves Gaussian Elimination faster than for the 7 or 11 qubit inputs (which have check matrices that are more dense).

We also tested the full running time of the algorithm given the same examples. From the total performance graph we can see a similar trend as in the linear subparts graph - the whole running time is highly dependent on the last process and thus running the algorithm yields similar results (complexity of $O(n^4)$). It is important to note that even at that complexity the algorithm does not produce a deterministic output as we do not check all of the possibilities of logical X and Z variants (complexity of that would be factorial and unreasonable). This might also be the reason why this algorithm hasn't been used in quantum error correction software thus far.

Total running time could be potentially lowered by removing the output file writing (and instead feeding the output into the connecting algorithm for which it is needed). We can also lower the running time by using the algorithm with a compiled language such as Java or C++.

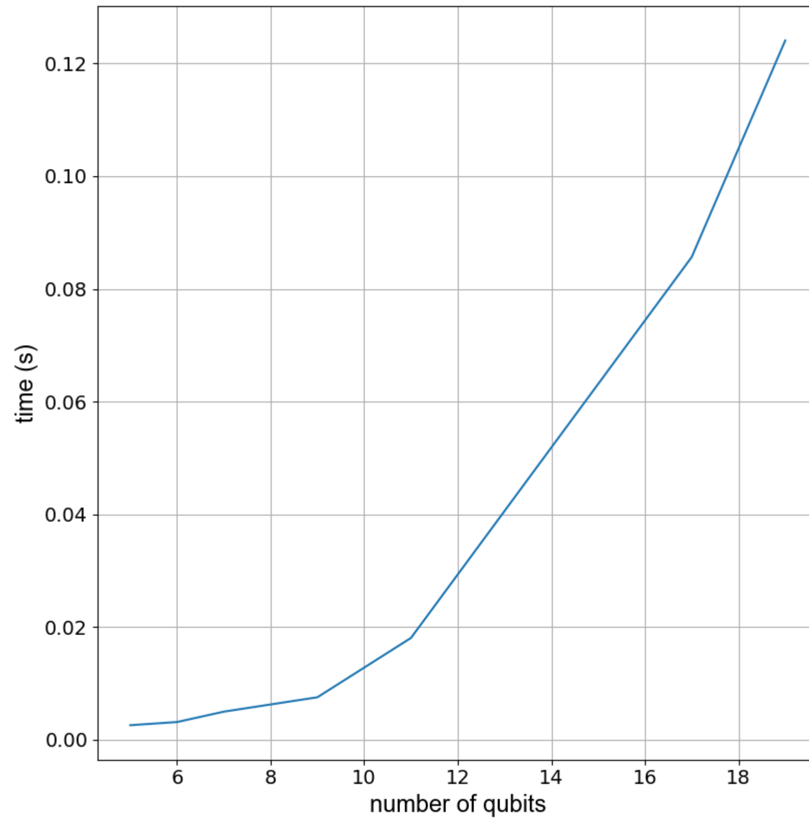

Figure 5. Time performance of the QEC algorithm

4 Conclusion

The thesis discussed a method for computing logical X and Z gates given a set of Pauli group elements - a process often used in quantum error correction. This problem was chosen for the thesis due to the fact that currently available software solutions do not support this functionality as of the time of writing this thesis. The theory of the algorithm is based on [NC10] and is implemented in Python.

Within the thesis we showed:

- The required mathematical background from group theory and stabilizer theory. We showed how we can theoretically use a set of Pauli group generators to find suitable logical X and Z operators for the logical qubit that is defined through the Pauli group generators.

- We implemented the algorithm in Python and explained different ways to implement the subparts of the algorithm

- We ran tests with different sets of Pauli group elements. These sets were differentiated by the number of qubits of the Pauli group. Tests were run with growing number of qubits $5 \leq n \leq 19$.

- We analysed the complexity and performance of the algorithm through the tests. We also drew attention to the computationally inefficient parts of the algorithm and how to potentially speed these processes up.

The resulting algorithm can be used in future research as a research software tool. By separating the input-output from the program itself, we have also allowed this algorithm to be integrated into other software tools more easily.

In the future this algorithm can be made more computationally efficient by developing new methods for finding a whole set of logical operator solutions given a single logical operator. In addition the running time of the algorithm can be brought down by using parallel computation (given the algorithm has a lot of nested loops, it also has a lot of opportunities to optimize through parallelization). Future research on this topic can look into expanding this software to work with dynamically changing logical qubits (such as are used with subsystem codes).

References

[Aar13] Scott Aaronson. *Skepticism of quantum computing*, page 217–227. Cambridge University Press, 2013.

[ACJG21] Anthony Annunziata, Jerry Chow, Blake Johnson, and Jay Gambetta. Ibm's roadmap for scaling quantum technology, Feb 2021.

[Bre19] Nikolas P. Breuckmann. The present and future of quantum error correction, Sep 2019. A short talk on quantum error correction at the UCLQ Annual Industry Event 2019. It is aimed towards a broad audience and gives a brief overview over the field as well as an outlook towards the coming 10 years.

[CB18] Christopher Chamberland and Michael E. Beverland. Flag fault-tolerant error correction with arbitrary distance codes. *Quantum*, 2:53, feb 2018.

[Got96] Daniel Gottesman. Class of quantum error-correcting codes saturating the quantum hamming bound. *Phys. Rev. A*, 54:1862–1868, Sep 1996.

[Got97] Daniel Gottesman. Stabilizer codes and quantum error correction, 1997.

[Gra12] Granade, Cassandra and Criger, Ben. *QuaEC: Library for working with quantum error-correcting codes (QECCs).* University of Waterloo, 2012.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[Lan05] S. Lang. *Algebra*. Graduate Texts in Mathematics. Springer New York, 2005.

[LMPZ96] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. Perfect quantum error correcting code. *Phys. Rev. Lett.*, 77:198–201, Jul 1996.

[Mer07] N. David Mermin. *Quantum error correction*, pages 99–135. Cambridge University Press, 2007.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*, pages 425–499. Cambridge University Press, 2010.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[Sho95] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995.

[Ste96] Andrew Steane. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 452(1954):2551–2577, nov 1996.

[SWO+08] Bilal Shaw, Mark M. Wilde, Ognyan Oreshkov, Isaac Kremsky, and Daniel A. Lidar. Encoding one logical qubit into six physical qubits. *Physical Review A*, 78(1), jul 2008.

[Tuc20] David Kingsley Tuckett. *Tailoring surface codes: Improvements in quantum error correction with biased noise*. PhD thesis, University of Sydney, 2020. (qecsim: `https://github.com/qecsim/qecsim`).

Appendix

I. Source code

The algorithm introduces in the thesis was implemented in the Python programming language. It was uploaded to Github and the instructions on how to run the program are found there. The reposatory also includes the examples which we used to run the performance tests. Link to Github repository: `https://github.com/ukangur/QEC`

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Uku Kangur**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Computing logical X and Z gates for stabilizer codes,

   supervised by Francisco Javier Gil Vidal and Dirk Oliver Theis.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Uku Kangur
16/05/2022