Universiteit Leiden
The Netherlands

Opleiding Informatica

Vecs2Pauli - an Algorithm
for Finding Stabilizers and
Transformations of Quantum States

Lieke Vertegaal

Supervisors:
Tim Coopmans (daily supervisor) & Alfons Laarman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

10/07/2023

Abstract

Quantum stabilizer states have many uses in different quantum fields, such as quantum error correction and classical simulations. We propose a recursive algorithm that can be used to determine whether a state is a stabilizer state, as well as to determine what Pauli operators transform one quantum state to another. This algorithm can be used in classical simulations, for hypothesis checking or for educational purposes. The algorithm is based on the LIMDD algorithm by Vinkhuijzen et al. [VCE+22].

Contents

1 Introduction

In recent times, quantum computing has become more widely known as well as used. While quantum computers are certainly not mainstream yet, there is no denying this rise and the potential advantages it has. But why has it been receiving more attention and what could some of these advantages be?

Regular computing (as we have been using for the past few decades) is based on representing information as bits and manipulating those bits. Bits have 2 possible values: they are either 0 or 1. The bits are manipulated using gates, made up of transistors. For decades, computing power has been increasing, but more recently the growth of chip sizes is slowing down [MM19]. However, while the tasks performed by computers are ever-growing, chip sizes are not. To try to keep up with the growing problem sizes, other computing ways must be explored.

This is where quantum computing comes in. Already in 1982 Feynman came with the notion that quantum physics could be useful in computing [Fey18]. An important advantage of quantum computing is that certain algorithms can have up to an exponential speed-up when compared to classical computing. This means that the order of magnitude of the number of operations performed by the original algorithm is exponentially larger than that of the new algorithm. It is important to note that this order of magnitude of number of operations is measured in terms of the input size of an algorithm. As a quick example, take a problem with input size 32. A 'slow' algorithm may perform $2^16 = 4.3 \cdot 10^9$ operations, while a new algorithm performs only 32 operations. Since $log_2(4.3 \cdot 10^9) = 32$, this signifies an exponential speed-up.

An example of an algorithm with such speed-up is the black box graph traversal problem stated by Childs et al [CCD+03], which can be solved exponentially faster on a quantum computer than a regular one. Furthermore, quantum computing could be very useful in optimization problems, which have various industrial applications [S.19]. Quantum computing is also very important for cryptography, since many cryptographic algorithms are based on the fact that finding a prime factorization is computationally very hard. In 2016, a (powerful) quantum computer could factor a 200-bit number in a day using Shor's algorithm [Sho94], while it took hundreds of computers over 2 years to factor a 768-bit number [Mon16].

However, when building a functional quantum computer, engineers face a lot of challenges [ALF+17]. Many of those are solved theoretically but not in practice. A big issue is the presence of *noise* in quantum computers. Noise is the effect that small environmental changes have on the state of the quantum computer, altering the state from the expected state to a slightly different one. In order to find out how noisy a quantum computer is allowed to be while still giving correct answers, is by simulating the quantum computer on a regular computer. We will refer to simulating a quantum computer on a regular computer as *classical simulation*. Noise levels from small, built quantum computers can be measured and the effect that the noise level would have on a larger computer can be measured by doing performance analysis on the simulated version.

Simulating quantum computing efficiently is a challenge, partly due to the amount of computer memory that is necessary. While it will be explained in more detail in Section 2.2, in order to store

a quantum state on a regular computer, we need to store a number of values that is exponential in the number of qubits (quantum bits) we want to simulate. This is doable for small numbers of qubits, but it quickly becomes an issue for slightly larger numbers of qubits. If we assume a RAM memory of 8GB, and floating point numbers of 8 bytes, we can store 1 billion floating point values, which would be enough for representing one quantum state of 29 qubits.

The good news is that there are ways to more efficiently represent these quantum states. That way, only a linear amount of 'values' need to be stored. The bad news, this representation does not work for all quantum states. In some cases, when certain quantum gates have been applied, it is necessary to save the entire vector (exponential in the number of qubits). But when possible, it would be preferred to store the linear amount, which is where Vecs2Pauli, the algorithm we propose, could come in handy. The Vecs2Pauli algorithm can be used to figure out if a quantum state can be saved in the shorter representation or whether the entire vector is necessary.

While this might seem very useful for simulating large quantum computers, a big drawback is the fact that Vecs2Pauli's input grows exponentially in the number of qubits. The algorithms runtime is polynomial in the input size, but if the input size is too big to be able to run the algorithm, a good runtime doesn't matter. However, the algorithm could be quite useful in hypothesis testing as well as for educational purposes.

The approach of the Vecs2Pauli algorithm is a translation from the decision-diagram approach in a paper by Vinkhuijzen et al [VCE+22]. The Vecs2Pauli algorithm also uses the coset intersection algorithm as well as the algorithm to find a minimal generating set finding algorithm for a group mentioned in this paper in Appendix D.

This paper is organized as follows. To start, some necessary mathematical and quantum basics are explained in Section 2. Section 3 contains the problem definition and explains a naive approach, the Vecs2Pauli algorithm and its subalgorithms. Possible applications for the Vecs2Pauli algorithm are more elaborately explained in Section 4, while Section 5 discusses some challenges related to this algorithm.

2 Background

2.1 Relevant math

2.1.1 Relevant linear algebra

Some background mathematics is required before we start introducing quantum computing mechanics. The *tensor product* of two matrices A and B is denoted by $A \otimes B$ and is computed by essentially replacing each entry a_{ij} of A by $a_{ij} \cdot B$. If A is a $m \times n$ matrix and B is of size $p \times q$, the resulting matrix $A \otimes B$ will have size $m \cdot p \times n \cdot q$. Let's illustrate this with an example.

Example 2.1. *In this example, A has size 3×1, and B is of size 1×2. The resulting matrix has*

size 3×2.

$$A = \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 5 \end{pmatrix}$$

$$A \otimes B = \begin{pmatrix} 3 \cdot B \\ 1 \cdot B \\ 0 \cdot B \end{pmatrix} = \begin{pmatrix} 6 & 15 \\ 2 & 5 \\ 0 & 0 \end{pmatrix}.$$

Useful properties of the tensor product that will be used later on are the following:

$$c(A \otimes B) = (cA) \otimes B = A \otimes (cB) \quad \forall c \in \mathbb{C} \tag{1}$$

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD) \tag{2}$$

2.1.2 Relevant group theory

The algorithm discussed in this paper makes use of algebraic groups and cosets. A group is a combination of a set of values and an operation, such that by applying this operation to any tuple of members of the set, the resulting value will also be a member of the set: the set is closed under the group operation. This operation should satisfy associativity; parentheses can be rearranged without changing the result. For example: $a + (b + c) = (a + b) + c$, so the regular $+$ operation is associative. Furthermore, the group always contains an identity element, as well as an inverse for every member of the group. An identity element is an element that, when applying the group operation on the identity and a second element, the result will be that second element (so applying the identity doesn't change anything).

Example 2.2. *Let's take the group* $G = \{1, -1, i, -i\}$ *with multiplication as its operation, which is an associative operation. By multiplying any of these elements, the result will also be in this group.* 1 *is the identity for multiplying complex numbers, since multiplying a complex number with* 1 *will not change the complex number. We see the identity is included in this group. Furthermore, every element has an inverse: another element such that the result is* 1 *if they are multiplied together:* $-1^{-1} = -1$, $i^{-1} = -i$, $-i^{-1} = i$. *All group axioms are hereby satisfied, so group* G *is a valid group.*

A coset is a specific type of subset: it is like a 'shifted group'. It is obtained by taking a group and one other element of the same type as the members of the group. This single element is called the *coset representative*. The members of the coset are then acquired by applying the operation with the coset representative as one operator and all elements of the group as the second operator. In this paper we use the word *coset* for a left coset: the single element operator is to the left of the operation, while the group elements are to the right. This is relevant for non-commutative operations, such as matrix multiplication. Mathematically, a left coset with multiplication as its operation is denoted as follows: $\{\pi \cdot g \mid g \in G\}$ where π is the coset representative and G is the group.

Example 2.3. *Let's take the same group* G *as in Example 2.2. Our coset representative can be* 5. *The members of the (left) coset will then be* $\{5 \cdot 1, 5 \cdot -1, 5 \cdot i, 5 \cdot -i\} = \{5, -5, 5i, -5i\}$. *Note that a coset doesn't necessarily have all the properties that a group has. For example, it doesn't contain the identity element (which is* 1 *in this example), nor does every element have an inverse.*

2.2 Quantum states

2.2.1 Superposition

In quantum computing, quantum bits or qubits are used. The special thing about qubits is that they are not necessarily 0 or 1. They can also be in a state called *superposition*. This can be seen as being in both states at once, with certain amplitudes for each possible state. For single qubits, it's as follows: the classical 0 is denoted as $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, while the classical 1 is written as $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

A single qubit in superposition can take on any value of the form

$$|\phi\rangle = \alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle ,$$

where $\alpha_0, \alpha_1 \in \mathbb{C}$ and $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Often a quantum state is written as a vector of length 2^n, with n the number of qubits considered:

$$|\phi\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{2^n-1} \end{pmatrix} .$$

This vector is also called a 'ket'.

While a qubit can be in superposition while computations are being performed, it can only be in one of the classical states when checking the value during a quantum circuit. In other words, when checking the value of the qubit, it will always be exactly $|0\rangle$ or $|1\rangle$. The probability of seeing a certain state is its amplitude squared: $|\alpha_i|^2$. So if $|\phi\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{pmatrix}$, the probability of observing classical state $|0\rangle$ is just as big as state $|1\rangle$, with both having a probability of 0.5.

2.2.2 Multiple qubits

Before we generalize to many qubits, let's first look at two. With two qubits, there are four basic computational states. The first qubit is either $|0\rangle$ or $|1\rangle$, and the second qubit is also either $|0\rangle$ or $|1\rangle$, which makes a total of four possible combinations. The combined state is $|\phi\rangle \otimes |\psi\rangle$, where $|\phi\rangle$ is the state of the first qubit, and $|\psi\rangle$ is the state of the second qubit. This is why we needed to know about the tensor product before generalizing to multiple qubits. For multiple qubits where each qubit is either $|0\rangle$ or $|1\rangle$, such as $|1\rangle \otimes |0\rangle \otimes |1\rangle$, we often shorten this notation to $|101\rangle$.

In general, when generalizing to n qubits, there are 2^n possible basic computational states. For a superposition of n states, we therefore need 2^n amplitudes, which can be written as a vector of length 2^n. This is a big reason why naively doing classical simulation is challenging: it takes a lot of memory space to perform computations with simulated qubits.

2.3 Stabilizer states

2.3.1 Pauli matrices

Performing computations on qubits can be done by essentially multiplying the amplitude vector with a unitary operator. A unitary operator is a square matrix U such that $U^\dagger U = UU^\dagger = I$ where the $\dagger$ denotes the adjoint operator: take the transpose of a matrix and then replace every entry in this transposed matrix by its conjugate transpose. The conjugate transpose of a complex number $a + b \cdot i$ is $a - b \cdot i$ for $a, b \in \mathbb{R}$.
The most well-known quantum operators (or *gates*) work on either 1 or 2 qubits. An example is the bit-flip operator: $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.
There are certain gates that are used quite often in quantum physics. These are:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

These four matrices are called the Pauli matrices. They have some rules: $X^2 = Y^2 = Z^2 = I$, and

$$XY = iZ, \quad YZ = iX, \quad ZX = iY,$$

$$YX = -iZ, \quad ZY = -iX, \quad XZ = -iY.$$

The Pauli matrices form a group of 16 elements: each of the four matrices appears four times: multiplied by $1, -1, i$, and $-i$. A group always contains the identity element, which is I for matrices when the operation is multiplication, and when multiplying elements of a group, one will always get another element of the group as an outcome. This is also shown by the rules above. Furthermore, the operation must be associative, which multiplication is when regarding matrices: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$. Finally, every element must also have its inverse in the group. This is easily shown, since $X^2 = Y^2 = Z^2 = I$, each of I, X, Y, Z is its own inverse. This also works when talking about $-I, -X, -Y, -Z$. If the factor is i, the inverse is $-i \cdot$ *the same matrix* and vice versa, since $i \cdot -i = -i^2 = - - 1 = 1$.

These Pauli matrices operate on single qubits. To let them operate on multiple qubits, we can take the tensor product of multiple single-qubit Pauli matrices, creating a so-called Pauli string. An example of a Pauli string is $X \otimes Y \otimes Z$, which is often shortened to XYZ. A Pauli string can be multiplied by ± 1 or $\pm i$ to obtain a member of the n-qubit Pauli group. Computing the product of a member of the n-qubit Pauli group and an n-qubit state might seem challenging due to the size of the matrix and the vector, but using Equation 2, it becomes rather simple:

$$-iXYZ \cdot |101\rangle = -i((X \otimes Y \otimes Z) \cdot (|1\rangle \otimes |0\rangle \otimes |1\rangle)) = -i \cdot (X \cdot |1\rangle) \otimes (Y \cdot |0\rangle) \otimes (Z \cdot |1\rangle).$$

2.3.2 Stabilizer groups

Stabilizer groups are groups containing elements, that when multiplied with a vector describing a quantum state, will produce that same vector (*stabilizing* it) [Got97, Got98a]. Each quantum state has a stabilizer group, which contains at least the identity operator. A stabilizer group has at most 2^n elements for an n-qubit state, and if it has exactly 2^n elements, that group uniquely describes

the corresponding quantum state. If a stabilizer group has less than 2^n elements, multiple quantum states may have that group as their stabilizer group. Stabilizer groups are abelian groups.
The stabilizer group for $|0\rangle = \{I, Z\}$ and for $\frac{1}{\sqrt{2}}(|000\rangle + i\,|111\rangle$ the stabilizer group is given by $\{III, YXX,\ XYX, XXY, ZZI, IZZ, ZIZ, -YYY\}$. These stabilizer groups have the same properties as the Pauli group; they contain the identity element, and when multiplying two elements, a third element of the group will be the outcome. This second property makes it possible to reduce the number of elements necessary to note. A group of size $\leq 2^n$ can be *generated* by $\leq n$ elements that are independent. We can therefore save these at most n elements, greatly reducing the necessary memory. The other elements of the stabilizer group can then be found by multiplying the elements in the *generating set*.

2.4 Clifford gates

Many quantum circuits use Clifford gates. All Clifford gates can be generated with the following gates:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, \quad S = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}, \quad CNOT = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}.$$

Clifford gates are gates that normalize the Pauli group. This means that when one multiplies a tensor product of Pauli matrices with a Clifford gate, the outcome will still be a member of the Pauli group.

This is particularly useful when simulating quantum circuits since it also means that if a Clifford gate is applied to a quantum state with a full stabilizer group, the resulting state will also have a full stabilizer group [Got98b].

2.5 Use of stabilizer states

Stabilizer states are used in many different fields within quantum computing. This is because the subset formed by stabilizer states is large enough to use it in many applications without needing to extend this subset with non-stabilizer states. They are used a lot in quantum communication [FWH+10] and they are fundamental in quantum error correction [Got97]. As discussed before, especially when only using Clifford gates, stabilizer states are also used frequently in classical simulations.

3 Algorithm

3.1 Problem Definition

The goal we are trying to achieve is to find all PauliLIMs that will map input vector v to input vector w, where v and w can be amplitude vectors of quantum states, though they don't necessarily have to be. The input vector can contain any values $x \in \mathbb{C}$. A *PauliLIM* is a combination of a factor (in the rest of this paper referred to as α) which can be any complex number, and a *Pauli string*, which is the tensor product of Pauli matrices. In mathematical notation, we are looking for

all combinations of $\alpha \in \mathbb{C}$ and $P = Q_1 \otimes Q_2 \otimes \ldots \otimes Q_n$, where n is the amount of qubits the vector represents ($n = log_2(length(v))$) and $Q_i \in \{I, X, Y, Z\}$ such that

$$\alpha \cdot P \cdot v = w, \tag{3}$$

Lemma 3.1. *All solutions to Equation 3 can be represented as a coset. We can use a complex number α and a Pauli operator P, such that $\alpha \cdot P$ maps v to w. $\alpha \cdot P$ will be used as coset representative, and the stabilizer group of v is the group of the coset.*

Proof. Lemma 16 in the paper by Vinkhuizen et al. [VCE+22] shows that the set of solutions to a mapping from quantum state $|v\rangle$ to quantum state $|w\rangle$ is of the form $\pi \cdot Stab(|v\rangle)$ with π a mapping from $|v\rangle$ to $|w\rangle$ and $Stab(|v\rangle)$ the stabilizer group for $|v\rangle$. This is a generalization of that Lemma. Since the proof of Lemma 16 does not use any properties that only apply to quantum states, but in reality uses only linear algebra and group theory, the proof of Lemma 16 still holds. □

A way to solve this is by simply checking all possible solutions one by one. If we first assume $\alpha = 1$ for simplicity, there are 4^n possible solutions we have to check. If $\alpha \neq 1$, we can start by still multiplying any possible P with v. If $P \cdot v$ and w don't match, we can find a possible α by dividing w's first non-zero element by $P \cdot v$'s value on the same index. This will not work if $w = \mathbf{0}$ (which can easily be checked before even starting the algorithm) or $P \cdot v$'s value is 0, but in that case, no α would work anyway. After finding this possible α, simply multiply $P \cdot v$ by α and check the result against w.

This algorithm can be improved upon by using one of the characteristics of a PauliLIM. In the matrix of a PauliLIM, there will always be exactly one non-zero value in each row and column. As such, applying a PauliLIM can also be seen as applying a permutation, since every entry in v has a corresponding entry in the result vector it is directed to, possibly multiplied by $-1, i$ or $-i$. By applying the PauliLIM as a permutation of v instead of using matrix multiplication, we need fewer operations.

Example 3.2. *Take PauliLIM $P = iXIZ$ and $v = (1, 0, 2, 1, 0, 0, 0, 1)^T$. v can also be written as $1 \cdot |000\rangle + 2 \cdot |010\rangle + 1 \cdot |011\rangle + 1 \cdot |111\rangle$. We apply the PauliLIM as follows using Equation 2:*

$$iXIZ \cdot (1 \cdot |000\rangle + 2 \cdot |010\rangle + 1 \cdot |011\rangle + 1 \cdot |111\rangle) = i \cdot (X \cdot |0\rangle \otimes I \cdot |0\rangle \otimes Z \cdot |0\rangle) + i \cdot (X \cdot |0\rangle \otimes I \cdot |1\rangle \otimes Z \cdot |0\rangle)$$

$$+ i \cdot (X \cdot |0\rangle \otimes I \cdot |1\rangle \otimes Z \cdot |1\rangle) + i \cdot (X \cdot |1\rangle \otimes I \cdot |1\rangle \otimes Z \cdot |1\rangle) = i \cdot |100\rangle + 2i \cdot |110\rangle - i \cdot |111\rangle - i \cdot |011\rangle$$

which can be written as $w = (0, 0, 0, -i, i, 0, 2i, -i)^T$.

In this paper, we provide an alternative approach which will be explained in the following section.

3.2 Vecs2Pauli

Below, we will provide an algorithm that will return the generating set for the Pauli strings that satisfy Equation 3. These Pauli strings will be found using a recursive approach, which is inspired by the approach in [VCE+22]. While the algorithm was intended for vectors representing quantum states, we have extended it to work for any vector with a length of 2^n and entries in $\mathbb{C}$. For simplicity, we will assume $\alpha = 1$ for now. We will address later how it is handled if $\alpha \neq 1$.

3.2.1 Base case

We first take a look at the base case for the Vecs2Pauli algorithm, where v and w concern 1 qubit and the corresponding vectors are of length 2. In this case we are searching for single-qubit Pauli operators P and factors $\alpha \in \mathbb{C}$ such that

$$\alpha \cdot P \cdot v = w, \tag{4}$$

with $v = \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}$ and $w = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}$ We know the solution will be the coset consisting of one solution to Equation 4 and the stabilizer group of $|v\rangle$. Finding this stabilizer group will be discussed later in Section 3.3, so for now we will focus on finding one combination of α and P that satisfy Equation 4. There are six distinct possibilities for combinations of α and P here which will all take different routes through the algorithm:

- There are no solutions (e.g. $\alpha \cdot P \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$).
- There are multiple possible α's (e.g. $\alpha \cdot P \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$).
- One possible α combined with one of I, X, Y, Z.

The algorithm for this base case is written in Algorithm 3. In lines 1-2 the third possibility is checked. If this is not the case, the first step is to see if there are any solutions where $P \in \{I, Z\}$. We need to satisfy the following equations:

$$\begin{cases} \alpha \cdot v_0 = w_0, \\ \alpha \cdot v_1 = w_1, \end{cases} \quad \text{if P = I} \tag{5}$$

$$\begin{cases} \alpha \cdot v_0 = w_0, \\ -\alpha \cdot v_1 = w_1, \end{cases} \quad \text{if P = Z} \tag{6}$$

If either $v_0 = 0$ and $w_0 \neq 0$, or $v_1 = 0$ and $w_1 \neq 0$, there will be no such solution, so we can skip lines 6-18. If v_0 and w_0 are both equal to 0, we only have v_1 and w_1 to determine a solution. For convention, if $\frac{w_1}{v_1} < 0$, we say $P = Z$ and return it with $\alpha = -\frac{w_1}{v_1}$. Otherwise we simply return $\alpha = \frac{w_1}{v_1}$ and $P = I$. If it is not the case that $v_0 = w_0 = 0$, we move on to lines 13-18. We know then that $v_0 \neq 0$, so we can calculate $\frac{w_0}{v_0}$. Here we assume that the top equations of Equations 5 and 6 hold, and check the corresponding α with the second equations in lines 15-18.

If no solutions with $P \in \{I, Z\}$ have been found, solutions with $P \in \{X, Y\}$ are considered in lines 20-34. The corresponding equations to be satisfied are as follows:

$$\begin{cases} \alpha \cdot v_0 = w_1, \\ \alpha \cdot v_1 = w_0, \end{cases} \quad \text{if P = X} \tag{7}$$

$$\begin{cases} i \cdot \alpha \cdot v_0 = w_1, \\ -i \cdot \alpha \cdot v_1 = w_0, \end{cases} \quad \text{if P = Y} \tag{8}$$

In lines 20-21 we again check whether there will be any solutions of this type. If this is not the case, there are no solutions at all, which will be returned in line 36. If there are solutions, the first step is to check whether $v_0 = w_1 = 0$. If so, we determine α only by the second equations of Equations 7 and 8. We choose to say $P = X$ if $\frac{v_1}{w_0} \in \mathbb{R}$, and $P = Y$ if the imaginary part of $\frac{v_1}{w_0} \neq 0$. Note that if $P = Y$, we need to modify the returned value for α in line 26, since we want α s.t. $-i \cdot \alpha \cdot v_1 = w_0$, while we calculated $\alpha = \frac{w_0}{v_1}$.
In case v_0 and w_1 are not both 0, we know $v_0 \neq 0$, so we can calculate $\alpha = \frac{w_1}{v_0}$. We then check the second equation of Equations 7 and 8 and return α and either X or Y. Note that the calculated α is different from the α in Equation 8, namely: calculated $\alpha = i \cdot \alpha$. This is the reason lines 33-34 don't correspond one-on-one with Equation 8.

3.2.2 Multi-qubit case

We provide a recursive algorithm for dealing with multiple qubits. We can rewrite Equation 3 using vector notation:

$$\alpha \cdot P_1 \otimes \ldots \otimes P_n \cdot \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}, \tag{9}$$

where v_0, v_1, w_0, w_1 are vectors themselves of half the length of v and w. Depending on P_1, there are 4 options for rewriting this equation.

$$\begin{pmatrix} \alpha \cdot P_{2\ldots n} & 0 \\ 0 & \alpha \cdot P_{2\ldots n} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \rightarrow \begin{cases} \alpha \cdot P_{2\ldots n} \cdot v_0 = w_0, \\ \alpha \cdot P_{2\ldots n} \cdot v_1 = w_1, \end{cases} \text{ if } \mathrm{P}_1 = I \tag{10}$$

$$\begin{pmatrix} 0 & \alpha \cdot P_{2\ldots n} \\ \alpha \cdot P_{2\ldots n} & 0 \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \rightarrow \begin{cases} \alpha \cdot P_{2\ldots n} \cdot v_1 = w_0, \\ \alpha \cdot P_{2\ldots n} \cdot v_0 = w_1, \end{cases} \text{ if } \mathrm{P}_1 = X \tag{11}$$

$$\begin{pmatrix} 0 & -i \cdot \alpha \cdot P_{2\ldots n} \\ i \cdot \alpha \cdot P_{2\ldots n} & 0 \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \rightarrow \begin{cases} -i \cdot \alpha \cdot P_{2\ldots n} \cdot v_1 = w_0, \\ i \cdot \alpha \cdot P_{2\ldots n} \cdot v_0 = w_1, \end{cases} \text{ if } \mathrm{P}_1 = Y \tag{12}$$

$$\begin{pmatrix} \alpha \cdot P_{2\ldots n} & 0 \\ 0 & -\alpha \cdot P_{2\ldots n} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \rightarrow \begin{cases} \alpha \cdot P_{2\ldots n} \cdot v_0 = w_0, \\ -\alpha \cdot P_{2\ldots n} \cdot v_1 = w_1, \end{cases} \text{ if } \mathrm{P}_1 = Z. \tag{13}$$

Each subequation corresponds to a call to Vecs2Pauli concerning one fewer qubit than in the original call. Continuing through the recursive tree, eventually a call will be performed with vectors of length two, which will be redirected to Vecs2PauliBase, as done in lines 7-10 of Algorithm 4.

Before going immediately to the recursive part of the algorithm, we check some special cases first, specifically where one or more parts of the vectors are filled with zeroes. If the entire vector v is filled with zeroes, we check two special cases: either w is also filled with zeroes, and every PauliLIM is a solution to Equation 9. Otherwise, w contains non-zero elements and no PauliLIM will be a solution. If only w is a zero vector, a special value is returned: $\alpha = 0$, since $0 \cdot any\ PauliLIM$ is a solution in that case. These cases are checked and handled in lines 1-6.

In lines 11-15 some prerequisites are handled: creating the subvectors of v and w, as well as finding the stabilizer group of v. We need the stabilizer group of v, since all PauliLIMs that

transform v into w form a coset of one such PauliLIM and v's stabilizer group. The algorithm for finding the stabilizer group can be found in Algorithm 2.

The recursive calls explained above are handled next. In lines 16-25 Equation 10 is handled by performing either 1 or 2 recursive calls and subsequently finding the intersection between the answers (using Algorithm 1) if necessary, since both subequations need to be satisfied for a PauliLIM to be a solution. If v_0 is filled with zeroes but w_0 is not, or v_1 is the zero vector and w_1 isn't, it is not necessary to perform these recursive calls, since we will not find a solution anyway. Also, if the first call does not yield a solution, it is not necessary to perform the second call. Then one element of the intersection is returned by the *computeCosetIntersectionElement* (Algorithm 1, and afterward a left-multiplication with the Pauli-I element is done. The coset intersection that is being determined in this algorithm is found using the coset intersection algorithm in [VCE+22]. Finally, factor α is returned along with the coset of the selected element and v's stabilizer group.

If no solution is found, this process is repeated a maximum of 3 times, with Pauli-matrices Z, X, and Y in lines 26-29, 30-39 and 40-49 respectively. If still not solution is found, the special value of *no α possible* is returned to indicate there is no solution.

3.3 Finding the stabilizer group

When finding the stabilizer group, we will do this recursively again, since it is the same question as for the Vecs2Pauli algorithm, where $v = w$. We will divide the explanation of the algorithm in two parts again, the base case and the case for multiple qubits. The algorithm can be found in Algorithm 2.

3.3.1 Base case

We are looking for all combinations of $\alpha \in \mathbb{C}$ and $P \in \{I, X, Y, Z\}$ such that

$$\alpha \cdot P \cdot |v\rangle = |v\rangle \,. \tag{14}$$

Given the characteristics of the Pauli matrices, we know that $\alpha \in \{\pm 1, \pm i\}$ in this case, since the Pauli matrices only perform rotations and reflections in the complex plane. This means applying a Pauli matrix will never change a vectors length, and as such, applying $\alpha \cdot P$ with $\alpha \notin \{\pm 1, \pm i\}$ will always result in a vector with a different length than the original vector. Only in the very special case that $v = \bar{0}$ (the vector filled with 0's), other values for α are possible. Specifically speaking, in that case, all values for $\alpha \in \mathbb{C}$ satisfy Equation 14, as well as any member P of the Pauli group.

If we work out Equation 14 for $P = I$ and $P = Z$, we get the following pairs of equations.

$$\begin{cases} \alpha \cdot v_0 = v_0, \\ \alpha \cdot v_1 = v_1, \end{cases} \quad \text{if P = I} \tag{15}$$

$$\begin{cases} \alpha \cdot v_0 = v_0, \\ -\alpha \cdot v_1 = v_1, \end{cases} \quad \text{if P = Z} \tag{16}$$

Clearly, $1 \cdot I$ will always be a member of the stabilizer group, since both equations in Equation 15 will always hold for $\alpha = 1$. In contrast, $-1 \cdot I, i \cdot I$ and $-i \cdot I$ will only be in the stabilizer group if both v_0 and v_1 are equal to 0. In a similar fashion, we check whether any multiples of Z are in the stabilizer group. $1 \cdot Z$ will be in the stabilizer group if $v_1 = 0$, while $-1 \cdot Z$ will be included if $v_0 = 0$. Again, $i \cdot Z$ and $-i \cdot Z$ will only be members of the stabilizer group in the case that both v_0 and v_1 are 0.

Working out Equation 14 for $P = X$ and $P = Y$ gives us the following pairs of equations.

$$\begin{cases} \alpha \cdot v_0 = v_1, \\ \alpha \cdot v_1 = v_0, \end{cases} \quad \text{if P = X} \tag{17}$$

$$\begin{cases} i \cdot \alpha \cdot v_0 = v_1, \\ -i \cdot \alpha \cdot v_1 = v_0, \end{cases} \quad \text{if P = Y} \tag{18}$$

It is easy to determine from Equation 17, $1 \cdot X$ will be a part of the stabilizer group if both entries in the vector are the same, while $-1 \cdot X$ will be in the stabilizer group if one element is $-1\cdot$ the other element, such as $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. Any other $\alpha \cdot X$ will only be in the stabilizer group if both entries of vector v are 0. Determining whether $\alpha \cdot Y$ is a solution to Equation 18 is slightly less trivial. By working out Equation 18 with the four possibilities for α: $1, -1, i, -i$, we get the following four sets of equations:

$$\begin{cases} i \cdot v_0 = v_1, \\ -i \cdot v_1 = v_0, \end{cases} \quad \text{if P = Y and } \alpha = 1 \tag{19}$$

$$\begin{cases} -i \cdot v_0 = v_1, \\ i \cdot v_1 = v_0, \end{cases} \quad \text{if P = Y and } \alpha = -1 \tag{20}$$

$$\begin{cases} i \cdot i \cdot v_0 = -v_0 = v_1, \\ -i \cdot i \cdot v_1 = v_1 = v_0, \end{cases} \quad \text{if P = Y and } \alpha = i \tag{21}$$

$$\begin{cases} i \cdot -i \cdot v_0 = v_0 = v_1, \\ -i \cdot -i \cdot v_1 = -v_1 = v_0, \end{cases} \quad \text{if P = Y and } \alpha = -i \tag{22}$$

It can be seen that Equations 21 and 22 will only be true in the case that $v_0 = v_1 = 0$, while Equations 19 and 20 can also be true for non-zero vectors.

Since the Vecs2Pauli algorithm will give a special output if the input vector v is the $0-$vector, it is not needed to determine the stabilizer group of the $0-$vector. As such, some of the beforementioned $\alpha \cdot P$ combinations do not need to be checked since these tests will never be true.
In Algorithm 2, the stabilizer group of a vector of length 2 is found in lines 1-16.

3.3.2 Multi-qubit case

Finding the stabilizer group for $|v\rangle$ can be done in a similar fashion to the algorithm described in Section 3.2.

$$P_2 \otimes \ldots \otimes P_n \cdot |v_{2\ldots n}\rangle = |v_{2\ldots n}\rangle \, , P_2 \otimes \ldots \otimes P_n \cdot |v'_{2\ldots n}\rangle = \pm \, |v'_{2\ldots n}\rangle \quad \text{if } P_1 \in \{\mathbb{I}, Z\}, \tag{23}$$

$$P_2 \otimes \ldots \otimes P_n \cdot |v_{2\ldots n}\rangle = y\,|v'_{2\ldots n}\rangle\,, P_2 \otimes \ldots \otimes P_n \cdot |v'_{2\ldots n}\rangle = y^*\,|v_{2\ldots n}\rangle \quad \text{with } y \in \{1, i\} \text{ if } P_1 \in \{X, Y\}. \tag{24}$$

In the equation above, y^* is the complex conjugate of y. It is calculated as follows: if $y = a + b \cdot i$, then $y^* = a - b \cdot i$. Combining these equations to find the stabilizer group of $|v\rangle$ gives the following, where $Stab(v)$ denotes the stabilizer group of v, and $Vecs2Pauli(v, w)$ denotes the coset of solutions that transform v to w.

$$\begin{aligned} Stab(|v\rangle) = \quad & \mathbb{I} \otimes (Stab(|v_{2\ldots n}\rangle) \cap Stab(|v'_{2\ldots n}\rangle)) \\ & \cup Z \otimes (Stab(|v_{2\ldots n}\rangle) \cap -1 \cdot Stab(|v'_{2\ldots n}\rangle)) \\ & \cup X \otimes (Vecs2Pauli(|v_{2\ldots n}\rangle\,, |v'_{2\ldots n}\rangle) \cap Vecs2Pauli(|v'_{2\ldots n}\rangle\,, |v_{2\ldots n}\rangle) \\ & \cup Y \otimes (Vecs2Pauli(|v_{2\ldots n}\rangle\,, i \cdot |v'_{2\ldots n}\rangle) \cap Vecs2Pauli(|v'_{2\ldots n}\rangle\,, -i * |v_{2\ldots n}\rangle). \end{aligned} \tag{25}$$

Using Lemma 3.1, we can rewrite Equation 25 as follows:

$$\begin{aligned} Stab(|v\rangle) = \quad & \mathbb{I} \otimes (Stab(|v_{2\ldots n}\rangle) \cap Stab(|v'_{2\ldots n}\rangle)) \\ & \cup Z \otimes (Stab(|v_{2\ldots n}\rangle) \cap -\mathbb{I} \cdot Stab(|v'_{2\ldots n}\rangle)) \\ & \cup X \otimes (\pi \cdot Stab(|v_{2\ldots n}\rangle) \cap \pi^{-1} \cdot Stab(|v'_{2\ldots n}\rangle)) \\ & \cup Y \otimes (\pi i \cdot Stab(|v_{2\ldots n}\rangle) \cap -\pi^{-1} i \cdot Stab(|v'_{2\ldots n}\rangle)), \end{aligned} \tag{26}$$

where π is a Pauli string transforming $|v_{2\ldots n}\rangle$ into $|v'_{2\ldots n}\rangle$.

The intersection of the returned cosets of two recursive calls can be computed in the same way as in the Vecs2Pauli algorithm in Section 3.2.2. It is also necessary to find the union of these intersections. For general cosets, a union of cosets doesn't necessarily form another coset. However, since the union of the separate parts of Equation 26 once again forms a stabilizer group, this union does form a coset. Specifically, the coset intersection will form a coset with the stabilizer group we intended to find as the group part of the coset, and $I^{\otimes n}$ as coset representative.

Furthermore, it is sufficient to find all stabilizers that are to be multiplied with $\mathbb{I}$, and one of each for those multiplied with X and Z. If either of the calls corresponding to X and Z doesn't give a solution, only then is it necessary to look for an operator σ such that $Y \otimes \sigma$ is part of the stabilizer group, to find its generators.

The reason we can do this is due to the fact that in the algorithm to find a smallest generating set, Gauss elimination is performed on a matrix representing the stabilizers. In Appendix A of [VCE+22] it is explained why this is allowed. During the Gauss elimination, any stabilizers starting with X, Y or Z will be eliminated in the row reduction, except for a maximum of one of each. Since $Y = \frac{ZX}{i}$, if we have both a stabilizer starting with Z and one starting with X, we can also reduce the stabilizer starting with Y, so in that case we don't need any stabilizers starting with Y.

Recursively finding the stabilizer group for multi-qubit vectors is done is lines 17-40 in Algorithm 2.

Algorithm 1 A subroutine for finding an element in a coset intersection and returning it

Procedure computeCosetIntersectionElement(cosets $Coset_1$ and $Coset_2$, complex numbers α_1 and α_2)

Output: Pauli string π and $\alpha \in \mathbb{C}$ or special value if no solution

```
if Coset_1 and Coset_2 are both non-empty then
    π ← one element in Coset_1 ∩ Coset_2
    if there is no such π then
        return no solution
    if α_1 = α_2 or α_2 = allAlphaPossible then
        return α_1 · π
    else if α_1 = allAlphaPossible then
        return α_2 · π
return no solution
```

4 Applications

There are several areas where the Vecs2Pauli algorithm could be used. We will discuss two of the applications that we had in mind when working on this algorithm. There could of course be more uses of this algorithm we have not thought of so far.

4.1 In classical simulations

One such area is classical simulations: the simulation of a quantum computer on a regular computer. As described in Section 2.3, to encode a quantum state of n qubits, we need a vector of length 2^n. However, if the quantum state has a full stabilizer group (generated by n elements), that stabilizer group uniquely describes the corresponding quantum state [NC10]. Storing this vector requires exponentially more space than storing the stabilizers, which is a very significant difference already for relative small n.

This can be used in classical simulations. Since Vecs2Pauli can be used to find a state's stabilizers, it can be quite easily checked whether a state can be described by a full stabilizer group. Many often-used quantum gates preserve the ability to write a quantum state using its stabilizer group, see Section 2.4 on Clifford gates. Therefore classical simulation can be sped up by using stabilizer group description where possible, switching to vectors when needed, and checking whether stabilizer group descriptions can be used again by using Vecs2Pauli. It is important to note, that due to Vecs2Pauli's exponential input size, Vecs2Pauli can only be used in simulations like this for a relatively small number of qubits.

Classical simulations are mostly done in order to determine how noisy a quantum computer could be to still be accurate enough for the problem one's trying to solve. A built quantum computer is tested by running a quantum circuit with a relatively small number of qubits. The levels of noise on this computer are measured and used as a parameter in the simulation, which uses a much larger number of qubits. The level of noise can be adapted and the outcomes are analyzed. This way, the highest noise level still allowed can be determined.

Algorithm 2 Finding the stabilizer group

Procedure FindStabilizerGroup(vector v of length 2^n)

Output: stabilizer group generating set G describing the stabilizer group of v.

if $n = 1$ **then**
  $S \leftarrow \{I\}$ {see Equation 15}
  if $v_1 = 0$ **then**
    $S \leftarrow S \cup \{Z\}$ {see Equation 16}
  if $v_0 = 0$ **then**
    $S \leftarrow S \cup \{-Z\}$ {see Equation 16}
  if $v_0 = v_1$ **then**
    $S \leftarrow S \cup \{X\}$ {see Equation 17}
  if $v_0 = -v_1$ **then**
    $S \leftarrow S \cup \{-X\}$ {see Equation 17}
  if $v_1 = i \cdot v_0$ && $v_0 = -i \cdot v_1$ **then**
    $S \leftarrow S \cup \{Y\}$ {see Equation 19}
  if $v_1 = -i \cdot v_0$ && $v_0 = i \cdot v_1$ **then**
    $S \leftarrow S \cup \{-Y\}$ {see Equation 20}
  $G \leftarrow findSmallestGeneratingSet(S)$
  return G
else
  $x \leftarrow v_{0 \ldots 2^{n-1}-1}$ {the top half of v}
  $y \leftarrow v_{2^{n-1} \ldots 2^n-1}$ {the bottom half of v}
  $Stab_x \leftarrow findStabilizerGroup(x)$
  $Stab_y \leftarrow findStabilizerGroup(y)$

  $S \leftarrow \{I \otimes g \quad | \quad g \in Stab_x \cap Stab_y\}$

  $SolutionZ \leftarrow Vecs2Pauli(y, -y)$
  if $SolutionZ$ not empty **then**
    $S \leftarrow S \cup \{Z \otimes g_0 \quad | \quad g_0 \ldots g_k = Stab_x \cap SolutionZ\}$

  $SolutionX1 \leftarrow Vecs2Pauli(x, y)$
  $SolutionX2 \leftarrow$ (inverse of coset representative of $SolutionX1$, group of $SolutionX1$)
  if $SolutionX1$ and $SolutionX2$ both non-empty **then**
    $S \leftarrow S \cup \{X \otimes g_0 \quad | \quad g_0 \ldots g_k = SolutionX1 \cap SolutionX2\}$

  if $Stab_x \cap SolutionZ$ is empty or $SolutionX1 \cap SolutionX2$ is empty **then**
    $SolutionY1 \leftarrow Vecs2Pauli(x, i \cdot y)$
    $SolutionY2 \leftarrow (-1 \cdot$ inverse of coset representative of $SolutionY1$, group of $SolutionY1)$
    if $SolutionY1$ and $SolutionY2$ both non-empty **then**
      $S \leftarrow S \cup \{Y \otimes g_0 \quad | \quad g_0 \ldots g_k = SolutionY1 \cap SolutionY2\}$
  $G \leftarrow findSmallestGeneratingSet(S)$
  return G

Algorithm 3 Finding a Pauli string converting v to w for 1 qubit

Procedure Vecs2PauliBase(vectors v and w both of length 2)
Output: A Pauli operator π and $\alpha \in \mathbb{C}$ s.t. $\alpha \cdot \pi \cdot v = w$.

1: **if** all entries in v and w are 0 **then**
2: **return special value: all α possible**
3:
4: **if** ($v_0 = 0$ and $w_0 \neq 0$) or ($v_1 = 0$ and $w_1 \neq 0$) **then**
5: **skip to line 20**
6: **else if** v_0 and w_0 are 0 **then**
7: **if** $v_1 \neq 0$ **then**
8: $\alpha \leftarrow \frac{w_1}{v_1}$
9: **if** $\alpha < 0$ **then**
10: **return** $-\alpha \cdot Z$
11: **else**
12: **return** $\alpha \cdot I$
13: **else**
14: $\alpha \leftarrow \frac{w_0}{v_0}$
15: **if** $\alpha \cdot v_1 = w_1$ **then**
16: **return** $\alpha \cdot I$
17: **if** $-\alpha \cdot v_1 = w_1$ **then**
18: **return** $\alpha \cdot Z$
19:
20: **if** ($v_0 = 0$ and $w_1 \neq 0$) or ($v_1 = 0$ and $w_0 \neq 0$) **then**
21: **skip to line 36**
22: **else if** $v_0 = 0$ and $w_1 = 0$ **then**
23: **if** $v_1 \neq 0$ **then**
24: $\alpha \leftarrow \frac{w_0}{v_1}$
25: **if** $imag(\alpha) \neq 0$ **then**
26: **return** $-\frac{\alpha}{i} \cdot Y$
27: **else**
28: **return** $\alpha \cdot X$
29: **else**
30: $\alpha \leftarrow \frac{w_1}{v_0}$
31: **if** $\alpha \cdot v_1 = w_0$ **then**
32: **return** $\alpha \cdot X$
33: **if** $-\alpha \cdot v_1 = w_0$ **then**
34: **return** $\frac{\alpha}{i} \cdot Y$
35:
36: **return special value: no α possible**

Algorithm 4 Finding all Pauli strings converting v to w for any number of qubits

Procedure Vecs2Pauli(vectors v and w both of length 2^n)

Output: Pauli string π, $\alpha \in \mathbb{C}$ and stabilizer group generating set G s.t. $\alpha \cdot \pi \cdot g \cdot v = w$ for $g \in \langle G \rangle$.

if v and w are both zero vectors **then**
  return special value: all α **possible**
else if only v is a zero vector **then**
  return special value: no α **possible**
else if only w is a zero vector **then**
  return special value: $\alpha = 0$**,** π **and** G **are irrelevant**
if $n = 1$ **then**
  $\alpha, \pi \leftarrow Vecs2PauliBase(v, w)$
  $Stab \leftarrow findStabilizerGroup(v)$
  return $(\alpha, (\pi, Stab))$
$x \leftarrow$ top half of v
$y \leftarrow$ bottom half of v
$p \leftarrow$ top half of w
$q \leftarrow$ bottom half of w
$Stab \leftarrow findStabilizerGroup(v)$
if x and p are zero vectors **then**
  $\alpha_2, MapI_2 \leftarrow Vecs2Pauli(y, q)$
  return $(\alpha_2, (I\otimes$ coset representative of $MapI_2, Stab))$
$\alpha_1, MapI_1Z_1 \leftarrow Vecs2Pauli(x, p)$
if y and q are zero vectors **then**
  return $(\alpha_1, (I\otimes$ coset representative of $MapI_1Z_1, Stab))$
$\alpha_2, MapI_2 \leftarrow Vecs2Pauli(y, q)$
$\alpha \cdot \pi \leftarrow computeCosetIntersectionElement(MapI_1Z_1, MapI_2, \alpha_1, \alpha_2)$
if $\alpha \cdot \pi$ is not *no solution* **then**
  return $(\alpha, (I \otimes \pi, Stab))$ {solution to Equation 10}
$\alpha_2, MapZ_2 \leftarrow Vecs2Pauli(-y, q)$
$\alpha \cdot \pi \leftarrow computeCosetIntersectionElement(MapI_1Z_1, MapZ_2, \alpha_1, \alpha_2)$
if $\alpha \cdot \pi$ is not *no solution* **then**
  return $(\alpha, (Z \otimes \pi, Stab))$ {solution to Equation 13}
if x and q are zero vectors **then**
  $\alpha_2, MapX_2 \leftarrow Vecs2Pauli(y, p)$
  return $(\alpha_2, (X\otimes$ coset representative of $MapX_2, Stab))$
$\alpha_1, MapX_1 \leftarrow Vecs2Pauli(x, q)$
if y and p are zero vectors **then**
  return $(\alpha_1, (X\otimes$ coset representative of $MapX_1, Stab))$
$\alpha_2, MapX_2 \leftarrow Vecs2Pauli(y, p)$
$\alpha \cdot \pi \leftarrow computeCosetIntersectionElement(MapX_1, MapX_2, \alpha_1, \alpha_2)$
if $\alpha \cdot \pi$ is not *no solution* **then**
  return $(\alpha, (X \otimes \pi, Stab))$ {solution to Equation 11}

if x and q are zero vectors **then**
    $\alpha_2, MapY_2 \leftarrow Vecs2Pauli(-i \cdot y, p)$
    return $(\alpha_2, (Y \otimes$ coset representative of $MapY_2, Stab))$
$\alpha_1, MapY_1 \leftarrow Vecs2Pauli(i \cdot x, q)$
if y and p are zero vectors **then**
    return $(\alpha_1, (Y \otimes$ coset representative of $MapY_1, Stab))$
$\alpha_2, MapY_2 \leftarrow Vecs2Pauli(-i \cdot y, p)$
$\alpha \cdot \pi \leftarrow computeCosetIntersectionElement(MapY_1, MapY_2, \alpha_1, \alpha_2)$
if $\alpha \cdot \pi$ is not *no solution* **then**
    return $(\alpha, (Y \otimes \pi, Stab))$ {solution to Equation 12}
return special value: no α **possible**

In one particular simulator, NetSquid [CKD+21], switching from a ket to its corresponding stabilizer group was still an open question. The simulator does have the possibility to go from a stabilizer group back to vector/ket notation bu doing it the other way around was still a challenge. Vecs2Pauli can be used to do exactly that, since it will calculate a vector v's stabilizer group if v is used for both input vectors.

4.1.1 Example

To illustrate how using Vecs2Pauli in simulations would be useful, we will work out an example where it is possible to describe many intermediary states using the stabilizer group's generators, but it is necessary to switch between that and vectors at some point.

Quantum circuits typically start with all qubits in the $|0\rangle$ state. With three qubits, that makes $|000\rangle$ which has $\{ZII, IZI, IIZ\}$ as stabilizer group generators. Since the set of generators has the same amount of elements as there are qubits, this stabilizer group is full and these generators uniquely describe this quantum state.

After applying the H, S and $CNOT$ gates in the example shown in Figure 1, the value of the quantum state is $\frac{1}{\sqrt{2}}(|000\rangle + |110\rangle)$, which can be described by these stabilizer group generators: $\{IIZ, ZZI, XXI\}$. We then run into a problem when applying the $T = \sqrt{S}$ gate in the next step. The T gate is not a Clifford gate and does not map our full stabilizer group to another full stabilizer group. After applying the T gate, the value of the state is $\frac{1}{\sqrt{2}}(|000\rangle + e^{\frac{i\pi}{4}}|110\rangle)$. While IIZ and ZZI are still generators of this state's stabilizer group, there are no others, which means this state doesn't have a full stabilizer group. It is therefore not possible to describe this state using only the stabilizer group generators and it is necessary to switch to an amplitude vector to represent it. The next $CNOT$ will preserve the number of stabilizers, but will

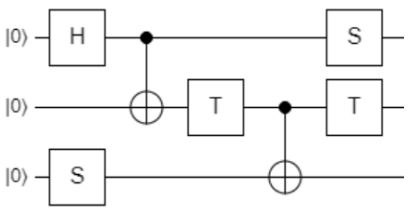

Figure 1: The circuit that is used in the example.

not change the fact that there aren't enough, so a vector is still necessary.

Only after the last gates can we go back to the stabilizer group generator notation, since we encounter another T gate. Since we apply the T gate on the same qubit as before, we have applied T to that qubit twice, making it equivalent to applying the S gate once. Since S is a Clifford gate, this should imply that it should be possible to return to stabilizer group generator notation. This is indeed the case since the quantum state at the end of the circuit can be written as $\frac{1}{\sqrt{2}}(|000\rangle - |111\rangle)$. The stabilizer group of this state is generated by $\{IZZ, ZZI, -XXX\}$ which is a set of size 3, making it once again a full stabilizer group.

Where Vecs2Pauli comes in is checking whether a state has a full stabilizer group and can thus be represented by it. When checking the stabilizer group of $\frac{1}{\sqrt{2}}(|000\rangle + e^{\frac{i\pi}{4}}|110\rangle)$, Vecs2Pauli will only return 2 generators, but for $\frac{1}{\sqrt{2}}(|000\rangle - |111\rangle)$ the algorithm will give 3 generators, so one knows it has a full stabilizer group.

4.2 Educative tool/hypothesis testing

The Vecs2Pauli algorithm could also be used as an educative tool or for hypothesis testing. In both cases, a logical ways to use it would be to either fill in both vectors v and w and see what the outcome is (and if it matches what the user thought it would be), or fill in one vector and trying to find the second vector such that the outcome matches a pre-determined outcome.

Researchers can use Vecs2Pauli to check ideas they have for hypotheses by testing them for example on different inputs with different numbers of qubits. Since manually figuring out which operators map an 8-qubit state to another 8-qubit state becomes quite tedious, using Vecs2Pauli will make it easier to test hypotheses on a slightly higher number of qubits. An example of such a hypothesis would be that applying a certain subgroup of PauliLIMs to a quantum state will always result in a specific subgroup of quantum states.
It is also useful to know if a quantum state is a stabilizer state, since many different fields work with stabilizer states and Clifford gates. As mentioned before in Section 2.5, stabilizer states have many uses.

Vecs2Pauli could also be used in education, for example when teaching students about stabilizer groups. The students could check their answers by filling in the same vector for both input vectors, which will give its stabilizer group (in coset notation, but the coset representative will be the identity).

5 Discussion

5.1 Time complexity

5.1.1 Naive algorithm

In Section 3.1 we provided a naive algorithm for finding all solutions to Equation 3. For a vector containing values for n qubits, there are 4^n possible Pauli operators. To check whether an operator

P is correct ($P \cdot v = w$) we will have to multiply P with v, which would usually take $m^2 \cdot 2^n$ amount of multiplications for an arbitrary matrix and a vector of length 2^n, where m is the matrix size. Since the matrix size is 2^n by 2^n, multiplying P with v takes $2^n \cdot 2^n = 4^n$ multiplications. Checking whether $P \cdot v$ and w match costs 2^n checks. All in all, performing the naive version of the naive algorithm makes for a time complexity of $O(4^n \cdot 4^n) = O(16^n)$.

If we use permutations to apply PauliLIMs, we can reduce this complexity. To apply the PauliLIM as a permutation to vector v of length 2^n, we need $O(2^n \cdot n)$ operations: applying n single-qubit Pauli operators on each entry in v, possibly multiplying the result by $-1, i$ or $-i$. This reduces total complexity to $O(4^n \cdot 2^n \cdot n) = O(8^n \cdot n)$.

Adding in the operations necessary for dealing with $\alpha \neq 1$ costs at most $3 \cdot 2^n$ extra operations, so this will not change our big-O complexity.

5.1.2 Vecs2Pauli

While the Vecs2Pauli algorithm described in Algorithm 4 works completely as intended, it performs many recursive calls. In the worst case, it performs 7 calls to Vecs2Pauli with one qubit less, and in the findStabilizerGroup call, another 7 recursive calls are done. All in all, the complexity of Vecs2Pauli as described in Algorithm 4 is of order $O(14^n \cdot n^3)$. This is significantly slower than the naive algorithm we discussed. However, with some changes to the algorithm, we believe it can be done in $O(7^n \cdot n^3)$.

To explain how this speed-up can be achieved, it is useful to look at all the recursive calls that are performed. If we have vectors $\bar{v} = \begin{pmatrix} a \\ b \end{pmatrix}$ and $\bar{w} = \begin{pmatrix} c \\ d \end{pmatrix}$, the following 7 calls are done:

$$a \to c, \quad b \to d, \quad -b \to d, \quad a \to d, \quad b \to c, \quad i \cdot a \to d, \quad -i \cdot b \to c.$$

Of these 7 calls, the outcomes of the third, sixth, and seventh can be found without performing a recursive call. Take the third one for example, where we want to find α and P such that

$$\alpha \cdot P \cdot -c = d. \tag{27}$$

From the second call, we get an α' and P' such that

$$\alpha' \cdot P' \cdot c = d. \tag{28}$$

To find a solution to Equation 27, we only need to take α' and multiply by -1. Similar reasoning can be used to explain how we can get solutions to the sixth and seventh recursive calls by adapting the solution found in the fourth and fifth calls.

In a similar fashion, the number of recursive calls in the findStabilizerGroup algorithm (Algorithm 2) can be reduced. In one level of findStabilizerGroup performed on vector $\bar{v} = \begin{pmatrix} a \\ b \end{pmatrix}$, the following calls are performed:

$$a \to a, \quad b \to b, \quad -b \to b, \quad a \to b, \quad b \to a, \quad i \cdot a \to b, \quad -i \cdot b \to a.$$

Using the same method as explained before, it will suffice to only perform the following three calls, since the outcomes of the other calls can be derived from these:

$$a \to a, \quad b \to b, \quad a \to b.$$

Note we need one less call here compared to Vecs2Pauli, since $b \to a = (a \to b)^{-1}$.

By implementing the improvements mentioned above, we reduce the number of recursive calls per layer by 7. The complexity of Vecs2Pauli should therefore reduce to $O(7^n \cdot n^3)$.

5.2 Future work

Improving Vecs2Pauli's runtime is a clear way to improve the algorithm, but there are also some other issues that could be improved upon in future work.

5.2.1 Preprocessing the inputs

One of the issues we face at the moment is the fact that computers only have finite memory, while some numbers that are used frequently in quantum states are irrational. One such example is $\frac{1}{\sqrt{2}}$. It is therefore not possible to fully store the numbers, so the decimals are cut off at some point. It is known, that this can sometimes cause discrepancies. In an exaggeration, $\frac{1}{\sqrt{2}}$ can be stored as 0.707106781 at one point and 0.707106782 somewhere else. While they represent the same number, the computer will not recognize them as being equal, which presents difficulties in our algorithms.

Furthermore, if a quantum state has a unique stabilizer group describing it, the values in its corresponding vector will all be of the form $\frac{\{\pm 1, \pm i\} \cdot m}{\sqrt{2^k}}$ with $k \in \{0, \ldots, n\}$, where n is the number of qubits and k is the same for each entry.

To simplify dealing with both of the notions mentioned above, we can use an algorithm to preprocess the input vectors. We will try to simplify all entries to the form $\frac{a+b \cdot i}{\sqrt{r^k}}$ with $k \in \{0, \ldots, n\}$ and $r \in \{1, 2, 3, 5\}$ with n still the number of qubits. If this works, we pull out the common factor and can perform Vecs2Pauli with a vector containing only integers, dealing with the factor after running the algorithm. Preprocessing the vectors like this can be done using Algorithm 5.

Simplifying the input vectors using this has several purposes. Its main purpose is to correct very small errors in numbers. For example, $\begin{pmatrix} 1.0000001 \\ 0.9999998 \end{pmatrix}$ would be recognized as *almost* $\frac{1}{\sqrt{1}} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and the algorithm will work with $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ which saves storage space. Also, it would recognize that Z will send $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ to $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$, while the algorithm won't give Z as a solution when looking for transformations from $\begin{pmatrix} 1.0000001 \\ 0.9999998 \end{pmatrix}$ to $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

We start by going through all entries of the input vector v until a non-zero value is found. We then loop over all k from 0 until n and all $r \in \{1, 2, 3, 5\}$ in lines 2-6 and try to find out if the

value can be written as $\frac{m}{\sqrt{r}^k}$ with m an integer (which may or may not be multiplied by i). If so, it is tested whether the other values in the vector can also be written in that form with the same value for k. This check is performed in lines 10-13. If that is the case, $\frac{1}{\sqrt{r}^k}$ is returned as well as a vector containing m_i's such that $\frac{1}{\sqrt{r}^k} \cdot m_i = v_i$ on line 15.

It might be the case that we first find a value $\frac{2}{\sqrt{2}}$ but later in the vector find $\frac{1}{\sqrt{8}}$. This second value cannot be written in the desired form if we keep $k = 1$, but we miss out if we don't check if the whole vector can also be simplified with $k = 3$ since $\frac{4}{\sqrt{8}} = \frac{2}{\sqrt{2}}$. This is why we move on to the next value in v if we can't simplify the vector using a particular k we found. Only when all entries have been used to find a suitable k and none have worked, we will return $1 \cdot v$.

This preprocessing algorithm will only work if all entries of the vector are either 0 or able to be written as approximately $\frac{m}{\sqrt{r}^k}$ with $m \in \mathbb{Z} \cdot \{1, i\}$, $r \in \{1, 2, 3, 5\}$ and $k \leq n$. If any entry is too far from any value of this form, Vecs2Pauli will just have to work with the original input vector. We have a certain amount of control over this, since we can decide when something is *almost an integer*, but since quantum states for larger numbers of qubits will have small values in its vector, we can't allow to round off too much. For example, rounding 0.8 to 1 will give incorrect/unexpected results.

Algorithm 5 A subroutine for preprocessing the input to Vecs2Pauli, factoring out powers of square roots of $1, 2, 3$ or 5

Procedure preprocessInput(vector v of length 2^n)
Output: complex number λ and vector m s.t. $\lambda \cdot m = v$

1: **for** every non-zero entry v_j in v **do**
2: **for** k in $[0, n]$ **do**
3: **for** r in $\{1, 2, 3, 5\}$ **do**
4: **if** $v_j \cdot \sqrt{r}^k$ is almost integer **then**
5: $foundK \leftarrow k$, $foundR \leftarrow r$
6: **exit** k-for-loop
7: **if** no $foundK$ **then**
8: **return** $1, v$
9: $\lambda \leftarrow \frac{1}{\sqrt{foundR}^{foundK}}$
10: **for** p in $[0, 2^n - 1]$ **do**
11: $m_p \leftarrow v_p \cdot \sqrt{foundR}^{foundK}$
12: **if** m_p is not almost integer **then**
13: **return to line 1 with next entry**
14: **if** $p = 2^n$ **then**
15: **return** λ, m
16: **return** $1, v$

References

[ALF+17] C. G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K Bertels. The engineering challenges in quantum computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 836–845, 2017.

[CCD+03] Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 59–68, New York, NY, USA, 2003. Association for Computing Machinery.

[CKD+21] Tim Coopmans, Robert Knegjens, Axel Dahlberg, David Maier, Loek Nijsten, Julio de Oliveira Filho, Martijn Papendrecht, Julian Rabbie, Filip Rozpedek, Matthew Skrzypczyk, and et al. Netsquid, a network simulator for quantum information using discrete events. *Communications Physics*, 4(1), 2021.

[Fey18] R.P. Feynman. Feynman and computation, 2018.

[FWH+10] Austin G. Fowler, David S. Wang, Charles D. Hill, Thaddeus D. Ladd, Rodney Van Meter, and Lloyd C. L. Hollenberg. Surface code quantum communication. *Phys. Rev. Lett.*, 104:180503, May 2010.

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

[Got98a] Daniel Gottesman. The heisenberg representation of quantum computers, 1998.

[Got98b] Daniel Gottesman. Theory of fault-tolerant quantum computation, 1998.

[MM19] Engineering Medicine., National Academies of Sciences and Medicine. *Quantum Computing: Progress and Prospects*. National Academies Press, 2019.

[Mon16] Ashley Montanaro. Quantum algorithms: An overview, Jan 2016.

[NC10] Michael Aaron NIELSEN and Isaac L. CHUANG. *Section 10.5*, page 454–455. Cambridge University Press, 10th anniversary edition, 2010.

[S.19] Gamble S. Quantum computing: What it is, why we want it, and how we're trying to get it., 2019. In: National Academy of Engineering. Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2018 Symposium.

[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.

[VCE+22] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. Limdd a decision diagram for simulation of quantum computing including stabilizer states, Aug 2022.