A COEFFICIENT MATRIX REPRESENTATION AND THE INVERSE COMPOSITION OF PAULI POLYNOMIAL. *

Hyunseong Kim
Department of Physics and Photon Science,
Gwangju Institute of Science and Technology
Gwangju
qwqwhsnote@gm.gist.ac.kr

ABSTRACT

In the research, we explored the proper coefficient matrix for Pauli polynomials focusing on enhancing the group structure representation. With coefficient elements and their indexes on the matrix, we can simultaneously manipulate the group and linear structure, more efficiently than the standard matrix representation of Pauli terms. The construction of the matrix is based on XZ simplex representation of each Pauli terms in the polynomial. Moreover, Pauli polynomial could be represented with a single complex matrix. We investigated two composition algorithms based on the coefficient matrix and tensorized decomposition algorithm suggested by Hantzko et al[1]. One is a naive basis transformation in each tensor producted spaces. The other is a modified transformation with effective term chasing routine. The composition time and spatial complexity of the investigated naive algorithms is atmost $\frac{1}{2}8^n$, and 4^n, respectively. Since, the coefficient matrix already contains all information of the terms in the polynomial, it is more efficient than previous term-by-term construction methods, which have $8^n + 2^n(f(n) - 2^n)$ and $2 * 4^n$.

Keywords Matrix composition · Pauli polynomial · Tensor product

1 Introduction

In quantum computing, converting between the Hamiltonian and Pauli polynomials is fundamental for both analyzing the system and constructing the evolution operator and its optimization. Since the Pauli polynomial indicates the local structures of the system, and Hamiltonian with Hermitian matrix form shows overall characteristics.

$$H = \sum_{i}^{n} \lambda_i P_i \tag{1}$$

Therefore, if the Hamiltonian is represented by a Hermitian matrix, a decomposition is required to get the coefficients of the terms. On the other hand, if the Pauli polynomial is given, one needs to construct Hermitian matrix by linear combinations of each term and condition. This process is called *composition*. Since, the Pauli matrices and their tensor products form an orthonormal basis of $2^n \times 2^n$ matrix Hilbert space[2], the decomposition and composition are well-defined with Hilbert-Schmidt inner product.

$$\sigma_0 = I, \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2}$$

There are well-known matrix representation of basic Pauli algebra, see Eq(2). In higher dimensions, the n-fold Pauli terms are represented with a tensor producted terms of the above single Pauli-matrices. However, in larger systems, the usage of the common matrix representation requires too much computational costs for its group and linear structures.

A simplex representation could be a solution of the problem. It maps each Pauli term to geometrical language of symplectic polar spaces[3]. There are several ways to map each term to the simplex representation, however, a common method is using a XZ family string representing Pauli term as product of two Pauli terms whose elements are only I, X or I, Z. The representation is expressed as a positive integer tuple by decompose the product string as matrix product of X, Z family strings. It is possible to implement the well-define binary operation for Pauli-group algebra. Since, the algebra implementation on matrix space have huge operational complexity comparing to single integer or binary operation, the simplex representation and their algebra implementation provide us a lot of convenience for generating and manipulating of the terms efficiently in computation frameworks. For example, Reggio et al showed that we can accelerate a determination of their commutation relationship with their simplex representation[4].

However, the manipulation and properties of Pauli terms does not only live in group structure. Hamiltonian analysis is also based on their vector property on complex field. Meanwhile, the simplex representation itself does not provide a linear structure of the Pauli basis in matrix space. One solution is constructing a single matrix which is a summation of all Pauli terms in their matrix form.

Therefore, it is also intense of study that the fast construction of a n-fold Pauli matrix from single pauli string, or simplex representations or decomposition of the given matrix into Pauli polynomials. There have been some studies about fast decomposition[1, 5, 6] and composition[7] routines,when a system is given with a hermite matrix, a set of coefficient, or a single Pauli string.

Another solution is allocating standard basis to each Pauli term, to indicate their linear structure. Since, Pauli matrices are orthonormal basis with Hilbert-Schmidt inner product[2], the mapping is well-defined. However, what is a proper basis mapping, that is useful to manipulate their group and linear structure simultaneously, is questionable. The standard basis has a lot of convenience, since, it is not only indicating of the linear structure, but also, preserves coefficient and Pauli terms in Pauli polynomial, as value and index of the elements.

Therefore, in this paper, we proposed a proper coefficient matrix to be used in Pauli polynomial representation. The matrix construction from simplex representations, and conversion methods to computational basis were investigated. The noticiable work in the previous studies is a tensorized pauli decomposition(TPD) algorithm studied by Hantzko et al[1]. The coefficient matrix refer in the paper was came out as a result of TPD algorithm.

In the following sections, we showed that the TPD is just a basis transformation represented with sequential local operations of the tensor producted spaces. By their property, unitary, it is automatically achieved the existence of inverse transformation. Lastly, a simple conversion operation was suggested that mapping between a simplex representation and index of the coefficient matrix of Pauli terms. With the knowledge, the coefficient matrix could be used for representing group and linear structure of Pauli matrices in general n-fold Pauli terms, fascillating fast conversion between matrix and its pauli decomposition. The main advantage of the coefficient matrix in composition routine is that the term by term construction and summation is not needed to build matrix representation of the Pauli polynomial. The summation is already archieved in the coefficient matrix, and the time complexity of the algorithm does not depend on the number of terms of the polynomial.

2 Background knowledges

2.1 Simplex representation

If we use σ_2' as one of pauli basis, where $\sigma_2 = i\sigma_2'$, and seperate phase to the outside, the tensor producted representation of n-fold Pauli terms has a next form.

$$P_g = \{\sigma_0, \sigma_1, \sigma_2', \sigma_3\} \tag{3}$$

$$P = (i)^m \otimes_j^n \sigma_j \tag{4}$$

where, m is a number of occurrence of σ_2' in the product. Since $\sigma_2' = \sigma_1\sigma_3$ holds true, we can decompose the given n-fold Pauli term as next two parts; elements of families. The example families are z-family and x-family refered by Reggio et al[4].

$$\otimes_j^n \sigma_j = \left(\otimes_{j\in\{0,1\}}^n \sigma_j\right)\left(\otimes_{k\in\{0,3\}}^n \sigma_k\right) \tag{5}$$

Eq(5) yields an unique representation as integer tuple of length 2 by replace $I \to 0, X, Z \to= 1$ in the each memebers.

$$P = (n_x, n_z) \tag{6}$$

where n_x, n_z are integers whose binary representation indicates $I = 0, X = 1, Z = 1$.

For example, $(6, 5)$ of 3-qubit system is a simplex representation of YXI.

$$\begin{array}{c} 6 = 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = X\otimes X\otimes I \\ 5 = 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = Z\otimes I\otimes Z \end{array} \tag{7}$$

IBM implemented the above simplex representation for their *Pauli* class in python library, Qiskit, to use the above binary implementation[8].

2.2 Tensorized decomposition

In 2023, Hantzko et al, showed that using tensorized notation general $\mathbf{M}_{2^n}(\mathbb{C})$ matrix efficiently decomposed into several Pauli terms with corresponding coefficients[1].

$$\sum_{i=0}^{3} c_i\sigma_i = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \to_{TPD} \begin{bmatrix} c_0 & c_1 \\ c_2 & c_3 \end{bmatrix} \tag{8}$$

where,

$$\begin{array}{rcl} A_{11} & = & c_0 + c_3 \\ A_{12} & = & c_1 - ic_2 \\ A_{21} & = & c_1 + ic_2 \\ A_{22} & = & c_0 - c_3. \end{array} \tag{9}$$

The basic idea is decoupling the coefficients of each tensor producted space, iteratively. See Figure **??**.

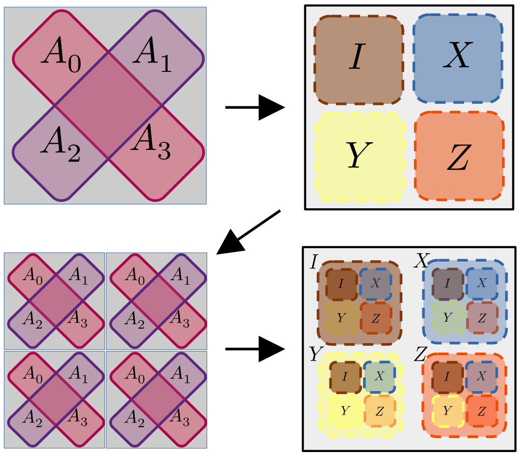


Figure 1: Iterative diagram of Tensorized Pauli Decomposition algorithm.

In their implementation, they did not map the index of the result cofficient matrix to each Pauli term. Therefore, they mapped the coefficients by adding a character to each string variables in each step of the iteration.

The decomposition process is non-linear in $2^n \times 2^n$ matrix space. However, it is a basis transformation in a higher dimension $\mathbb{C}^N$ space, which is isomorphic to the vector spaces with $N = 4^n$ dimension.

For example, the 2×2 dimension matrix of 1 qubit system, the process could be expressed as $\mathbf{v} \in \mathbb{C}^4$.

$$\frac{1}{2}\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & i & -i & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix}_{TPD} \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \tag{10}$$

In 2×2 matrix with computational basis, the next matrix is a *coefficient matrix*.

$$\begin{bmatrix} c_0 & c_1 \\ c_2 & c_3 \end{bmatrix} \tag{11}$$

In the vectorized representation, the intermiediate step of TPD algorithm could be represented with the next notation.

$$\begin{gathered} \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_1 \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_2 \otimes \cdots \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_{n-1} \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_n \\ \Downarrow \\ \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}_1 \otimes \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}_2 \otimes \cdots \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_{n-1} \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_n \end{gathered} \tag{12}$$

For the n-fold matrix, the researcher can choose the basis trnasformation, freely. The transformation even permits the different basis in each producted and each sub-matrix operations. The result coefficient matrices have identical coefficients without considering indexing. By choosing appropriate basis, the decompositon yields the Latin matrix with XZ simplex representation in Reggio et al[4]. In below sections, we refer the index of coefficients as *ij-index* in the result matrix by basis transformation refered from Hantzko et al.

3 Simplex to Coefficient map

The mapping could be vary in general cases, however for convenience, we only consider the general XZ simplex representation by Reggio et al[4] and ij-index generated from [1]. It is possible to use XZ representation as an index of Pauli terms, however, there is an efficient issues. T he XZ index is well-organized for group structure and the ij-index has a benefit in computational cost in decomposition and composition. By the object of the Pauli manipulation, the proper index could be different.

With these two representation of the same Pauli term, the next relationship is achieved.

Theorem 1. *For a given simplex representation, (n_x, n_z) of the given Paili term, P, their index, (i, j), in coefficient matrix is determined as*

$$(i, j) = (n_z, n_x {}^\wedge n_z)$$

where, $^\wedge$ is a XOR bitwise operator.

Proof From i-th iteration of the TPD algorithm of 2^n dim square matrix, the unit sub-matrix dimension is 2^{n-i} and there are 4 block matrices, see Figure **??**. With Eq(5), the result matrix of $i - th$ iteration is

$$\begin{bmatrix} \sigma_0 \cdot \sigma_0 & \sigma_1 \cdot \sigma_0 \\ \sigma_1 \cdot \sigma_3 & \sigma_0 \cdot \sigma_3 \end{bmatrix} = \begin{bmatrix} 0_x \cdot 0_z & 1_x \cdot 0_z \\ 1_x \cdot 1_z & 0_x \cdot 1_z \end{bmatrix} \tag{13}$$

where, $0_z, 1_z, 0_x, 1_x$ are XZ binary representation of Pauli term of i-th decimal. For row index, $2^i * nz_i, nz_i \in \{0_z, 1_z\}$ the Z-binary determine the row index movement, if $nz_i = 1_z$, the row location is changed else it is not. For column index, the column index changed by $+0$ if $(1_x, 0_z)$

or $(0_x, 1_z)$, else $+2^{n-i}$ if $(0_x, 0_z)$ or $(1_x, 1_z)$. It is a simple XOR binary opertor, therby $2^{n-i} * nz_i^{\wedge} nx_i$, $nz_i \in \{0_z, 1_z\}, nx_i \in \{0_x, 1_x\}$

Thus, we have (i, j) coefficient index of XZ representation by iteration from 1-th to n-th.

$$\begin{array}{rlcl} i = & \sum_{k=0}^{n-1} 2^k nz_k & = & nz \\ j = & \sum_{k=0}^{n-1} 2^k nz_k^{\wedge} nx_k & = & nz^{\wedge} nx \end{array} \tag{14}$$

where nz_k, nx_k are k-th binary element of nz, nx binary representation of the given Pauli term □.

With this relationship, we can construct a matrix representation of the given Pauli polynomial by inversing TPD algorithm. The terms would be located in corresponding elements in the coefficient matrix and the inverse composition algorithm would generate the original matrix.

3.1 Inverse algorithm

In the previous section, we already showed that TPD algorithm is a sequential applying of unitary transformation for each vectors in a product representation. By the property of unitary, the inverse transformation is well defined in 4^n dimension and we can implement it in 2^n with sub-block matrix additions. The $\{c_i\}_{i=0}^3$ coefficients in Eq (9) is restored as

$$\begin{array}{rcl} c_0 & = & \frac{1}{2}(A_{11} + A_{22}) \\ c_1 & = & \frac{1}{2}(A_{12} + A_{21}) \\ c_2 & = & \frac{i}{2}(A_{12} - A_{21}) \\ c_3 & = & \frac{1}{2}(A_{11} - A_{22}) \end{array} \tag{15}$$

.

The composition is achieved by iteratively applying the Eq (15) in reverse order of TPD algorithm. See, Fig (**??**).

The important of the algorithm is that the maximum time complexity is same between one pauli term and general pauli-polynomial. In addition, as has mentioned in [1], it does not need further instant matrix to save the terms, so that spatial complexity of the algorithm is also practical in large system. See details in the below complexity analysis section.

3.1.1 Effective term chasing algorithm

If we know an index set of Pauli terms, where their coefficients are not zero, we could avoid the operation for zero sub matrices terms in the intermediate steps of the composition. The non zero terms is denoted with *effective* terms. Considering I-Z and X-Y calculation, when k number of Pauli terms were given, there is an k_{eff} number of effective terms where

$$k = k_{eff} + d \, d \geq 0 \tag{16}$$

, d is a number of the duplicated terms.

Algorithm 1 Naive Inverse Composition Algorithm

Require: $M \leftarrow$ Coefficient matrix of $(2^n, 2^n)$
  matdim $\leftarrow 2^n$
  steps $\leftarrow$ n
  unit_size $\leftarrow$ 1
  for step **in** steps **do**
    step1 $\leftarrow$ step+1
    mat_size $\leftarrow 2 *$ unit_size
    indexes $\leftarrow$ [matdim/2^step1]
    indexes_ij $\leftarrow$ mat_size*indexes
    for i **in** indexes_ij **do**
      for i **in** indexes_ij **do**
        $r_{1s} \leftarrow$ i
        $r_{1f2s} \leftarrow r_{1s}$ + unit_size
        $c_{1s} \leftarrow$ j
        $c_{1f2s} \leftarrow c_{1s}$ + unit_size
        $r_{2f} \leftarrow r_{1f2s}$ + unit_size
        $c_{2f} \leftarrow c_{1f2s}$ + unit_size
        coef $\leftarrow$ 1
        $M[r_{1s}$: r_{1f2s}, c_{1s}:$c_{1f2s}]$ += coef*$M[r_{1f2s}$: r_{2f}, c_{1f2s}:$c_{2f}]$
        $M[r_{1f2s}$: r_{2f}, c_{1f2s}:$c_{2f}]$ = $M[r_{1s}$: r_{1f2s}, c_{1s}:$c_{1f2s}]$ -2*coef *$M[r_{1f2s}$: r_{2f}, c_{1f2s}:$c_{2f}]$
        coef $\leftarrow -\sqrt{-1}$
        $M[r_{1f2s}$: r_{2f}, c_{1s}:$c_{1f2s}]$ += coef*$M[r_{1s}$: r_{1f2s}, c_{1f2s}:$c_{2f}]$
        $M[r_{1s}$: r_{1f2s}, c_{1f2s}:$c_{2f}]$ = $M[r_{1f2s}$: r_{2f}, c_{1s}:$c_{1f2s}]$ -2*coef *$M[r_{1s}$: r_{1f2s}, c_{1f2s}:$c_{2f}]$
      end for
    end for
    unit_size $\leftarrow$ 2*unit_size
  end for

From the i-th effective index set, the effective index set for the next step is calculated by quotient of 2, such as

$$\begin{array}{rcl} row_{i+1} & = & r_i \\ col_{i+1} & = & c_i \end{array} \tag{17}$$

where, $row_i = 2 * m_i + r_i$ and $col_i = 2 * n_i + c_i$. The k_i number is same with $(k_{eff})_{i-1}$ number.

For example, in $2^4 \times 2^4$ coefficient matrix, we have (1, 14), (2, 13), (3, 1), (6, 4), (7, 4), (7, 5), (13, 9), (14, 10) elements as non-empty Pauli terms. We can chase the non-empty unit indexes with Eq (17)

$$\begin{array}{cccccc} & (1,14) & (0,7) & (0,3) & (0,1) & (0,0) \\ & (2,13) & (1,6) & (0,0) & (0,0) & \\ & (3,1) & (1,0) & (1,1) & (1,1) & \\ & (6,4) & (3,2) & (3,2) & - & \\ & (7,4) & (6,4) & - & - & \\ & (7,5) & (7,5) & - & - & \\ & (13,9) & - & - & - & \\ & (14,10) & - & - & - & \\ k & 8 & 6 & 4 & 3 & 1 \\ k_{eff} & 6 & 4 & 3 & 2 & 1 \end{array} \tag{18}$$

.

See 2 for further details.

Algorithm 2 Effective term algorithm

```
Require: poly = {((i, j))}_{l=1}^{k}          ▷ ij converted Pauli terms
Require: M ← Coefficient matrix of (2^n, 2^n)
  matdim ← 2^n
  steps ← n
  unit_size ← 1
  for step in steps do
      pstep ← []                                ▷ Empty list
      dup ← []
      for (i, j) in poly do
          if (i, j) in dup then continue
          end if
          n, o ← i%2, j%2                       ▷ IZ, XY determination
          l, m ← (i+1-2*(n), j+ 1-(2*(o)))      ▷ Get a corresponding location
          dup.insert((l,m) , (i,j))
          if n == 1 then
              pair ← ((l, m), (i, j))
          else
              pair ← ((i, j), (l, m))
          end if
          if (i+j)%2 == 1 then
              coef ← -√-1
          else
              coef ← 1
          end if
          r_1s ← unit_size * pair[0][0]
          r_1f ← r_1s + unit_size
          c_1s ← unit_size * pair[0][1]
          c_1f ← c_1s + unit_size
          r_2s ← unit_size * pair[1][0]
          r_2f ← r_2s + unit_size
          c_2s ← unit_size * pair[1][1]
          c_2f ← c_2s + unit_size
          M[r_1s: r_1f, c_1s:c_1f] += coef*M[r_2s: r_2f, c_2s:c_2f]
          M[r_2s: r_2f, c_2s:c_2f] = M[r_1s: r_1f, c_1s:c_1f] - 2*coef *M[r_2s: r_2f, c_2s:c_2f]
          i »=1                                 ▷ Bit shift operation
          j »=1
          if (i, j) in pstep then continue
          else: pstep.insert((i,j))
          end if
      end for
      poly ← pstep
      unit_size ← 2*unit_size
  end for
```

4 Comparsion to other method

4.1 Complexity analysis

4.1.1 Term-by-term methods

The general Pauli-composition methods focus on term-by-term matrix implementation. That is, with the given k-term Pauli polynomial, the previous methods generate k matrices corresponding to each tern and sum the matrices.

For $2^n \times 2^n$ matrices, if we denote the complexity of an algorithm for constructing a single Pauli matrix, $f(n)$, then the total composition complexity is estimated as,

$$k * f(n) + (k-1)4^n \tag{19}$$

where the 4^n term represents element wise addition complexity. Since k ranges from 1 to 4^n, the maximum complexity is

$$8^n + 2^n(f(n) - 2^n) \tag{20}$$

therefore, the term-by-term algorithms are fast with at-most 8^n time-complexity, unless the single routine complexity is lower than 2^n. Spatial complexity of temr-by-term methods is $2 \cdot 4^n$ by preparing zero matrix and iteratively adding each terms to the zero matrix.

4.1.2 Algorithm complexity

In the naive algorithm 1, the time complexity of each step is 4^n, and there are 2^{n-1} number of steps. Therefore, the total time complexity is

$$2^{n-1}4^n = \frac{1}{2}8^n \tag{21}$$

For the effective term algorithm, it is very complicated for estimating time-complexity, however, by taking worst case of effective term, we can estimate its upper bound.

With initial $k = k_{eff}+d$ number non zero-terms, the most hardness of the estimation is came out from the varying of the effective term numbers in the iteration. We assume that the worst case that $d = 0$ and we ignore the steps where the all elements is non zero. With the assumptions, we have

$$k_{eff,0} \in [\frac{1}{2}4^n], k_{eff,i} = \frac{1}{2}k_{eff,i-1} \tag{22}$$

with duplication search step of $O(k_{eff,i})$ complexity, the total time complexity consists of

$$2n * k_{eff}^2 + 34(2^{n+1} - 1)k_{eff} \tag{23}$$

The maximum complexity is not different in the naive version but it is still lower than any term-by-term algorithm. About the effective term algorithm, at some range of k_{eff}

value it is the most efficient but at very small or worts case, the naive version is more efficient. The worst complexity of the effective algorithm arises when $k = \frac{1}{2}4^n$ so that,

$$\frac{n}{2}16^n + 17(2 * 8^n - 4^n) \tag{24}$$

.

From Eq (23), the benefit region to use the effective term algorithm is

$$k < \frac{1}{2n}\left(\sqrt{289(2^{n+1})^2 + n8^n} - 17(2^{n+1} - 1)\right) \tag{25}$$

The efficient is archieved when the non zero terms are under 0.5% of 4^n number of terms.

4.2 Benchmarks with the other frameworks

The belows are brief reviews of current quantum frameworks. List of the fraameworks and methods are provides Pauli composition routine.

- PauliComposer[7].
- Qiskit Pauli, to_matrix method [8].
- Pennylane, Pauli to matrix [9].
- Cirq: unitary matrix transform [10].

Pennylane and Cirq's matrix conversion are just a simple kronecker product of each matrix terms. In the recent version of Pennylane, they provide *PauliSetence* class and matrix routine. However, the current implementation is not stable for test the general matrix composition.

Meanwhile, Qiskit routine is based on X, Z simplices representation of Pauli term and they implemented the composition routine with PauliComposer method which uses row wise mapping. They provides *PauliList* class aslike in Pennylane. The *to_matrix* routine generates rank-3 matrices for the given Pauli terms. However, the class does not support coefficient supports. The composition needs alternative summation with coefficient multiplication.

The main conversion were conducted with 4 implementations the inverse tensorized algorithm with using efficient term chasing routine or not, or pure python-numpy routine and numba acceleration.

Therefore, the comparsion with naive tensor product method and PauliComposer method is enough to see Pennylane and Qiskit frameworks. About the comparsion, the each routines are prepared as their intended data. For example, Pennylane prepares the single Pauli term as Pauli object class , and Qiskit requires them to be in the simplex representation.

The estimation was conducted for n-qubit system random matrices from $n = 1$ to $n = 9$ with 20%, 40%, 60%, 80%, 100% terms of 4^n polynomial. In the previous section we already showed that the effective term algorithm is efficient when only 0.5% terms are non zero in the space, however, for the practical application, we started from 20% non zero terms. The system specification and libraries are denoted on Table 1

Table 1: System specification for simulation.

Processor	AMD Ryzen 5 1600, Six-Core Processor, 3.20 GHz
RAM	32.0 GB
OS	Windows 10 Home, 64bit, 22H2
Python	3.11.8
Numpy[11]	1.26.4
Scipy[12]	1.13.0
Qiskit[8]	1.0.2
Pennylane[9]	0.35.1
PauliComposer[7]	Original paper version.

4.3 Results

In Fig (2), we tested the Pauli-composition methods in various quantum frameworks. There was no method considering Pauli-polynomials for matrix conversion, so that all the methods are term-by-term methods. *paulicomposer* has a $f(n) = 2^n$ time complexity, therefore, the total composition complexity is 8^n and standard tensor product method has $f(n) = 4^n$ time complexity, so that the total complexity is $2 \cdot 8^n - 4^n$.

The naive algorithm showed most efficient time costs for higher $n > 6$ system for all cases. It took 10^1 or 10^3 times faster than term-by-term methods. Even in the smaller system $n < 5$, the time costs were compatible with the term-by-term methods. The effective term algorithm showed better time costs in $n < 5$ and the non-zero terms accounted for the matrix below 60% percentage of the whole system. however, comparing to the naive algorithm there was no significant time benefit for the term chasing, it can be adopted to further applications but in the current stage, the improvement was not noticiable. Moreover, in the higher dimension system it overwhelms the term-by-term methods.

5 Conclusion

We showed that the tensorized decomposition algorithm was a sequential basis transformation of the given matrix. The common XZ simplex representation is one type of the transformation. A simple conversion between the simplex representation and an index of the coefficient matrix where the output result the original decomposition, TPD algorithm[1], was invetigated and the inverse composition algorithms were desgined. First algorithm is a naive tensor composition algorithm by inversing the TPD and using coefficient matrix. The other is an effective term algorithm by calculate non-zero coefficients during the iterative steps of the naive version. The algorithms are designed to compose the multiple terms at once, so that archieves better computational complexity in Pauli polynomial composition, in time and spatial both. The naive composition algorithm consists of basis transformation mapping between the coefficient matrix and the original matrix representation. Comparing to the previous term-by-term methods, the naive method is faster than the lowest complexity routines of the previous methods, 8^n with $\frac{1}{2}8^n$ complexity.

It means that we can construct the matrix with computational basis corresponding to the given Pauli-polynomial at process of the algorithm. The inverse algorithm could chasing an effective terms during the composition process. In addition, the composition speed comparsion between the current quantum computing frameworks, Qiskit, Pennylane, and naive tensor product routines. The inverse composition algorithm showed better speed for all cases, single, multi, worst terms for from $n = 2$ to $n = 9$ qubit cases. The naive algorithm was 10 or 1000 times faster than term-by-term methods. The effective term algorithm could chase the effective terms, however in the current stage the time-cost benefit was not noticiable in the implementation.

Acknowledgments

The research was funded by Quantum Sapiens Human resources center.

Data and code available

The research was conducted for sub module of OptTrot python package for fast manipulation and optimization routine for Hamiltonian. The tested code and the packages are available on OptTrot repository. The same version of the code in the paper is on Zenodo repository[13].

References

[1] Lukas Hantzko, Lennart Binkowski, and Sabhyata Gupta. Tensorized Pauli decomposition algorithm. 2023. Publisher: [object Object] Version Number: 2.

[2] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[3] Hans Havlicek, Boris Odehnal, and Metod Saniga. Moebius Pairs of Simplices and Commuting Pauli Operators, August 2009. arXiv:0905.4648 [math-ph, physics:quant-ph].

[4] Ben Reggio, Nouman Butt, Andrew Lytle, and Patrick Draper. Fast Partitioning of Pauli Strings into Commuting Families for Optimal Expectation Value Measurements of Dense Operators, June 2023. arXiv:2305.11847 [hep-lat, physics:hep-ph, physics:quant-ph].

[5] Océane Koska, Marc Baboulin, and Arnaud Gazda. A tree-approach Pauli decomposition algorithm with application to quantum computing. 2024. Publisher: [object Object] Version Number: 1.

[6] JiaWei Ying, JunChen Shen, Lan Zhou, Wei Zhong, MingMing Du, and YuBo Sheng. Preparing a Fast Pauli Decomposition for Variational Quantum Solving Linear Equations. *Annalen der Physik*, 535(11):2300212, November 2023.

[7] Sebastián Vidal Romero and Juan Santos-Suárez. PauliComposer: compute tensor products of Pauli matrices efficiently. *Quantum Information Processing*, 22(12):449, December 2023.

[8] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[9] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[10] Cirq Developers. Cirq, December 2023.

[11] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

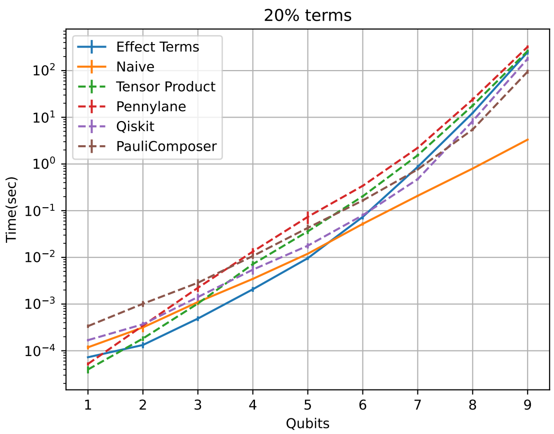
[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[13] Hyunseong Kim. A coefficient matrix representation and the inverse composition of Pauli polynomial., April 2024.

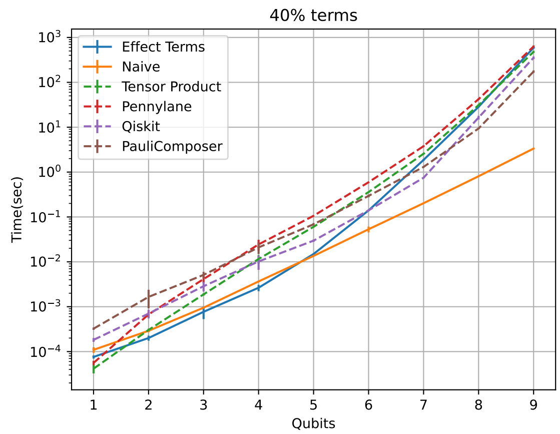
A Computational aspects

In real implementation, the chasing the efficient calculation term during the algorithm requires huge time complexity. The current implementation use bitwise operation, since, Python is not good for manipulate binary data, efficiently[2]. The above routines would be more appropriate for C/C++ or Rust like language implementation. We could observe that the binary compiled routines did not show difference whether the algorithm has a effective term chasing routine or not. Now, in python or the other interpreter language. It is wise to use the naive algorithm. and if the language environment naturally manipulate the bits, the effective term chasing version would be more appropriate.

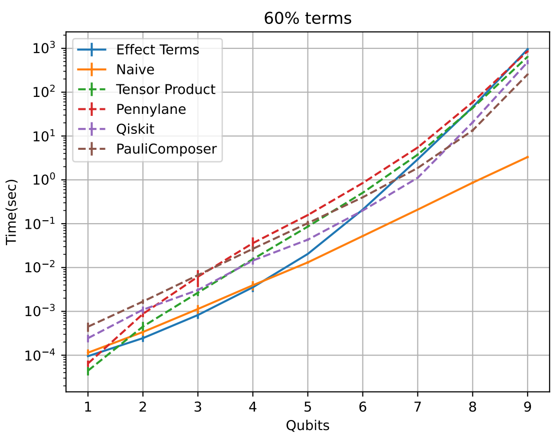
[2]Bitwise operators are even slower than string manipulation in Python.



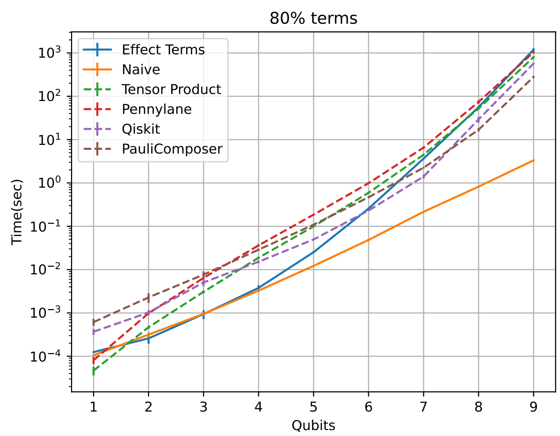
(a)



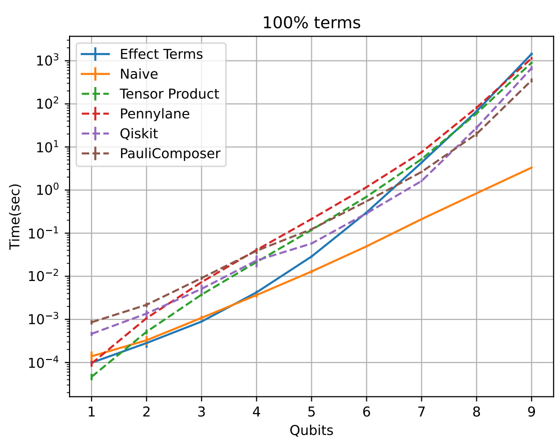
(b)



(c)



(d)



(e)

Figure 2: Benchmarks for matrix composition of Puali polynomials with the algorithm 1, 2 with Qiskit, Pennylane, PauliComposer, and standard tensor product methods, for $n = 1$ to $n = 9$. The percentages of the each case represents how many coefficients are non-empty in 4^n number of spaces.