

---

# FAST MATRIX RECONSTRUCTION OF GENERAL COMBINATION OF CLIFFORD ALGEBRA ELEMENTS

---

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 symplectic 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. One is a naive basis transformation in each tensor product spaces. The other is a modified transformation with effective term chasing routine. The composition time and spatial complexity of the investigated naive algorithms is at most  $\frac{1}{2}8^n$ , and  $4^n$ , respectively. Since, the coefficient matrix already contains all information of the terms in the polynomial, it is more efficient than general term-by-term construction methods, which have  $k * (f(n) + 4^n) \leq 16^n + 4^n(f(n) - 1)$  time complexity and  $2 * 4^n$  spatial complexity, where  $f(n)$  is a time complexity of single term construction method.

**Keywords** Matrix composition · Pauli polynomial · Tensor product

## 1 Introduction

Clifford algebra has become a dominant approach in various fields for analyzing complex systems, including computer graphics and robotics. In physics, it serves as a fundamental tool for representing the algebra of quantum quantities and has extensive applications in mathematical physics.

The algebra is not only used to replace linear algebra-based methods but is also served to study another approach in linear algebra. Conversely, by utilizing matrix representations, we can define general functions and inverses within Clifford algebra. Thus, both matrix representation and the Clifford approach are crucial in the study of linear and Clifford algebra systems, both practically and theoretically.

From a computational perspective, the matrix representation of Clifford elements offers advantages in terms of speed. Due to the inherently higher dimensions of the algebra, its time complexity is significantly greater than that of matrix calculations. Consequently, developing efficient implementations of Clifford algebra systems is an important area of research in computer science. Even

with highly optimized frameworks, matrix-based methods can demonstrate better performance in terms of time efficiency compared to geometric-based algorithms. For instance, when rapidly applying the same transformation to a large number of objects, matrix-based methods have the advantage, with the only bottleneck being the matrix initialization process.

Therefore, practical computation with Clifford algebra often requires switching representations depending on the specific situation to optimize the overall speed of the algorithm. In this paper, we present an inverse algorithm to generate a matrix representation of general elements of Clifford algebra, based on a generalized Pauli element schema. The method integrates tensorized decomposition with the symplectic representation of generalized Pauli elements, which is common in Clifford frameworks as binary tree method.

In general case, matrix reconstruction routines were \*single element reconstruction\* algorithm. Meanwhile, in here, the method is for general elements. For example, with quaternion system,  $\mathbb{H} \approx Cl_3$ .

$$M = a + bi + cj + dk \rightarrow \begin{bmatrix} a + bi & c + di \\ -c + di & a - bi \end{bmatrix}$$

$$H = \sum_i^n \lambda_i P_i \quad (1)$$

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, the decomposition and composition are well-defined with Hilbert-Schmidt inner product.

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

There are well-known matrix representation of basic Pauli algebra, see Eq(??). 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 symplectic representation could be a solution of the problem. It maps each Pauli term to geometrical language of symplectic polar spaces. There are several ways to map the each term to the symplectic 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 symplectic 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 symplectic representation.

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 symplectic 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 an intense of study that the fast construction of a  $n$ -fold Pauli matrix from single pauli string, or symplectic representations or decomposition of the given matrix into

Pauli polynomials. There have been some studies about fast decomposition??? and composition? 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, the mapping is well-defined. However, what is a proper basis mapping, that is useful to manipulate their group and linear structure simultaneously, is a 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 symplectic representations, and conversion methods to computational basis were investigated. The noticeable work in the previous studies is a tensorized pauli decomposition(TPD) algorithm studied by Hantzko et al. The coefficient matrix refer in the paper is 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 symplectic representation and index of the coefficient matrix of Pauli terms. With these knowledges, 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 symplectic representation

If we use  $\sigma'_2$  as one of pauli basis, where  $\sigma_2 = i\sigma'_2$ , and separate phase to the outside, the tensor producted representation of  $n$ -fold Pauli terms has a next form.

$$P_g = \{\sigma_0, \sigma_1, \sigma'_2, \sigma_3\} \quad (3)$$

$$P = (i)^m \otimes_j^n \sigma_j \quad (4)$$

where,  $m$  is a number of occurrence of  $\sigma'_2$  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 referred by Reggio et al?.

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

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

$$P = (n_x, n_z) \quad (6)$$

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 symplectic representation of YXI.

$$\begin{aligned} 6 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = X \otimes X \otimes I \\ 5 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = Z \otimes I \otimes Z \end{aligned} \quad (7)$$

IBM implemented the above symplectic representation for their *Pauli* class in python library, Qiskit, to use the above binary implementation?. However, in the paper, the order of the symplectic representation is reversed order of the IBM implementation. There is no difference in algebra implementation and commutation conversion routine, however, the conversion to index of coefficient matrix is more direct in the reversed order than the IBM order.

## 2.2 Tensorized decomposition

In 2023, Hantzko et al showed that general  $M_{2^n}(\mathbb{C})$  matrices are efficiently decomposed into several Pauli terms with corresponding coefficients, using a tensorized notation?.

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

where,

$$\begin{aligned} A_{11} &= c_0 + c_3 \\ A_{12} &= c_1 - ic_2 \\ A_{21} &= c_1 + ic_2 \\ A_{22} &= c_0 - c_3. \end{aligned} \quad (9)$$

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

In their implementation, they did not map the index of the result coefficient 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.

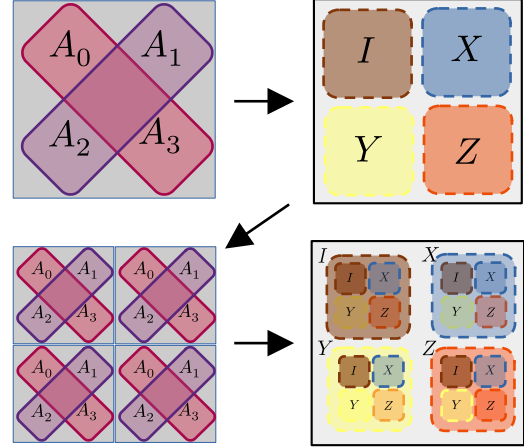


Figure 1: Iterative diagram of Tensorized Pauli Decomposition algorithm.

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

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

In  $2 \times 2$  matrix with computational basis, the next matrix is a *coefficient matrix* of 1 qubit system.

$$\begin{bmatrix} c_0 & c_1 \\ c_2 & c_3 \end{bmatrix} \quad (11)$$

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

$$\begin{aligned} & \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_1 \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_2 \otimes \dots \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_{n-1} \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_n \\ & \quad \downarrow \text{isteps, } i \geq 2 \\ & \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}_1 \otimes \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}_2 \otimes \dots \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_{n-1} \otimes \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix}_n \end{aligned} \quad (12)$$

For the  $n$ -fold matrix, the researcher can choose the basis transformation, 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 decomposition yields the Latin matrix whose element indexes are XZ symplectic representation of Pauli terms in Reggio et al?. In below sections, we refer

the index of coefficients as *ij-index* in the result matrix by basis transformation referred from Hantzko et al.

### 3 symplectic to Coefficient map

The mapping could be vary in general cases, however for convenience, we only consider the general XZ symplectic representation by Reggio et al? and ij-index generated from ?. It is possible to use XZ representation as an index of Pauli terms, however, there is an efficient issues. The 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 symplectic representation,  $(n_x, n_z)$  of the given Pauli 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(??), the result matrix of  $i$  - th iteration is

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

where,  $0_z, 1_z, 0_x, 1_x$  are XZ binary representation of Pauli term of  $i$ -th decimal. For row index,  $2^i * n_{z_i}, n_{z_i} \in \{0_z, 1_z\}$  the Z-binary determine the row index movement, if  $n_{z_i} = 1_z$ , 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 operator, thereby  $2^{n-i} * n_{z_i} \wedge n_{x_i}, n_{z_i} \in \{0_z, 1_z\}, n_{x_i} \in \{0_x, 1_x\}$

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

$$\begin{aligned} i &= \sum_{k=0}^{n-1} 2^k n_{z_k} = n_z \\ j &= \sum_{k=0}^{n-1} 2^k n_{z_k} \wedge n_{x_k} = n_z \wedge n_x \end{aligned} \quad (14)$$

where  $n_{z_k}, n_{x_k}$  are  $k$ -th binary element of  $n_z, n_x$  binary representation of the given Pauli term  $\square$ .

With this relationship, we can construct a matrix representation of the given Pauli polynomial by inverting 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 (??) is restored as

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

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

---

#### Algorithm 1 Naive Inverse Composition Algorithm

---

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

---

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 ?, 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

$$k = k_{eff} + dd \geq 0 \quad (16)$$

,  $d$  is a number of the duplicated terms.

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

$$\begin{aligned} row_{i+1} &= r_i \\ col_{i+1} &= c_i \end{aligned} \quad (17)$$

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, if we have (1, 14), (2, 13), (3, 1), (6, 4), (7, 4), (7, 5), (13, 9), (14, 10) elements were non-empty Pauli terms. We can chasing the non empty unit indexes with Eq (??)

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

See ?? for further details.

## 4 Comparson 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 term 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,

$$k * f(n) + (k - 1)4^n \quad (19)$$

---

### Algorithm 2 Effective term algorithm

---

**Require:** poly =  $\{((i, j))\}_{l=1}^k \triangleright$  ij converted Pauli terms

**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**

    pstep  $\leftarrow []$

$\triangleright$  Empty list

    dup  $\leftarrow []$

**for** (i, j) **in** poly **do**

**if** (i, j) **in** dup **then** continue

**end if**

        n, o  $\leftarrow i \% 2, j \% 2 \quad \triangleright$  IZ, XY determination

        l, m  $\leftarrow (i+1-2*(n), j+1-2*(o)) \quad \triangleright$  Get a

corresponding location

        dup.insert((l, m), (i, j))

**if** n == 1 **then**

            pair  $\leftarrow ((l, m), (i, j))$

**else**

            pair  $\leftarrow ((i, j), (l, m))$

**end if**

**if** (i+j)%2 == 1 **then**

            coef  $\leftarrow -\sqrt{-1}$

**else**

            coef  $\leftarrow 1$

**end if**

$r_{1s} \leftarrow \text{unit\_size} * \text{pair}[0][0]$

$r_{1f} \leftarrow r_{1s} + \text{unit\_size}$

$c_{1s} \leftarrow \text{unit\_size} * \text{pair}[0][1]$

$c_{1f} \leftarrow c_{1s} + \text{unit\_size}$

$r_{2s} \leftarrow \text{unit\_size} * \text{pair}[1][0]$

$r_{2f} \leftarrow r_{2s} + \text{unit\_size}$

$c_{2s} \leftarrow \text{unit\_size} * \text{pair}[1][1]$

$c_{2f} \leftarrow c_{2s} + \text{unit\_size}$

$M[r_{1s}: r_{1f}, c_{1s}: c_{1f}] += \text{coef} * M[r_{2s}: r_{2f},$

$c_{2s}: c_{2f}]$

$M[r_{2s}: r_{2f}, c_{2s}: c_{2f}] = M[r_{1s}: r_{1f}, c_{1s}: c_{1f}]$

$-2 * \text{coef} * M[r_{2s}: r_{2f}, c_{2s}: c_{2f}]$

        i  $\gg= 1$

$\triangleright$  Bit shift operation

        j  $\gg= 1$

**if** (i, j) **in** pstep **then** continue

**else:** pstep.insert((i, j))

**end if**

**end for**

    poly  $\leftarrow$  pstep

    unit\_size  $\leftarrow 2 * \text{unit\_size}$

**end for**

---

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

$$16^n + 4^n(f(n) - 1) \quad (20)$$

therefore, the term-by-term algorithms are fast with atmost  $16^n$  time-complexity. Spatial complexity of term-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 ??, the time complexity of each step is  $4^n$ , and there are  $2^{n-1}$  number of steps. Therefore, the total time complexity is

$$2^{n-1}4^n = \frac{1}{2}8^n \quad (21)$$

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

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

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

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

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,

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

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

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

The efficient is achieved 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?.
- Qiskit Pauli, to\_matrix method ?.
- Pennyane, Pauli to matrix ?.
- Cirq: unitary matrix transform ?.

Pennyane and Cirq's matrix conversion are just a simple kronecker product of each matrix terms. In the recent version of Pennyane, 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 Pennyane. 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 Pennyane and Qiskit frameworks. About the comparsion, the each routines are prepared as their intended data. For example, Pennyane prepares the single Pauli term as Pauli object class , and Qiskit requires them to be in the symplectic 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 ??

## 4.3 Results

In Fig (??), 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

Table 1: Summary of complexity of the algorithms with big-O notation.

-	Case	Time complexity	Spatial complexity
Term by term	Common	$O(k(f(n) + 4^n))$	$O(4^n)$
	Worst	$O(16^n)$	
Naive	Common	$O(8^n)$	
	Worst		$O(4^n)$
Effective term	Common	$O(n * k_{eff}^2)$	
	Worst	$O(n16^n)$	

Table 2: 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?	1.26.4
Scipy?	1.13.0
Qiskit?	1.0.2
PennyLane?	0.35.1
PauliComposer?	Original paper version.

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 noticable. 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 symplectic representation is one type of the transformation. A simple conversion between the symplectic representation and an index of the coefficient matrix where the output result the original decomposition, TPD algorithm?, was investigated and the inverse composition algorithms were designed. First algorithm is a naive tensor composition algorithm by inverting 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 achieves 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 which have  $O(16^n + 4^n(f(n) - 1))$  complexity in the worts case, the naive algorithm is at least, twice faster than the common term-by-term methods with  $O(8^n)$  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. However, chasing routine requires many computational costs it is not efficient as much as the naive version, and even comparable with the term-by-term methods with  $O(n16^n)$  complexity in the worst case.

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. Practically, even though the  $k$  is small, the naive version is comparable with the term-by-term methods. We assume that it is caused from the spatial complexity effect. The effective term algorithm could chase the effective terms, however in the current stage the time-cost benefit was not noticable 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?.

## 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<sup>1</sup>.

<sup>1</sup>Bitwise operators are even slower than string manipulation in Python.

## Convolution

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.



## Convolution

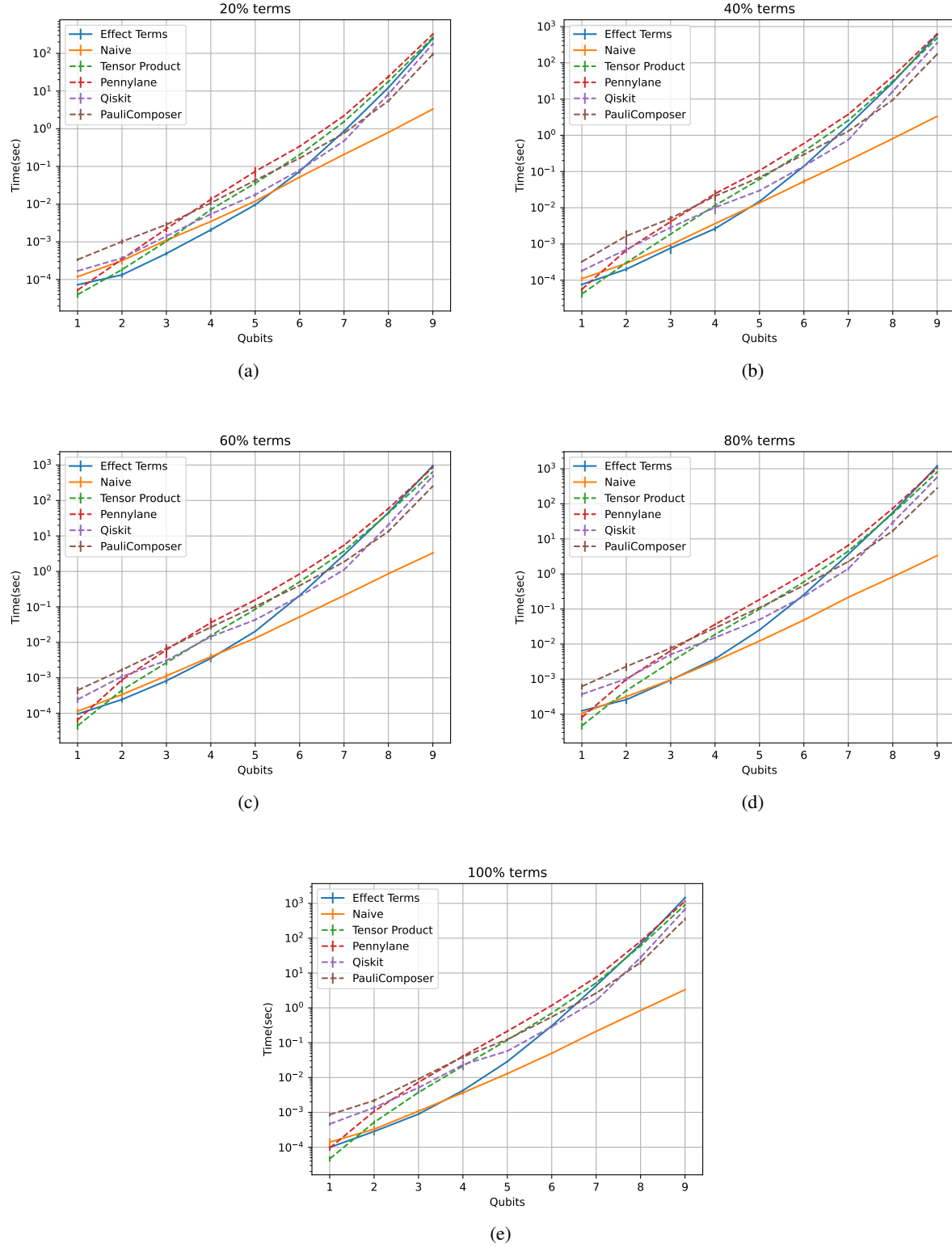


Figure 2: Benchmarks for matrix composition of Pauli polynomials with the algorithm ??, ?? with Qiskit, PennyLane, PauliComposer, and standard tensor product methods, for  $n = 1$  to  $n = 9$ . The percentages of the each case represents how many coefficients are non-empty in  $4^n$  number of spaces.