

# Tensor Networks for quantum circuit simulation and quantum error correction

*Jin-Guo Liu and Zhong-Yi Ni*

Advanced Materials Thrust, Function Hub, HKUST(GZ)

## Contents

Tensor Networks .....	2
Definition .....	2
Einsum notation and computational complexity .....	3
Contraction order optimization and slicing .....	7
Data Compression and Tensor Decomposition .....	12
Automatic Differentiation .....	16
Quantum Circuit Simulation .....	18
Quantum states and quantum gates .....	18
Example: Hadamard test .....	20
Example: Quantum teleportation .....	21
Quantum channel simulation .....	22
Kraus operators .....	22
Tensor network representation of channels .....	23
Quantum Error Correction .....	25
Stablizers and Quantum Codes .....	25
Surface code .....	26
Decoding problem .....	26
Tensor network decoder .....	28
Bibliography .....	28

# Tensor Networks

A *tensor network* is a fundamental concept in quantum physics and quantum information theory that provides a powerful diagrammatic representation for multilinear algebra operations. This framework shares similarities with *einsum* notation[1], *unweighted probability graphs*[2], *sum-product networks*, and *junction trees*[3] found in other computational domains.

Tensor networks have found widespread applications across diverse fields, including quantum circuit simulation[4], quantum error correction[5], neural network compression[6], and many-body quantum system dynamics[7]. Their versatility stems from their ability to efficiently represent and manipulate high-dimensional mathematical objects through intuitive graphical representations.

## Definition

At its core, a *tensor network* provides a diagrammatic representation of *multilinear algebra*. To understand this concept, let's first recall that linear algebra deals with linear functions satisfying two fundamental properties:

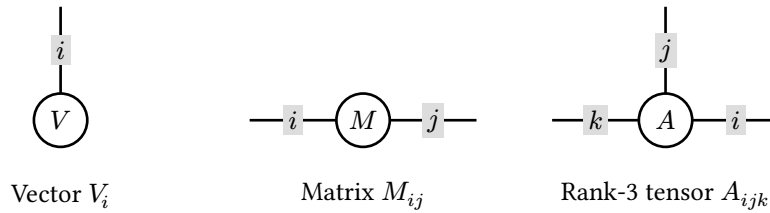
- Additivity:  $f(x + y) = f(x) + f(y)$  for any vectors  $x$  and  $y$
- Homogeneity:  $f(\alpha x) = \alpha f(x)$  for any scalar  $\alpha$

A function  $f$  is called *multilinear* if it maintains linearity with respect to each of its multiple arguments. For instance, the inner product of two vectors  $x$  and  $y$  is bilinear since it is linear in both  $x$  and  $y$ .

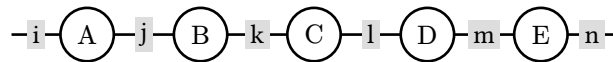
Consider the chain multiplication of matrices:

$$O_{in} = \sum_{j,k,l,m} A_{ij} B_{jk} C_{kl} D_{lm} E_{mn} \quad (1)$$

The output  $O_{in}$  depends linearly on each input tensor, making this a *multilinear map* known as *tensor contraction*. Tensor networks extend this concept to arbitrary tensors with multiple indices, where we represent each tensor as a node and each index as a connecting edge or "leg." This graphical notation provides an intuitive way to visualize complex multilinear operations.



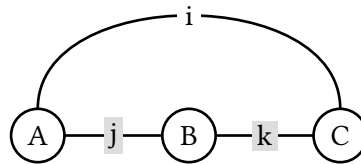
The diagrammatic representation of Equation 1 reveals the underlying structure more clearly:



This diagrammatic representation offers significant advantages over algebraic notation by making the computational structure immediately visible. The connected indices represent summation variables, while unconnected indices correspond to the output tensor's dimensions. This visual clarity becomes particularly valuable when analyzing complex tensor contractions, as demonstrated in the following example.

**Example: Proving the trace permutation rule**

Consider three square matrices  $A$ ,  $B$ , and  $C$  of the same dimension. The trace permutation rule states that  $\text{tr}(ABC) = \text{tr}(CAB) = \text{tr}(BCA)$ . This identity can be elegantly demonstrated using tensor network diagrams.



In this diagram, the cyclic connection of indices creates a closed loop that represents the trace operation. Regardless of which matrix we designate as the “starting point,” the topological structure remains invariant. This visual proof immediately reveals why the three expressions  $\text{tr}(ABC)$ ,  $\text{tr}(CAB)$ , and  $\text{tr}(BCA)$  are equivalent—they correspond to identical tensor network contractions. The diagrammatic approach thus provides a more intuitive understanding than algebraic manipulation alone.

## Einsum notation and computational complexity

In computational implementations, tensor network topologies are commonly specified using einsum notation—a compact string representation that encodes the contraction structure. In this notation,

- `->` separates the input and output tensors
- `,` separates the indices of different input tensors
- each char represents an index

For example, matrix multiplication  $C = AB$  is represented as `ij,jk->ik`, where the two input matrices are represented by `ij` and `jk`, and the output matrix is represented by `ik`.

The following examples use the [OMEinsum](#) package to demonstrate tensor network specification, contraction order optimization, and execution. Tensor network topologies can be defined using either the convenient `ein` string literal or the more flexible `EinCode` constructor for programmatic construction.

```
julia> using OMEinsum

julia> code = ein"ab,bc,cd->ad" # using string literal
ab, bc, cd -> ad

julia> EinCode([1,2], [2, 3], [3, 4]), [1, 4]) # alternatively
1*2, 2*3, 3*4 -> 1*4
```

Its defining properties can be obtained with the `getixsv` and `getiyv` functions.

```
julia> getixsv(code) # get the indices of the input tensors
3-element Vector{Vector{Char}}:
 ['a', 'b']
 ['b', 'c']
 ['c', 'd']

julia> getiyv(code) # get the indices of the output tensor
2-element Vector{Char}:
 'a'
 'd'
```

```
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

The complexity of the contraction can be computed with the `contraction_complexity` function. It requires the sizes of the indices, which can be specified with a dictionary that maps the indices to their sizes. Here, we use the `uniformsize` function to specify that all indices have the same size.

```
julia> label_sizes = uniformsize(code, 100) # define the sizes of the indices
Dict{Char, Int64} with 4 entries:
  'a' => 100
  'c' => 100
  'd' => 100
  'b' => 100

julia> contraction_complexity(code, label_sizes)
Time complexity: 2^26.575424759098897
Space complexity: 2^13.287712379549449
Read-write complexity: 2^15.287712379549449
```

Contraction complexity can be analyzed from multiple complementary perspectives:

- **Time complexity** ( $100^4$  operations): Represents the total number of floating-point operations (FLOPs) required for the contraction. For einsum operations, this equals the product of all unique index dimensions, since each unique index either participates in summation or appears in the output. As we'll see later, smart contraction ordering can dramatically reduce this complexity.
- **Space complexity** ( $100^2$  elements): Measures the peak memory requirement for storing the largest intermediate tensor generated during contraction. This determines the minimum memory needed to execute the computation.
- **Read-write complexity** ( $4 \times 100^2$  operations): Quantifies total memory bandwidth usage by counting all floating-point numbers transferred between memory and processor throughout the contraction. This metric captures the cumulative cost of accessing all intermediate tensors and often determines real-world performance on bandwidth-limited systems.

While EinCode objects are callable and can directly perform contractions:

```
julia> code(randn(2, 2), randn(2, 2), randn(2, 2)) # not recommended
2×2 Matrix{Float64}:
-0.974692  3.06151
-0.674225  1.40281
```

This approach is **strongly discouraged** because OMEinsum uses an unoptimized contraction order that may be exponentially inefficient. A better approach explicitly specifies the contraction order using parentheses:

```
julia> nested_code = ein"(ab,bc),cd->ad"
ac, cd -> ad
├─ ab, bc -> ac
│ └─ ab
```

```
|  └ bc
└  └ cd
```

The resulting NestedEinsum object represents a structured two-step contraction: first computing the intermediate tensor from the first two inputs, then contracting this result with the third tensor. This explicit ordering achieves dramatic complexity reduction:

```
julia> contraction_complexity(nested_code, label_sizes)
Time complexity: 2^20.931568569324174
Space complexity: 2^13.287712379549449
Read-write complexity: 2^15.872674880270605
```

Beyond theoretical complexity improvements, practical performance gains are even more substantial. OMEinsum leverages optimized BLAS routines for binary tensor contractions, leading to remarkable speedups:

```
julia> using BenchmarkTools

julia> @btime code(randn(100, 100), randn(100, 100), randn(100, 100)); #
unoptimized
 86.418 ms (36 allocations: 385.48 KiB)

julia> @btime nested_code(randn(100, 100), randn(100, 100), randn(100, 100)); #
optimized
 133.167 μs (157 allocations: 486.06 KiB)
```

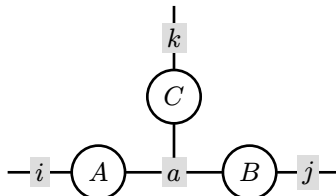
This represents over 600× speedup, demonstrating how proper contraction ordering transforms intractable computations into practical ones.

### Example A: Star contraction

The star contraction of three matrices  $A, B, C \in \mathbb{R}^{n \times n}$  is defined as:

$$O_{ijk} = \sum_a A_{ia} B_{aj} C_{ak} \quad (2)$$

This operation creates a 3-way tensor by connecting all matrices through a shared summation index:



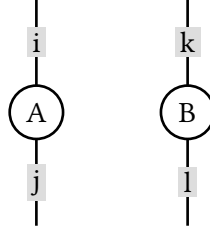
The einsum notation is  $ai, aj, ak \rightarrow ijk$  with time complexity  $O(n^4)$ , where the shared index  $a$  creates the characteristic “star” topology.

### Example B: Kronecker product

The Kronecker product of two matrices  $A, B \in \mathbb{R}^{n \times n}$  is defined as:

$$C_{ijkl} = A_{ij}B_{kl} \quad (3)$$

Unlike the star contraction, this operation has no shared indices:



The einsum notation is  $ij, kl \rightarrow ijkl$  with time complexity  $O(n^4)$ . The absence of connections reflects the direct product structure.

### Tensor network contraction is #P-complete

Contracting a tensor is hard, which is in #P-complete (harder than the famous NP-complete). Showing a problem is hard can be done through reduction. If we can reduce problem  $\mathcal{A}$  to problem  $\mathcal{B}$ , which means by solving problem  $\mathcal{B}$  (in time polynomial to input size), we can solve problem  $\mathcal{A}$  with the answer to  $\mathcal{B}$ . Then  $\mathcal{B}$  is not easier than  $\mathcal{A}$  from computational complexity perspective.

The computational complexity of general tensor network contraction can be established by reduction from a known #P-complete problem: counting satisfying assignments of 2-SAT formulas.

**Definition 1** (2-SAT formula): A 2-SAT formula is a Boolean formula in conjunctive normal form (CNF) where each clause contains at most two literals. For those who are not familiar with boolean logic, a clause is a disjunction (logical or:  $\vee$ ) of literals, and a literal is a boolean variable or its negation ( $\neg$ ). A boolean formula can always be represented as a conjunction (logical and:  $\wedge$ ) of clauses, which is called the conjunctive normal form.

#### Example:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_4 \vee x_5) \wedge (x_5 \vee x_1) \wedge (x_3 \vee \neg x_5) \quad (4)$$

- A satisfying assignment is:  $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1$ .
- A non-satisfying assignment is:  $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 1$ , since it violates  $x_2 \vee x_3, x_3 \vee x_4$  and  $x_3 \vee \neg x_5$ .

The counting of 2-SAT formula asks how many satisfying assignments are there.

While determining satisfiability (finding any solution) for 2-SAT formulas is polynomial-time solvable, counting the **number** of satisfying assignments is #P-complete—a complexity class considered even more challenging than NP-complete problems.

The reduction proceeds by encoding the 2-SAT counting problem as a tensor network:

**Step 1: Clause encoding.** The boolean variables  $x_1, x_2, \dots, x_n$  directly maps to (hyper)edges in tensor network, now we need to decide the tensors relating these variables. Each clause becomes a rank-2 tensor encoding its truth table. For the clause  $(x_3 \vee \neg x_5)$ , we construct tensor  $T_{+-}$ :

$$T_{+-} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad (5)$$

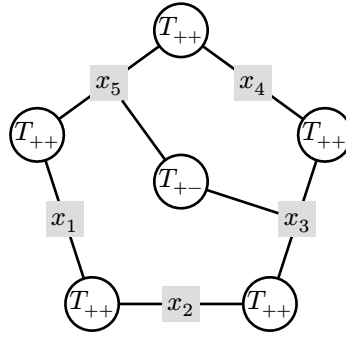
where rows correspond to  $x_3 \in \{0, 1\}$  and columns to  $x_5 \in \{0, 1\}$ . The entry  $(T_{+-})_{0,1} = 0$  indicates that  $x_3 = 0, x_5 = 1$  makes the clause false. Similar tensors  $T_{++}$ ,  $T_{--}$ , and  $T_{-+}$  encode other clause types.

**Step 2: Network construction.** The counting problem reduces to the tensor contraction:

$$\text{count} = \sum_{x_1, x_2, \dots, x_n} \prod_{\text{clauses}} T_{\text{clause}} \quad (6)$$

where the summation spans all Boolean assignments and the product combines all clause tensors. This contraction precisely counts satisfying assignments. Note tensor network contraction corresponds to sum of product of elements from each tensor, whenever a tensor contributes a zero multiplication factor, the net contribution of this assignment is 0. On the other hand, since we have a 0-1 element only, if all the tensors contribute a 1 multiplication factor (means this constraint is satisfied), the net contribution of this assignment is 1. Hence the contraction corresponds to the counting of true assignments.

For the 2-SAT formula in Equation 4, the corresponding tensor network is:

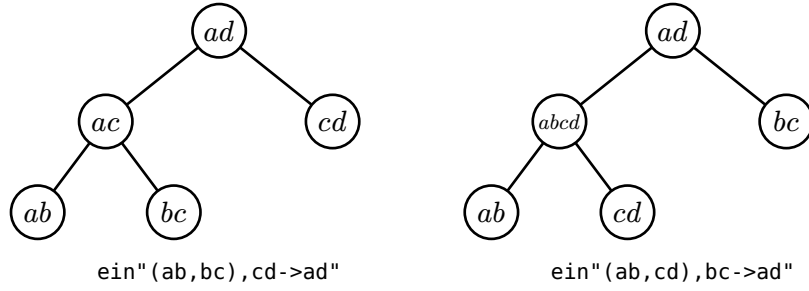


Since counting satisfying assignments for 2-SAT is #P-complete, and we have demonstrated a polynomial-time reduction from this problem to tensor network contraction, it follows that general tensor network contraction is also #P-complete. This establishes tensor network optimization as fundamentally intractable, motivating the development of approximation algorithms and heuristic methods discussed in subsequent sections.

### Contraction order optimization and slicing

The computational cost of tensor network contraction depends critically on the chosen **contraction order**—the sequence in which pairwise tensor multiplications are performed. This order can be represented as a binary tree where leaves correspond to input tensors and internal nodes represent intermediate results.

Consider the contraction  $\text{ein} "ab, bc, cd \rightarrow ad"$ , which admits multiple valid orderings with dramatically different costs:



The left ordering is dramatically superior: it achieves  $O(n^3)$  time and  $O(n^2)$  space complexity by first contracting compatible matrices. The right ordering creates a  $O(n^4)$  intermediate tensor through an inefficient Kronecker product, illustrating how ordering choice can determine computational feasibility.

Finding the globally optimal contraction order constitutes an NP-complete optimization problem[4]. Fortunately, near-optimal solutions often suffice for practical applications and can be obtained efficiently through sophisticated heuristic methods. Modern optimization algorithms have achieved remarkable scalability, successfully handling tensor networks with over  $10^4$  tensors[8],[9].

The optimal contraction order has a deep mathematical connection to the *tree decomposition*[4] of the tensor network's line graph.

**Definition 2** (Tree decomposition and treewidth): A *tree decomposition* of a (hyper)graph  $G = (V, E)$  is a tree  $T = (B, F)$  where each node  $B_i \in B$  contains a subset of vertices in  $V$  (called a "bag"), satisfying:

1. Every vertex  $v \in V$  appears in at least one bag.
2. For each (hyper)edge  $e \in E$ , there exists a bag containing all vertices in  $e$ .
3. For each vertex  $v \in V$ , the bags containing  $v$  form a connected subtree of  $T$ .

The *width* of a tree decomposition is the size of its largest bag minus one. The *treewidth* of a graph is the minimum width among all possible tree decompositions.

The line graph of a tensor network is a graph where vertices represent indices and edges represent tensors sharing those indices. The relationship between a tensor network's contraction order and the tree decomposition of its line graph can be understood through several key correspondences:

- Each leg (index) in the tensor network becomes a vertex in the line graph, while each tensor becomes a hyperedge connecting multiple vertices.
- The tree decomposition's first two requirements ensure that all tensors are accounted for in the contraction sequence - each tensor must appear in at least one bag, with each bag representing a contraction step.
- The third requirement of the tree decomposition maps to an important constraint in tensor contraction: an index can only be eliminated after considering all tensors connected to it.
- For tensor networks with varying index dimensions, we can extend this relationship to weighted tree decompositions, where vertex weights correspond to the logarithm of the index dimensions.

The figure below illustrates these concepts with (a) a tensor network containing four tensors  $T_1, T_2, T_3$  and  $T_4$  and eight indices labeled  $A$  through  $H$ , (b) its corresponding line graph, and (c) a tree decomposition of that line graph.



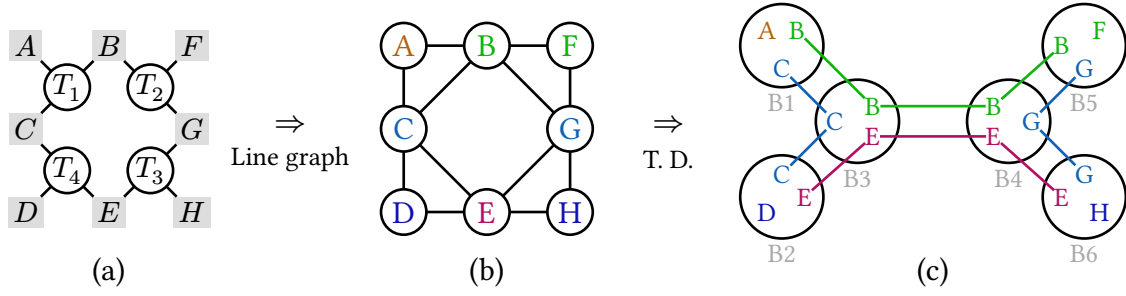
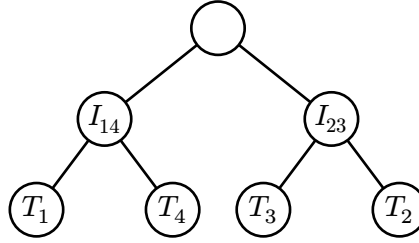


Figure 2: (a) A tensor network. (b) A line graph for the tensor network. Labels are connected if and only if they appear in the same tensor. (c) A tree decomposition (T. D.) of the line graph.

The tree decomposition in (c) consists of 6 bags, each containing at most 3 indices, indicating that the treewidth of the tensor network is 2. The tensors  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_4$  are contained in bags  $B_1$ ,  $B_5$ ,  $B_6$  and  $B_2$  respectively. Following the tree structure, we perform the contraction from the leaves. First, we contract bags  $B_1$  and  $B_2$  into  $B_3$ , yielding an intermediate tensor  $I_{14} = T_1 * T_4$  (where “ $*$ ” denotes tensor contraction) with indices  $B$  and  $E$ . Next, we contract bags  $B_5$  and  $B_6$  into  $B_4$ , producing another intermediate tensor  $I_{23} = T_2 * T_3$  also with indices  $B$  and  $E$ . Finally, contracting  $B_3$  and  $B_4$  yields the desired scalar result.



Finding the optimal contraction order is almost equivalent to finding the minimal-width tree decomposition of the line graph. The log time complexity for the bottleneck contraction corresponds to the largest bag size in the tree decomposition. The log space complexity is equivalent to the largest separator (the set of vertices connecting two bags) size in the tree decomposition.

### Heuristic methods for finding the optimal contraction order

OMEinsum provides multiple heuristic methods for finding the optimal contraction order. They are implemented in the dependency `OMEinsumContractionOrders`. To demonstrate the usage, we first generate a large enough random tensor network with the help of the `Graphs` package.

```
julia> using OMEinsum, Graphs

julia> function demo_network(n::Int)
    g = random_regular_graph(n, 3)
    code = EinCode([e.src, e.dst] for e in edges(g)), Int[]
    sizes = uniformsize(code, 2)
    tensors = [randn([sizes[leg] for leg in ix]...) for ix in
getixsv(code)]
    return code, tensors, sizes
end

demo_network (generic function with 1 method)

julia> code, tensors, sizes = demo_network(100);

julia> contraction_complexity(code, sizes)
```

```
Time complexity: 2100.0  
Space complexity: 20.0  
Read-write complexity: 29.231221180711184
```

We first generate a random 3-regular graph with 100 vertices. Then we associate each vertex with a binary variable and each edge with a tensor of size  $2 \times 2$ . The time complexity without contraction order optimization is  $2^{100}$ , which is equivalent to brute-force. The order can be optimized with the `optimize_code` function.

```
julia> optcode = optimize_code(code, sizes, TreeSA());  
  
julia> cc = contraction_complexity(optcode, sizes)  
Time complexity: 217.241796993093228  
Space complexity: 213.0  
Read-write complexity: 216.360864226366807
```

The `optimize_code` function takes three inputs: the `EinCode` object, the tensor sizes, and the contraction order solver. It returns a `NestedEinsum` object of time complexity  $\sim 2^{17.2}$ . It is much smaller than the number of vertices. It is a very reasonable number because the treewidth of a 3-regular graph is approximately upper bounded by  $1/6$  of the number of vertices[10].

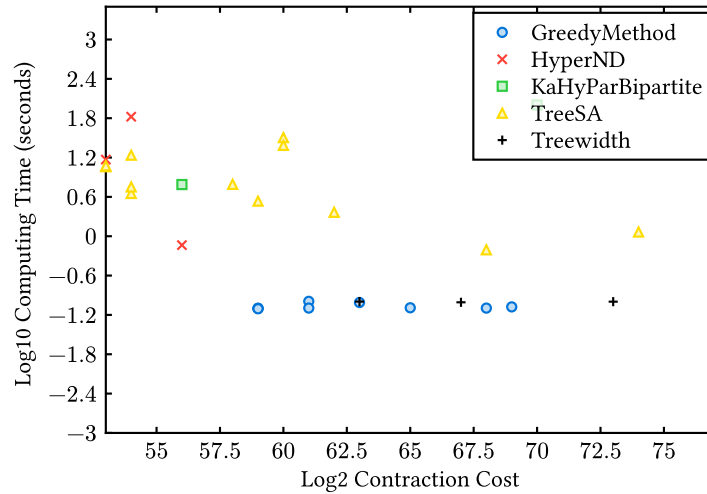


Figure 3: The contraction order quality measured by the space complexity ( $x$ -axis) and time complexity ( $y$ -axis) for different methods with different hyper-parameters. For details, please check GitHub repository [OMEinsumContractionOrdersBenchmark](#).

Among the available solver backends, `TreeSA` and `HyperND` usually provide the best contraction order quality. However, they are slow. For overhead sensitive applications, one can use `GreedyMethod` or `Treewidth` method.

In the following, we introduce the local search method `TreeSA` in detail.

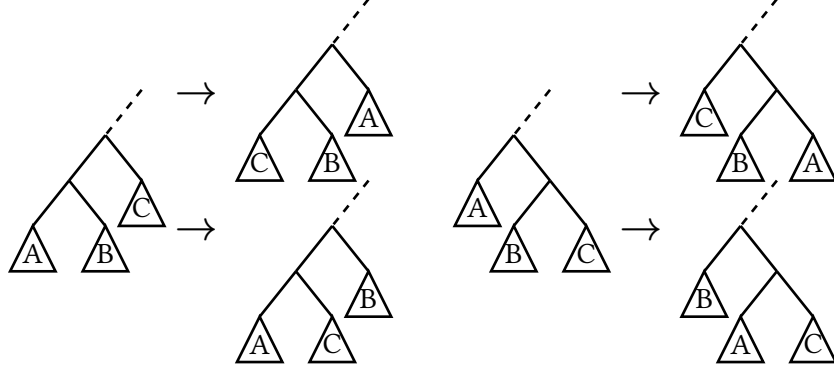


Figure 4: The four basic local transformations on the contraction tree, which preserve the result of the contraction.

The local search method[11] is a heuristic method based on the idea of simulated annealing. The method starts from a random contraction order and then applies the following four possible transforms as shown in Figure 4, which correspond to the different ways to contract three sub-networks:

$$\begin{aligned} (A * B) * C &= (A * C) * B = (C * B) * A, \\ A * (B * C) &= B * (A * C) = C * (B * A), \end{aligned} \quad (7)$$

where  $A, B, C$  are the sub-networks to be contracted. Due to the commutative property of the tensor contraction, such transformations do not change the result of the contraction. Even through these transformations are simple, all possible contraction orders can be reached from any initial contraction order. The local search method starts from a random contraction tree. In each step, the above rules are randomly applied to transform the tree and then the cost of the new tree is evaluated, which is defined as

$$\mathcal{L} = \text{tc} + w_s \text{sc} + w_{\text{rw}} \text{rwc}, \quad (8)$$

where  $w_s$  and  $w_{\text{rw}}$  are the weights of the space complexity and read-write complexity compared to the time complexity, respectively. Then the transformation is accepted with a probability given by the Metropolis criterion, which is

$$p_{\text{accept}} = \min(1, e^{-\beta \Delta \mathcal{L}}), \quad (9)$$

where  $\beta$  is the inverse temperature, and  $\Delta \mathcal{L}$  is the difference of the cost of the new and old contraction trees. During the process, the temperature is gradually decreased, and the process stop when the temperature is low enough. Additionally, the TreeSA method supports the slicing technique. When the space complexity is too large, one can loop over a subset of indices, and then contract the intermediate results in the end. Such technique can reduce the space complexity, but slicing  $n$  indices will increase the time complexity by  $2^n$ .

### Slicing Technique

Slicing is a technique to reduce the space complexity of the tensor network by looping over a subset of indices. This effectively reduces the size of the tensor network inside the loop, and the space complexity can potentially be reduced. For example, in Figure 5, we slice the tensor network over the index  $i$ . The label  $i$  is removed from the tensor network, at the cost of contraction multiple tensor networks.

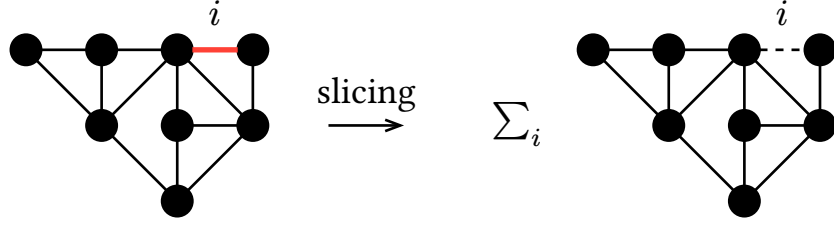


Figure 5: The slicing technique. The tensor network is sliced over the index  $i$ .

Continuing from the previous example, we can use the `slice_code` function to reduce the space complexity.

```
julia> sliced_code = slice_code(optcode, sizes,
TreeSASlicer(score=ScoreFunction(sc_target=cc.sc-3)));

julia> sliced_code.slicing
3-element Vector{Int64}:
 14
 76
 60

julia> contraction_complexity(sliced_code, sizes)
Time complexity: 2^17.800899899920303
Space complexity: 2^10.0
Read-write complexity: 2^17.199595668955244
```

The `slice_code` function takes three inputs: the `NestedEinsum` object, the tensor sizes, and the slicing strategy. Here, we use the `TreeSASlicer` with the `ScoreFunction` to reduce the space complexity by 3. The result type is `SlicedEinsum`, which contains a `slicing` field for storing the slice indices. After slicing, the space complexity is reduced by 3, while the time complexity is only slightly increased. The usage of `SlicedEinsum` is the same as the `NestedEinsum` object.

```
julia> @assert sliced_code(tensors...) ≈ optcode(tensors...)
```

## Data Compression and Tensor Decomposition

Let us define a complex matrix  $A \in \mathbb{C}^{m \times n}$ , and let its singular value decomposition be

$$A = USV^\dagger \quad (10)$$

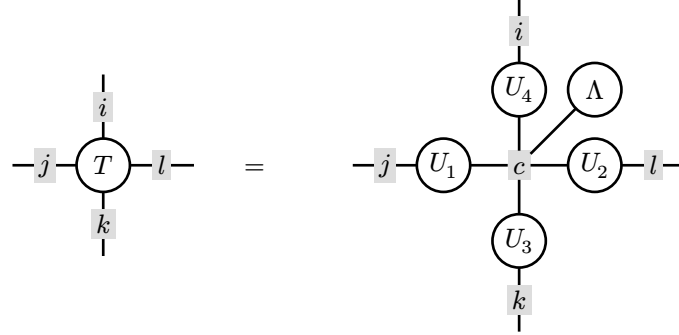
where  $U$  and  $V$  are unitary matrices and  $S$  is a diagonal matrix with non-negative real numbers on the diagonal. Let  $s$  be the diagonal part of  $S$ , the diagrammatic representation of SVD decomposition is

Let us denote  $d_i = \dim(i)$ ,  $d_j = \dim(j)$ ,  $d_k = \dim(k)$ ,  $d_s = \dim(s)$ . For data compression, we require  $d_k < \min(d_i, d_j)$ , the compression ratio can be computed as:  $\frac{d_i d_j}{d_k (d_i + d_j)}$ .

### CP-decomposition

For example, the CP-decomposition of a rank-4 tensor  $T$  can be represented as

$$T_{ijkl} = \sum_c U_1^{ic} U_2^{jc} U_3^{kc} U_4^{lc} \Lambda_c \quad (11)$$



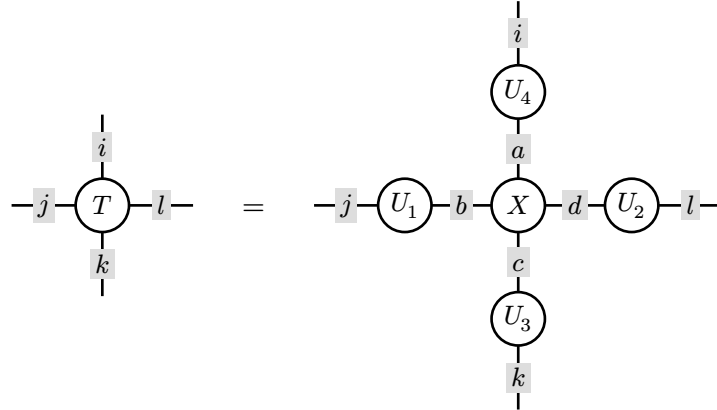
The data compression ratio for CP-decomposition is  $\frac{\prod_{i=1}^N d_i}{R \sum_{i=1}^N d_i}$ , where  $d_i$  is the dimension of the  $i$ -th mode,  $N$  is the number of modes, and  $R$  is the rank (dimension of the shared index  $c$ ). For the rank-4 case shown above, this becomes  $\frac{d_i d_j d_k d_l}{R(d_i + d_j + d_k + d_l + 1)}$ .

### Tucker decomposition

The Tucker decomposition of a rank-4 tensor  $T$  can be represented as

$$T_{ijkl} = \sum_{a,b,c,d} U_1^{ia} U_2^{jb} U_3^{kc} U_4^{ld} X_{abcd} \quad (12)$$

where  $U_1, U_2, U_3, U_4$  are unitary matrices and  $X$  is a rank-4 tensor.

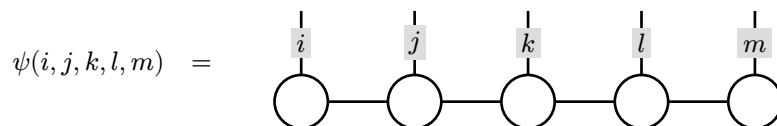


The data compression ratio for Tucker decomposition is  $\frac{\prod_{i=1}^N d_i}{\prod_{i=1}^N r_i + \sum_{i=1}^N d_i r_i}$ , where  $d_i$  is the dimension of the  $i$ -th mode,  $N$  is the number of modes, and  $r_i$  is the dimension of the  $i$ -th core tensor mode. For the rank-4 case shown above, this becomes  $\frac{d_i d_j d_k d_l}{r_a r_b r_c r_d + d_i r_a + d_j r_b + d_k r_c + d_l r_d}$ .

Tucker decomposition is more flexible than CP decomposition as it allows different compression ratios for different modes, but it suffers from the curse of dimensionality as the core tensor  $X$  grows exponentially with the number of modes.

### Tensor Train

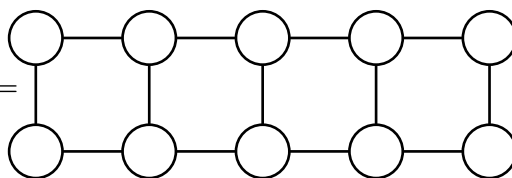
Tensor Train (TT) is a specific tensor network architecture that represents high-dimensional tensors as a chain of lower-rank tensors, providing an efficient compressed representation:



This architecture represents a high-dimensional tensor using a compact one-dimensional chain structure. With bond dimension  $\chi$  (the size of virtual indices connecting adjacent tensors), the storage requirement scales as  $O(d\chi^2L)$ , where  $d$  is the physical dimension and  $L$  is the chain length. This yields a compression ratio of  $O\left(\frac{d^L}{\chi^2L}\right)$  compared to the full tensor.

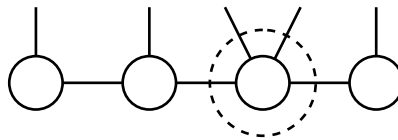
The tensor train format offers several computational advantages:

**1. Efficient inner products.** Computing overlaps between two tensor trains requires only local contractions:

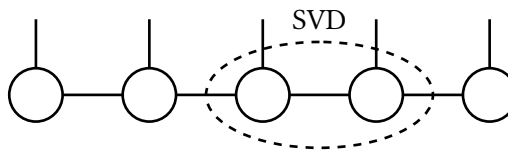
$$\sum_{ijklm} \varphi^*(i, j, k, l, m) \psi(i, j, k, l, m) =$$


**2. Polynomial-time compression.** Unlike many tensor decompositions, tensor trains admit efficient compression through iterative sweeping algorithms that alternately apply:

1. Contract two tensors



2. Tensor factorization



The factorization is usually done by first reshaping the tensor into a matrix and then applying singular value decomposition. By eliminating small singular values, the bond dimension can be reduced. Easy to compress is a feature of all loopless tensor networks, including the tensor train. In the following example, we show a uniform state can be represented as a tensor train of rank 1.

```
julia> uniform_state(n) = fill(sqrt(1/2^n), 2^n);

julia> L, M, R = fill(sqrt(0.5), 2, 1), fill(sqrt(0.5), 1, 2, 1),
fill(sqrt(0.5), 1, 2);

julia> @assert ein"ia,ajb,bkc,cld,dm->ijklm"(L, M, M, M, R) ≈ uniform_state(5)
```

#### Example: Compress a high dimensional tensor with tensor train

In this example, we show how to compress a high dimensional tensor with tensor train. We start from defining the data structure.

```
using OMEinsum, LinearAlgebra
```

```

struct MPS{T}
    tensors::Vector{Array{T, 3}}
end

```

The main algorithm is implemented as follows:

```

# Function to compress a tensor using Tensor Train (TT) decomposition
function tensor_train_decomposition(tensor::AbstractArray,
largest_rank::Int; atol=1e-6)
    dims = size(tensor)
    n = length(dims)
    tensors = Array{Float64, 3}[]
    rpre = 1 # virtual bond dimension size
    current_tensor = reshape(tensor, dims[1], :)
    for i in 1:(n-1)
        # Perform SVD
        U_truncated, S_truncated, V_truncated, r =
truncated_svd(current_tensor, largest_rank, atol)
        push!(tensors, reshape(U_truncated, (rpre, dims[i], r)))

        # Prepare the tensor for the next iteration
        current_tensor = reshape(S_truncated * V_truncated', r *
dims[i+1], :)
        rpre = r
    end
    push!(tensors, reshape(current_tensor, (rpre, dims[n], 1)))
    return MPS(tensors)
end

```

We basically iteratively call the truncated singular value decomposition (SVD) to reduce the virtual bond dimension.

```

function truncated_svd(current_tensor::AbstractArray, largest_rank::Int,
atol)
    U, S, V = svd(current_tensor)
    r = min(largest_rank, sum(S .> atol)) # error estimation
    S_truncated = Diagonal(S[1:r])
    U_truncated = U[:, 1:r]
    V_truncated = V[:, 1:r]
    return U_truncated, S_truncated, V_truncated, r
end

```

To recover the tensor, we construct the matrix product state, we construct the tensor network topology and

```

# Function to contract the TT cores to reconstruct the tensor
function contract(mps::MPS)

```

```

n = length(mps.tensors)
code = EinCode([[2i-1, 2i, 2i+1] for i in 1:n], Int[2i for i in 1:n])
size_dict = OMEinsum.get_size_dict(code.ixs, mps.tensors)
optcode = optimize_code(code, size_dict, GreedyMethod())
return optcode(mps.tensors...)
end

```

As an example, we compress a uniform tensor of size  $2^{20}$ .

```

tensor = ones(Float64, fill(2, 20)...);
mps = tensor_train_decomposition(tensor, 5)
reconstructed_tensor = contract(mps);

relative_error = norm(tensor - reconstructed_tensor) / norm(tensor)
# output: 5.114071183432393e-12

original_size = prod(size(tensor))
compressed_size = sum([prod(size(core)) for core in mps.tensors])
compression_ratio = original_size / compressed_size
# output: 26214.4

```

The virtual bond dimension has size  $\chi = 1$ , which means each tensor has only  $\chi^2 d = 2$  elements.

## Automatic Differentiation

**Backpropagation** constitutes a fundamental machine learning technique for computing gradients of loss functions  $\mathcal{L}$  with respect to model parameters. Its foundation rests on the **backward rule**, which efficiently propagates adjoint information through computational graphs. The adjoint of a variable  $a$  is defined as  $\bar{a} = \frac{\partial \mathcal{L}}{\partial a}$ , representing the sensitivity of the loss to changes in that variable.

For a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  with input  $x$  and known adjoint of the output  $\bar{y}$ , the backward rule computes the adjoint of the input as:

$$\bar{x} = \frac{\partial f^T}{\partial x} \bar{y} \quad (13)$$

This process efficiently propagates gradient information backward through the network, enabling optimization of complex models.

For matrix multiplication  $C = AB$ , the backward rule yields:

$$\bar{A} = \bar{C}B^T, \quad \bar{B} = A^T\bar{C} \quad (14)$$

This rule exemplifies the remarkable efficiency of backpropagation. While the full Jacobian matrix would contain  $O(n^4)$  elements, the backward computation requires only  $O(n^3)$  matrix operations—the same complexity as the forward pass. This efficiency breakthrough enables practical optimization of complex models and underlies the success of modern deep learning. Tensor network contraction generalizes matrix multiplication while preserving differentiation efficiency. We represent a tensor network as the triple  $(\Lambda, \mathcal{T}, \sigma_Y)$ :

$$Y = \text{contract}(\Lambda, \mathcal{T}, \sigma_Y) \quad (15)$$



where  $\Lambda$  contains all tensor indices,  $\mathcal{T}$  holds the tensor collection, and  $\sigma_Y \subset \Lambda$  specifies output indices.

The backward rule for computing input tensor gradients follows naturally:

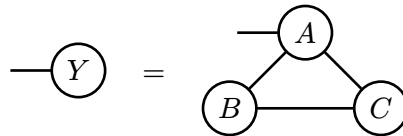
$$\bar{X} = \text{contract}(\Lambda, (\mathcal{T} \setminus \{X\}) \cup \{\bar{Y}\}, \sigma_X) \quad (16)$$

where  $\mathcal{T} \setminus \{X\}$  represents the tensor set excluding  $X$ , and  $\sigma_X$  denotes  $X$ 's indices.

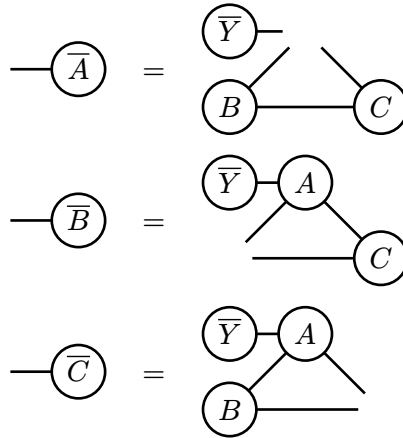
While naive implementation would require separate network contractions for each input (linear overhead), sophisticated binary contraction trees reduce this to constant overhead. Modern automatic differentiation achieves gradient computation at approximately twice the forward pass cost—remarkable efficiency considering the inherent complexity of multilinear operations.

#### Example: Backward rule for tensor network contraction

Consider the tensor network contraction:  $Y = \text{ein}''\text{a}ij,jk,ki\text{->a}''(A, B, C)$ , where  $A, B, C$  are tensors labeled by  $(a, i, j), (j, k), (k, i)$  respectively. Diagrammatically, the forward contraction is given by:



The backward rule is given by:



**Quiz:** If the forward contraction specified with a binary contraction order:  $Y = \text{ein}''(aij,jk),ki\text{->a}''(A, B, C)$ , how are gradients computed in the backward propagation?

In `OMEinsum`, the backward rule of `einsum` has already been ported to `ChainRulesCore`, which can be directly used in `Zygote` and `Flux`. It also implements a

```
julia> gradients = cost_and_gradient(optcode, (tensors...,));
```

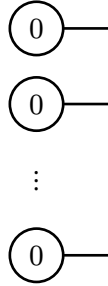
The returned `gradients` is a vector of arrays, each of which is an adjoint of an input tensor.

# Quantum Circuit Simulation

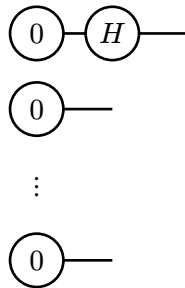
## Quantum states and quantum gates

Quantum circuits provide a natural framework for tensor network representations, where quantum states become vectors and quantum gates become tensors.

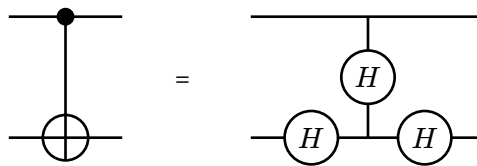
A quantum system initialized to  $|0\rangle^{\otimes n}$  (the  $n$ -fold tensor product of  $|0\rangle$  states) decomposes as a direct product of  $n$  individual qubits:



where each  $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  state is represented as a rank-1 tensor. Single-qubit gates correspond to rank-2 tensors (matrices) that transform individual qubits. For instance, applying a Hadamard gate  $H$  to the first qubit creates the following tensor network:



Multi-qubit gates create more complex tensor network structures. The CNOT gate, fundamental to quantum computation, can be decomposed into a tensor network representation:

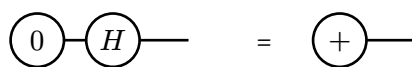


This decomposition (ignoring normalization factors) illustrates how two-qubit gates introduce entanglement through shared virtual indices connecting different physical qubits.

## Useful circuit identities

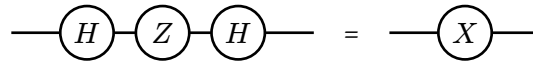
Tensor network representations make certain quantum circuit identities immediately apparent through graphical manipulation. Several fundamental rules simplify complex circuits:

### Identity 1: Hadamard on computational basis state



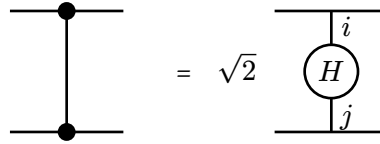
This transforms  $|0\rangle$  into the  $|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$  state.

### Identity 2: Hadamard conjugation of Pauli gates



The Hadamard gate transforms Pauli-Z into Pauli-X:  $HZH = X$ . This basis transformation is fundamental to many quantum algorithms.

### Identity 3: Controlled-Z

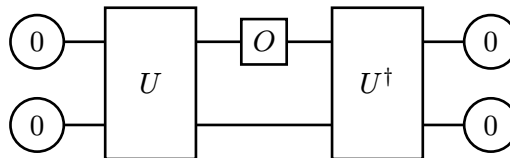


A controlled-Z gate (CZ) can be implemented using a single tensor connecting both qubits, demonstrating how entangling operations create shared virtual bonds in the tensor network. If you do not believe it, we can easily verify this equality with OMEinsum:

```
julia> reshape(ein"ij->ijij"([1 1; 1 -1]), 4, 4)
4×4 Matrix{Int64}:
 1  0  0  0
 0  1  0  0
 0  0  1  0
 0  0  0 -1
```

### Expectation values

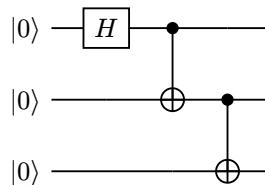
Computing expectation values of observables in quantum circuits translates to a specific tensor network contraction pattern. For a quantum state  $|\psi\rangle = U|0^n\rangle$  and observable  $O$ , the expectation value  $\langle\psi|O|\psi\rangle$  has the tensor network representation:



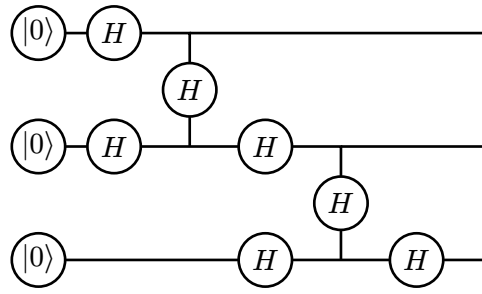
This “sandwich” structure represents the quantum expectation value formula  $\langle 0^n | U^\dagger O U | 0^n \rangle$ , where the observable  $O$  is inserted between the forward circuit  $U$  and its conjugate  $U^\dagger$ .

### Example: GHZ state preparation circuit

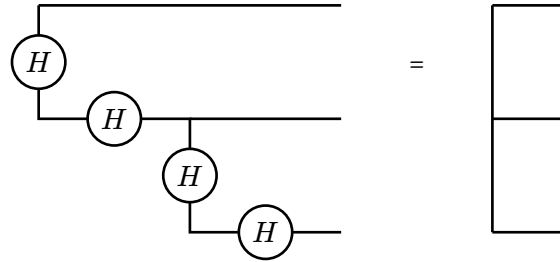
Consider a 3-qubit quantum circuit that prepares the GHZ state  $|\text{GHZ}\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ . The quantum circuit generating this state is shown below:



The corresponding tensor network diagram is:



which can be simplified to



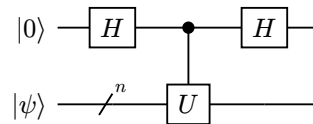
Question: How to compute  $\langle \text{GHZ} | O | \text{GHZ} \rangle$  and what is the complexity?

### Example: Hadamard test

The Hadamard test is a quantum algorithm used to estimate the expectation value of a unitary operator  $U$  with respect to a quantum state  $|\psi\rangle$ . It provides a way to measure  $\langle \psi | U | \psi \rangle$  using an ancilla qubit.

#### Hadamard test circuit

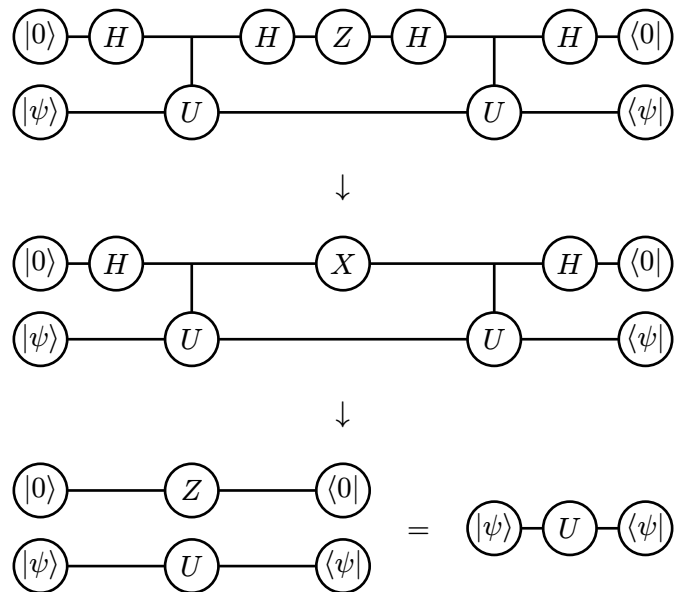
The Hadamard test circuit is shown below:



The expectation value of  $Z$  on the first qubit is given by

$$\langle Z \rangle = \text{Re}(\langle \psi | U | \psi \rangle) \quad (17)$$

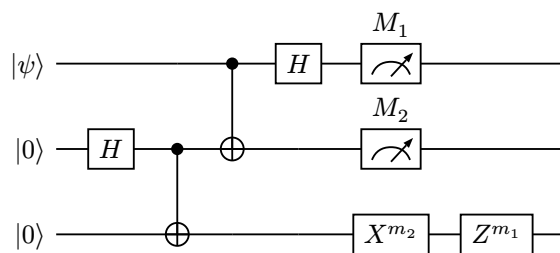
The corresponding tensor network representation is:



### Example: Quantum teleportation

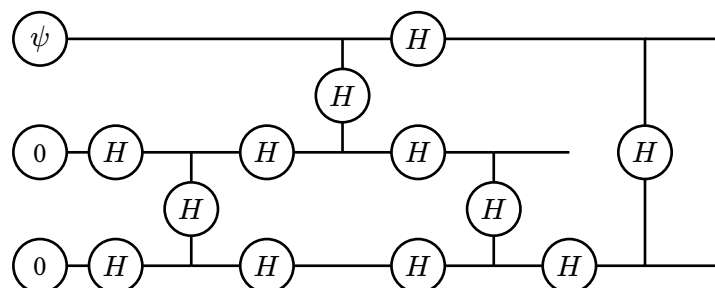
Teleportation transmits an unknown state  $|\psi\rangle$  from Alice to Bob using a shared Bell pair and two classical bits. The steps are: (1) prepare a Bell pair on qubits 2–3, (2) perform a Bell-basis measurement on qubits 1–2, (3) apply Pauli corrections  $Z^{m_1} X^{m_2}$  on qubit 3 according to outcomes  $(m_1, m_2)$ .

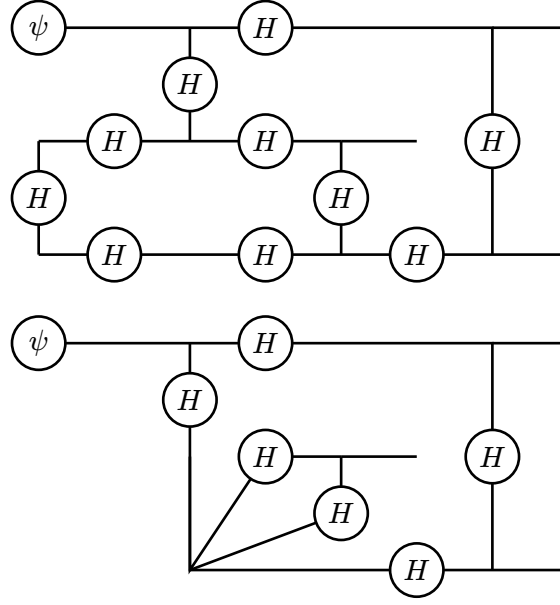
#### Circuit



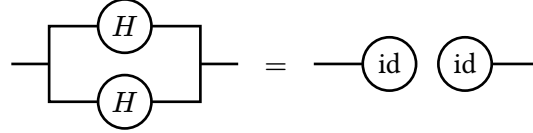
#### Tensor-network diagram and simplification

The circuit maps to a tensor network where the Bell pair is a rank-2 tensor, gates are rank-4 tensors, and measurements are projectors. Up to Pauli frame corrections, the network reduces to an identity wire from Alice's input to Bob's output.

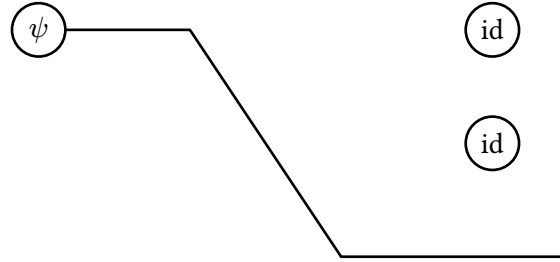




Here, we use the following identity:



Then we have



## Quantum channel simulation

Quantum channels represent the evolution of open quantum systems, capturing both unitary evolution and decoherence effects. In tensor network simulations, these are implemented through the Kraus operator formalism and density matrix evolution.

### Kraus operators

A quantum channel  $\mathcal{E}$  can be represented using Kraus operators  $\{K_i\}$  such that for any density matrix  $\rho$ :

$$\mathcal{E}(\rho) = \sum_i K_i \rho K_i^\dagger \quad (18)$$

where the Kraus operators satisfy the completeness relation:

$$\sum_i K_i^\dagger K_i = I \quad (19)$$

Kraus operators are a **completely positive (CP) and trace preserving (TP) map** on the density matrix space, which is a linear map that preserves the positivity and the probability of the density matrix.

This formalism allows us to describe various noise processes:

### Amplitude damping

Models energy loss processes with Kraus operators:

$$K_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \quad K_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix} \quad (20)$$

### Phase damping

Models pure dephasing with:

$$K_0 = \sqrt{1 - \frac{\gamma}{2}} I, \quad K_1 = \sqrt{\frac{\gamma}{2}} Z \quad (21)$$

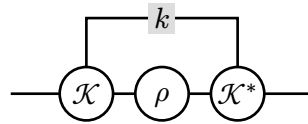
### Depolarizing channel

The most commonly used noise model, with Kraus operators:

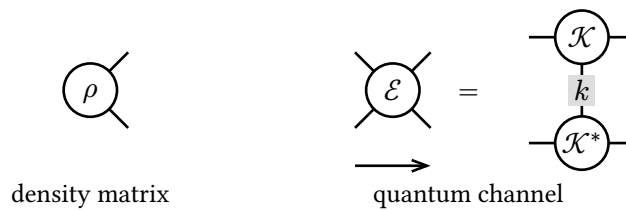
$$K_0 = \sqrt{1 - 3\frac{p}{4}} I, \quad K_1 = \sqrt{\frac{p}{4}} X, \quad K_2 = \sqrt{\frac{p}{4}} Y, \quad K_3 = \sqrt{\frac{p}{4}} Z \quad (22)$$

## Tensor network representation of channels

Consider applying a Kraus channel  $\mathcal{E}$  to a density matrix  $\rho$ . The result can be diagrammatically represented as



Sometimes, we use the superoperator representation, which corresponds to the contracted Kraus channels



For example, the superoperator representation of the depolarizing channel is

```
julia> using OMEinsum, Yao, SymEngine

julia> p = Basic(:p) # define a symbolic variables
p

julia> K = cat(sqrt(1-3p/4) * Matrix{Basic}(I2), sqrt(p/4) * Matrix{Basic}(X),
sqrt(p/4) * Matrix{Basic}(Y), sqrt(p/4) * Matrix{Basic}(Z); dims=3);
```

```
julia> superop_dep = reshape(ein"abk,cdk->acbd"(K, conj(K)), 4, 4)
4×4 Matrix{Basic}:
 1 + (-1/2)*p      0      0      (1/2)*p
      0 1 - p      0      0
      0      0 1 - p      0
 (1/2)*p      0      0 1 + (-1/2)*p
```

### Pauli transfer matrix formulation

The PTM formalism provides a powerful framework for classical simulation of noisy quantum circuits. In this representation, the normalized Pauli basis  $\mathbb{P} = \frac{\{I, X, Y, Z\}}{\sqrt{2}}$  forms an orthonormal basis for the operator space, where single-qubit quantum states become vectors  $|\rho\rangle\rangle_P$  with components:

$$(|\rho\rangle\rangle_P)_i = \text{tr}(\rho P_i), \quad P_i \in \mathbb{P} \quad (23)$$

Let us denote the superoperator (vectorized) representation of density matrix  $\rho$  as  $|\rho\rangle\rangle$ . The Pauli basis representation corresponds to the following basis transformation:

$$|\rho\rangle\rangle_P = U|\rho\rangle\rangle$$

$$U = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{-i}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{i}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & \frac{-1}{\sqrt{2}} \end{pmatrix} \quad (24)$$

The four columns correspond to the vectorized and normalized Pauli matrices. Then, we also apply this basis transformation to the quantum channel  $\mathcal{E}$ :

$$\mathcal{E}_P = U\mathcal{E}U^\dagger \quad (25)$$

Diagrammatically, this transformation is



For the depolarizing channel, the Pauli basis representation can be obtained by:

```
julia> U = Basic[1 0 0 1; 0 1 -im 0; 0 1 im 0; 1 0 0 -1] / sqrt(Basic(2));

julia> pauli_dep = SymEngine.expand.(U * superop_dep * U')
4×4 Matrix{Basic}:
 1      0      0      0
 0 1 - p      0      0
 0      0 1 - p      0
 0      0      0 1 - p
```

It is a diagonal matrix  $\mathcal{D}_P = \text{diag}(1, 1 - p, 1 - p, 1 - p)$ , enabling efficient multi-qubit simulation via tensor decomposition:

$$\mathcal{D}_P = (1 - p)I + p|0\rangle\rangle\langle\langle 0| \quad (26)$$



Or diagrammatically,

$$\text{Diagram of } \mathcal{D}_P = (1-p) \text{Diagram of } \text{---} + p \text{Diagram of } \begin{matrix} \text{---} \text{---} \\ \text{---} \text{---} \end{matrix}$$

In the path-integral point of view, we either pick the first term or the second term in a single path. The first term has the power of damping the amplitude of states, while the second term has rank 1, and can be used to truncate the tensor network. As a consequence, quantum circuits with finite depolarizing noise can be simulated in polynomial time[12], [13].

## Quantum Error Correction

Quantum error correction(QEC) is a process of protecting quantum information from errors[14], [15], [16]. The errors can be caused by the environment, the control system, or the quantum gates. The quantum error correction is a process of encoding the quantum information into a larger Hilbert space such that the quantum information can be recovered from the errors. Usually, a quantum error correction scheme is described by a stabilizer group.

### Stablizers and Quantum Codes

In this section, we will introduce the concept of stabilizers and quantum codes.

**Definition 3** (Pauli Group and Stabilizer Group): A *stabilizer group*  $\mathcal{S}$  is an Abelian subgroup of the  $n$ -qubit Pauli group. The  $n$ -qubit pauli group is the group generated by the  $n$ -qubit Pauli matrices[17]

$$\mathcal{P}_n = (\pm i)\{I, X, Y, Z\}^{\otimes n} \quad (27)$$

We usually call the elements of the Pauli group pauli operators or pauli strings.

The stabilizer group is abelian means that it is commutative and the measurement outcome of any two stabilizers will not affect each other. We can specify a stabilizer group by giving a set of independent stabilizer generators  $\{S_a\}_{a=1,\dots,m}$

$$\mathcal{S} = \langle S_1, S_2, \dots, S_m \rangle. \quad (28)$$

The code space is the  $+1$  eigenspace of all the stabilizers. We can detect whether a state is in the code space by only measuring the generators of the stabilizer group. If any of the generators gives  $-1$ , then the state is not in the code space. And we call such an outcome a syndrome. It is worth mentioned that this measurement will not cause the quantum computing collapse, since we only measure the stabilizers and the final state is in a subspace of the original Hilbert space, which is the code space or an error space. The entanglements in the code space are not destroyed.

**Definition 4** (Quantum Code): An  $[[n, k, d]]$  quantum code is a quantum error correction scheme that encodes a  $k$ -qubit subspace of an  $n$ -qubit Hilbert space with minimum distance  $d$ . The minimum distance is the minimum pauli operators that need to be applied on one code word to get to another code word.

Usually, we can specify an  $[[n, k, d]]$  quantum code by giving a set of  $n - k$  independent stabilizer generators.

**Definition 5** (CSS Code[16], [18], [19]): We call a quantum stabilizer code a Calderbank-Shor-Steane (CSS) code if the stabilizer group can be generated by pauli matrices that only contain  $X$  or  $Z$  operators, i.e.,

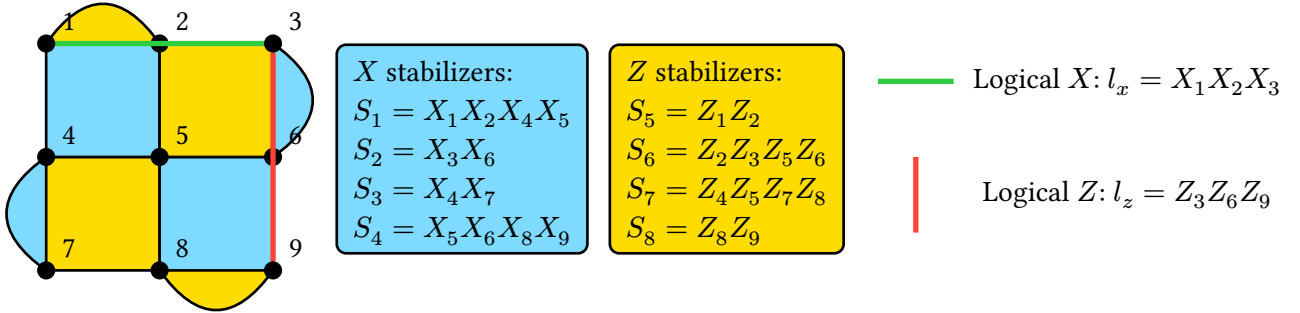
$$\mathcal{S} = \langle S_a \rangle_{a=1, \dots, n-k}, \text{ where } S_a \in \{I, X\}^{\otimes n} \cup \{I, Z\}^{\otimes n} \quad (29)$$

In short, a CSS code is a type of quantum code that can be constructed using only  $X$  and  $Z$  operators. Moreover, most quantum codes encountered in practice belong to the CSS family.

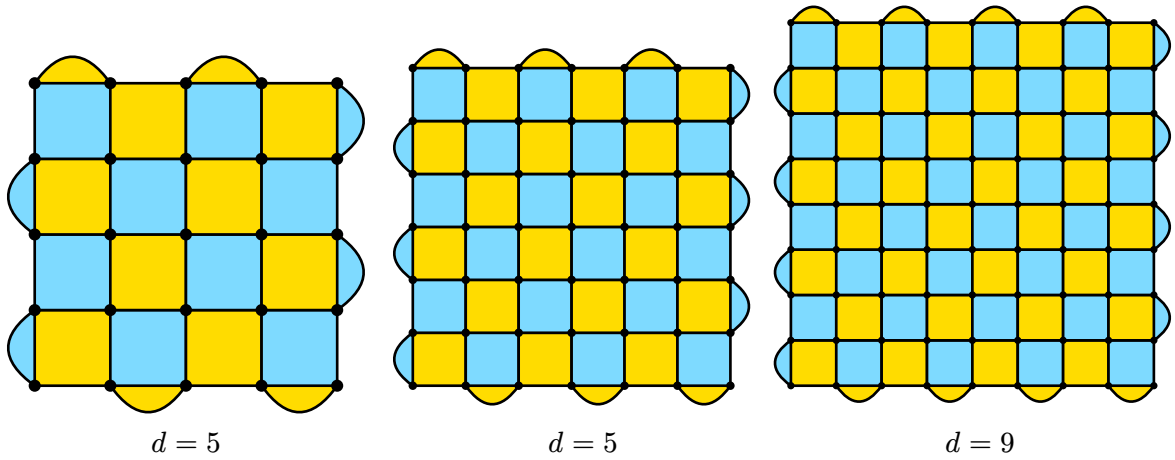
## Surface code

The surface code[20], [21] is a prominent example of a topological quantum error-correcting code, defined on a two-dimensional lattice of qubits arranged in a grid. Each plaquette (face) of the lattice is associated with a stabilizer operator, which acts on the qubits at the corners of the plaquette.

There are two types of stabilizers:  $X$ -type (acting with Pauli  $X$  operators) and  $Z$ -type (acting with Pauli  $Z$  operators), typically arranged in a checkerboard pattern. Here is an example of  $[[9, 1, 3]]$  surface code. The stabilizers are shown in the figure. The logical operator  $X_1 X_2 X_3$  and  $Z_3 Z_6 Z_9$  commute with all stabilizers and do not belong to the stabilizer group. The length of them is exactly the distance of the code.



Also we can have different sizes of the surface code.



## Decoding problem

If some Pauli errors happened, some of the stabilizers will be anti-commute with the errors. When we measure them We usually call the measurement outcome of the all stabilizers as syndrome. The

decoding problem is given the syndrome, find the probable error pattern that is consistent with the syndrome.

**Definition 6** (MLE Problem): The most-likely error(MLE) problem is given the syndrome, find the most probable error pattern that is consistent with the syndrome.

$$\begin{aligned} \operatorname{argmax}_e p(e) \\ \text{s.t. } H(e) = s \end{aligned} \quad (30)$$

where  $H(e)$  is the syndrome of the error pattern  $e$ .

For a given error, applying any stabilizer to it leaves the syndrome unchanged. All such errors within the same degenerate class have an equivalent effect on the logical information. Thus, a better decoding approach than MLE is MLD (Maximum Likelihood Decoding), which directly determines the most probable logical effect of the error rather than the exact physical error.

**Definition 7** (MLD Problem): The maximum likelihood decoding(MLD) problem is given the syndrome, find the most probable logical state by summing over all the error patterns that belong to the same degenerate class and are consistent with the syndrome.

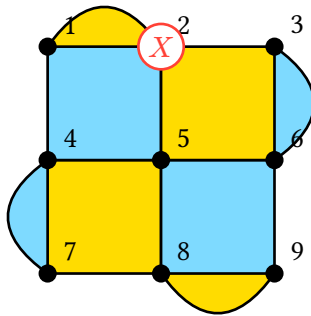
$$\operatorname{argmax}_l p(l) = \operatorname{argmax}_l \sum_{\substack{L(e)=l \\ H(e)=s}} p(e) \quad (31)$$

where  $H(e)$  is the syndrome of the error pattern  $e$ ,  $L(e)$  is the logical information of the error pattern  $e$ .

As shown, the MLD decoding problem involves summing over all error patterns within the same degenerate class that match the observed syndrome. This structure naturally lends itself to tensor network contraction methods. In the following section, we will introduce tensor network-based MLD decoder. Now we will give an example of the decoding problem.

### Example: Decoding problem

Suppose there is an  $X$  error on the qubit 2. The decoding process is to find the most probable error pattern that is consistent with the syndrome.



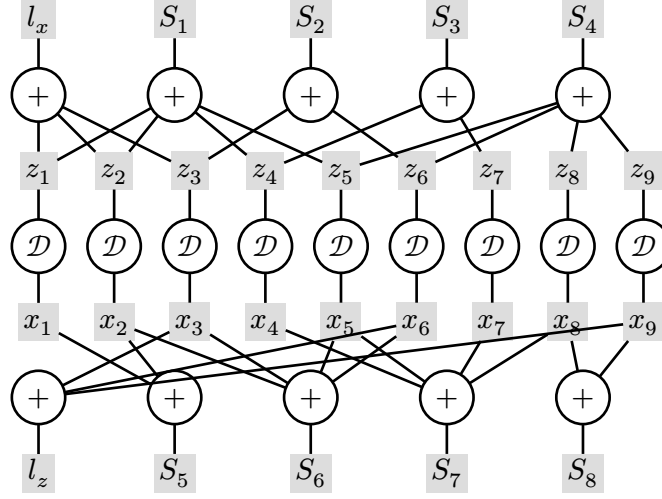
$X$  stabilizers:  
 $X_1 X_2 X_4 X_5$   
 $X_3 X_6$   
 $X_4 X_7$   
 $X_5 X_6 X_8 X_9$

$Z$  stabilizers:  
 $Z_1 Z_2$   
 $Z_2 Z_3 Z_5 Z_6$   
 $Z_4 Z_5 Z_7 Z_8$   
 $Z_8 Z_9$

Only stabilizer  $Z_1 Z_2$  and  $Z_2 Z_3 Z_5 Z_6$  is anti-commute with the error. So if we measure all the stabilizers, we will get six +1 and two -1. Base on this syndrome, decoders will try to find the most probable error pattern or logical state.

## Tensor network decoder

Here we directly give the tensor network representation[5], [22] of the decoding problem. The dimension of the variables is 2.



In the middle of the figure, we have 9 tensors represent the depolarizing channel acts on the physical qubits.

$$\mathcal{D} = \begin{pmatrix} p_I & p_Z \\ p_X & p_Y \end{pmatrix}$$

Depolarizing Channel:

$$\mathcal{D}(\rho) = (1 - p_X - p_Y - p_Z)\rho + p_X X\rho X + p_Y Y\rho Y + p_Z Z\rho Z$$

The variables connected to the depolarizing channel represent Boolean variables indicating  $X$  or  $Z$  errors on the physical qubits. The  $+$  tensors are the parity tensors.

$$\begin{array}{c} j_1 \\ | \\ \bigcirc + \\ / \quad \backslash \\ j_k \quad \dots \quad j_3 \end{array} : T(j_1, j_2, j_3, \dots, j_k) = \begin{cases} 1 & \text{if } j_1 + j_2 + \dots + j_k \text{ is even} \\ 0 & \text{if } j_1 + j_2 + \dots + j_k \text{ is odd} \end{cases}$$

$$j_1, j_2, j_3, \dots, j_k \in \{0, 1\}$$

The stabilizer variables are fixed to the measured syndrome values, while the logical variables represent the marginal probabilities computed via tensor network contraction. After contracting such a tensor network, we can get the marginal probability of the logical variables.

## Bibliography

- [1] C. R. Harris *et al.*, “Array Programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).

- [2] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, vol. 4, no. 4. Springer, 2006.
- [3] M. R. Villescas, J.-G. Liu, P. W. Wijnings, S. Stuijk, and H. Corporaal, “Scaling Probabilistic Inference Through Message Contraction Optimization,” in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, Jul. 2023, pp. 123–130. doi: [10.1109/CSCE60160.2023.00025](https://doi.org/10.1109/CSCE60160.2023.00025).
- [4] I. L. Markov and Y. Shi, “Simulating Quantum Computation by Contracting Tensor Networks,” *SIAM Journal on Computing*, vol. 38, no. 3, pp. 963–981, Jan. 2008, doi: [10.1137/050644756](https://doi.org/10.1137/050644756).
- [5] C. Piveteau, C. T. Chubb, and J. M. Renes, “Tensor-Network Decoding Beyond 2D,” *PRX Quantum*, vol. 5, no. 4, p. 40303, Oct. 2024, doi: [10.1103/PRXQuantum.5.040303](https://doi.org/10.1103/PRXQuantum.5.040303).
- [6] Y. Qing, K. Li, P.-F. Zhou, and S.-J. Ran, “Compressing Neural Network by Tensor Network with Exponentially Fewer Variational Parameters,” no. arXiv:2305.06058. arXiv, May 2024. doi: [10.48550/arXiv.2305.06058](https://doi.org/10.48550/arXiv.2305.06058).
- [7] J. Haegeman, C. Lubich, I. Oseledets, B. Vandereycken, and F. Verstraete, “Unifying Time Evolution and Optimization with Matrix Product States,” *Physical Review B*, 2016, doi: [10.1103/PhysRevB.94.165116](https://doi.org/10.1103/PhysRevB.94.165116).
- [8] J. Gray and S. Kourtis, “Hyper-optimized tensor network contraction,” *Quantum*, vol. 5, p. 410, Mar. 2021, doi: [10.22331/q-2021-03-15-410](https://doi.org/10.22331/q-2021-03-15-410).
- [9] M. Roa-Villescas, X. Gao, S. Stuijk, H. Corporaal, and J.-G. Liu, “Probabilistic Inference in the Era of Tensor Networks and Differential Programming,” *Physical Review Research*, vol. 6, no. 3, p. 33261, Sep. 2024, doi: [10.1103/PhysRevResearch.6.033261](https://doi.org/10.1103/PhysRevResearch.6.033261).
- [10] F. V. Fomin and K. Høie, “Pathwidth of Cubic Graphs and Exact Algorithms,” *Information Processing Letters*, vol. 97, no. 5, pp. 191–196, 2006, doi: [10.1016/j.ipl.2005.10.012](https://doi.org/10.1016/j.ipl.2005.10.012).
- [11] G. Kalachev, P. Panteleev, and M.-H. Yung, “Recursive Multi-Tensor Contraction for XEB Verification of Quantum Circuits.” 2021.
- [12] X. Gao and L. Duan, “Efficient Classical Simulation of Noisy Quantum Computation,” no. arXiv:1810.03176. arXiv, Oct. 2018. doi: [10.48550/arXiv.1810.03176](https://doi.org/10.48550/arXiv.1810.03176).
- [13] E. Fontana, M. S. Rudolph, R. Duncan, I. Rungger, and C. Cîrstoiu, “Classical Simulations of Noisy Variational Quantum Circuits,” no. arXiv:2306.05400. arXiv, Jun. 2023. doi: [10.48550/arXiv.2306.05400](https://doi.org/10.48550/arXiv.2306.05400).
- [14] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.
- [15] D. Gottesman, *Stabilizer codes and quantum error correction*. California Institute of Technology, 1997.
- [16] A. R. Calderbank and P. W. Shor, “Good quantum error-correcting codes exist,” *Physical Review A*, vol. 54, no. 2, p. 1098, 1996.
- [17] F. Gaitan, *Quantum error correction and fault tolerant quantum computing*. CRC Press, 2008.
- [18] A. M. Steane, “Error correcting codes in quantum theory,” *Physical Review Letters*, vol. 77, no. 5, p. 793, 1996.
- [19] A. Steane, “Multiple-particle interference and quantum error correction,” *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 452, no. 1954, pp. 2551–2577, 1996.

- [20] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, “Topological quantum memory,” *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, 2002.
- [21] A. Y. Kitaev, “Fault-tolerant quantum computation by anyons,” *Annals of physics*, vol. 303, no. 1, pp. 2–30, 2003.
- [22] C. T. Chubb, “General tensor network decoding of 2D Pauli codes,” *arXiv preprint arXiv:2101.04125*, 2021.