Quantum circuit simulation with tensor network contraction

Link to the tutorial repository: https://github.com/GiggleLiu/tutorial-tensornetwork

```julia
# check the current environment
using Pkg; Pkg.activate("../..")  ; Pkg.status()
```

```
  Activating project at `~/jcode/tutorial-tensornetwork`
Status `~/jcode/tutorial-tensornetwork/Project.toml`
  [6e4b80f9] BenchmarkTools v1.6.0
  [1f49bdf2] LuxorGraphPlot v0.5.1
  [ebe7aa44] OMEinsum v0.9.2
  [c3e4b0f8] Pluto v0.20.16
  [7f904dfe] PlutoUI v0.7.70
  [123dc426] SymEngine v0.12.0
  [0500ac79] TensorQEC v2.2.0
  [5872b779] Yao v0.9.2
  [9b173c7b] YaoToEinsum v0.2.8 `~/.julia/dev/Yao/lib/YaoToEinsum`
  [37e2e46d] LinearAlgebra v1.11.0
  [9a3f8284] Random v1.11.0
```

```julia
# `PlutoUI` is for control gadgets, e.g. the checkboxes
using PlutoUI
```

Table of Contents


```julia
PlutoUI.TableOfContents(aside=false)
```

Tutorial: einsum notation

In this tutorial, we use OMEinsum.jl as our default tensor network contractor.

- State of the art performance in optimizing the contraction order
- Has GPU support

```julia
using OMEinsum   # Tensor network contraction backend
```

Specify a tensor network with string literal `ein`

```
code = ab, bc, cd -> ad
```

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

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

or programmatically

```
1∘2, 2∘3, 3∘4 -> 1∘4
```

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

```
▾Vector{Char}[
    1:  ▾Char[
            1:  'a'
            2:  'b'
        ]
    2:  ▸['b', 'c']
    3:  ▸['c', 'd']
]
```

```julia
getixsv(code)    # indices for input tensors
```

```
▸['a', 'd']
```

```julia
getiyv(code)     # indices for the output tensor
```

variable_dimension = 100

```
label_sizes = ▾Dict{Char, Int64}(
                  'a' ⇒ 100
                  'c' ⇒ 100
                  'd' ⇒ 100
                  'b' ⇒ 100
              )
```

```julia
label_sizes = uniformsize(code, variable_dimension)  # define the sizes of the indices
```

```
Time complexity: 2^26.575424759098897
Space complexity: 2^13.287712379549449
Read-write complexity: 2^15.287712379549449
```

```julia
# Time complexity: number of arithematic operations
# Space complexity: number of elements in the largest tensor
# Read-write complexity: number of elemental read-write operations
contraction_complexity(code, label_sizes)
```

```
2×2 Matrix{Float64}:
 -1.5785    3.0381
  1.84091  -3.8551
```

```julia
code(randn(2, 2), randn(2, 2), randn(2, 2))  # not recommended
```

```
nested_code = ac, cd -> ad
              ├─ ab, bc -> ac
              │  ├─ ab
              │  └─ bc
              └─ cd
```

```julia
nested_code = ein"(ab,bc),cd->ad"  # recommended
```

```
Time complexity: 2^20.931568569324174
Space complexity: 2^13.287712379549449
Read-write complexity: 2^15.872674880270605
```

```julia
contraction_complexity(nested_code, label_sizes)
```

```julia
using BenchmarkTools  # use for benchmark
```

run_benchmark = ☑

```julia
run_benchmark && @btime code(randn(100, 100), randn(100, 100), randn(100, 100)); # unoptimized
```

```
85.595 ms (38 allocations: 385.55 KiB)
```

```julia
run_benchmark && @btime nested_code(randn(100, 100), randn(100, 100), randn(100, 100)); # optimized
```

```
146.875 μs (165 allocations: 486.31 KiB)
```

Reasons why order matters:

1. Contraction order reduces the computational complexity
2. Binary contraction can make use of BLAS

Contraction order optimization

- Contracting a tensor network is #P-hard, the complexity is $O(2^{\mathrm{tw}(\overline{T})})$, i.e. exponential to the tree width of the line graph of the tensor network hypergraph topology T.
- Optimizing the contraction order is NP-hard

```julia
using Graphs  # used for constructing graphs
```

```
demo_network (generic function with 1 method)
```

```julia
function demo_network(n::Int; seed=2)
    # random regular graph
    g = random_regular_graph(n, 3; seed)
    # place a matrix on each edge
    code = EinCode([[e.src, e.dst] for e in edges(g)], Int[])
    # each input matrix has size 2x2
    sizes = uniformsize(code, 2)
    tensors = [randn([sizes[leg] for leg in ix]...) for ix in getixsv(code)]
    return code, tensors, sizes
end
```

```julia
code_r3, tensors_r3, sizes_r3 = demo_network(100);
```

```julia
optcode = optimize_code(
    code_r3,    # tensor network topology
    sizes_r3,   # variable sizes
    TreeSA()    # optimizer
);
```

```
cc_r3 = Time complexity: 2^17.347033043146006
        Space complexity: 2^13.0
        Read-write complexity: 2^16.52724790138619
```

```julia
cc_r3 = contraction_complexity(optcode, sizes_r3)
```

For more choices of optimizers, please check: OMEinsumContractionOrdersBenchmark and issue

```julia
# reduce the memory cost by slicing
sliced_code = slice_code(
    optcode,
    sizes_r3,
    TreeSASlicer(score=ScoreFunction(sc_target=cc_r3.sc-3))  # keep slicing until the space complexity target `sc_target` is reached.
);
```

```
cc_r3_sliced = Time complexity: 2^17.675295499094062
               Space complexity: 2^10.0
               Read-write complexity: 2^17.126623819032357
```

```julia
cc_r3_sliced = contraction_complexity(sliced_code, sizes_r3)
```

Example 1: GHZ state generation circuit

We use Yao.jl as our default quantum simulation tool.

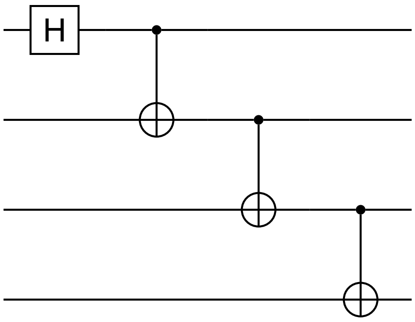
- State of the art performance, has GPU support
- Supports tensor network backend
- Supports noisy channel simulation

```julia
using Yao   # Quantum circuit simulator
```

Let us first define a GHZ state generation circuit.

```
ghz_circuit (generic function with 1 method)
```

```julia
# chain: connect the component gates
# put(n, k=>G): place gate G at location k of a n qubits system.
# control(n, c, k=>G): place controlled gate G at location k, c is the control qubit
ghz_circuit(n) = chain(put(n, 1=>H), [control(n, i-1, i=>X) for i=2:n]...)
```




```
vizcircuit(ghz_circuit(4))
```

```
net_ghz = TensorNetwork
          Time complexity: 2^5.807354922057604
          Space complexity: 2^4.0
          Read-write complexity: 2^7.044394119358453
```

```
# 1st argument is a Yao circuit
# 'initial_state' takes a dictionary
# 'final_state' is left unspecified
net_ghz = yao2einsum(ghz_circuit(4);
        initial_state= Dict(zip(1:4, zeros(Int,4))),
        optimizer = TreeSA(ntrials=1)  # contraction order optimizer
    )
```

The tensor network contraction is represented as a binary tree. It contains both the tensor network topology and an optimized contraction order.

```
6∘5, 8∘6∘7 -> 5∘6∘7∘8
├─ 6∘5∘2, 5∘1 -> 6∘5
│  ├─ 2, 6∘2∘5 -> 6∘5∘2
│  │  ├─ 2
│  │  └─ 6∘2∘5
│  └─ 1, 5∘1 -> 5∘1
│     ├─ 1
│     └─ 5∘1
└─ 8∘7∘4, 7∘6∘3 -> 8∘6∘7
   ├─ 4, 8∘4∘7 -> 8∘7∘4
   │  ├─ 4
   │  └─ 8∘4∘7
   └─ 3, 7∘3∘6 -> 7∘6∘3
      ├─ 3
      └─ 7∘3∘6
```

```
net_ghz.code  # contraction code in (nested) einsum notation
```

```
(
    1: :code
    2: :tensors
    3: :label_to_qubit
)
```

```
fieldnames(typeof(net_ghz))
```

```
[[3], [1], [4], [2], [5, 1], [6, 2, 5], [7, 3, 6], [8, 4, 7]]
```

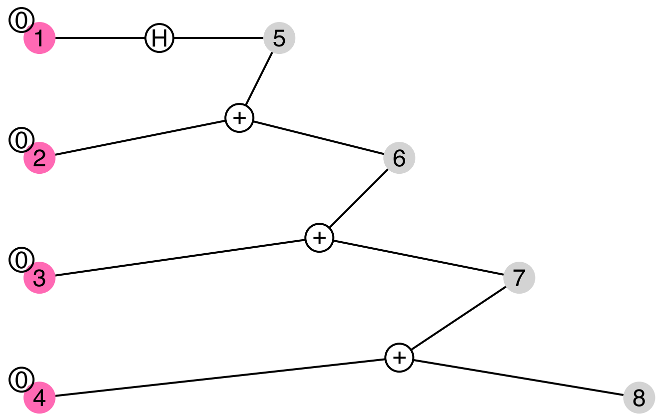
```
OMEinsum.getixsv(net_ghz.code)  # input tensor labels
```

8

```
length(net_ghz.tensors)   # input tensor data
```

```
▸[5, 6, 7, 8]
```

```
OMEinsum.getiyv(net_ghz.code)  # open indices
```




```
# red/gray nodes are variables/open variables, transparent nodes are tensors
# 0 tensor is defined as: [1, 0]
# + tensor is the XOR gate
viznet(net_ghz; scale=60)
```

```
Time complexity: 2^5.807354922057604
Space complexity: 2^4.0
Read-write complexity: 2^7.044394119358453
```

```
contraction_complexity(net_ghz)
```

```
2×2×2×2 Array{ComplexF64, 4}:
[:, :, 1, 1] =
 0.707107+0.0im  0.0+0.0im
      0.0+0.0im  0.0+0.0im

[:, :, 2, 1] =
 0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im

[:, :, 1, 2] =
 0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im

[:, :, 2, 2] =
 0.0+0.0im       0.0+0.0im
 0.0+0.0im  0.707107+0.0im
```

```
Yao.contract(net_ghz)
```

Example 2: Simulate quantum supremacy experiments

In this example, we will load the quantum supremacy circuit from the disk, and compute probability of having state $|0\rangle$ by computing $\langle 0|U|0\rangle$, where U is the quantum circuit of interest.

Step 1: circuit loading

Some popular shallow quantum circuits are placed in the `data` folder, they are from qfelx (Ref. qflex datasets, check bottom). To load the circuits to Yao, please use the `YaoCircuitReader` module provided in file `reader.jl`:

```julia
# circuit reader
include("reader.jl"); using .YaoCircuitReader: yaocircuit_from_file
```

```
String[
    1: "bristlecone_48_1-16-1_0.txt"
    2: "bristlecone_48_1-20-1_0.txt"
    3: "bristlecone_48_1-24-1_0.txt"
    4: "bristlecone_48_1-32-1_0.txt"
    5: "bristlecone_48_1-40-1_0.txt"
    6: "bristlecone_60_1-24-1_0.txt"
    7: "bristlecone_60_1-32-1_0.txt"
    8: "bristlecone_60_1-40-1_0.txt"
    9: "bristlecone_70_1-12-1_0.txt"
    10: "bristlecone_70_1-16-1_0.txt"
    11: "bristlecone_70_1-20-1_0.txt"
    12: "bristlecone_70_1-24-1_0.txt"
    13: "bristlecone_70_1-32-1_0.txt"
    14: "bristlecone_70_1-40-1_0.txt"
    15: "rectangular_11x12_1-16-1_0.txt"
    16: "rectangular_11x12_1-24-1_0.txt"
    17: "rectangular_11x12_1-32-1_0.txt"
    18: "rectangular_11x12_1-40-1_0.txt"
    19: "rectangular_2x2_1-2-1_0.txt"
    20: "rectangular_4x4_1-16-1_0.txt"
    21: "rectangular_6x6_1-16-1_0.txt"
    22: "rectangular_6x6_1-24-1_0.txt"
    23: "rectangular_6x6_1-32-1_0.txt"
    24: "rectangular_7x7_1-32-1_0.txt"
    25: "rectangular_7x7_1-40-1_0.txt"
    26: "rectangular_7x7_1-48-1_0.txt"
    27: "rectangular_8x8_1-24-1_0.txt"
    28: "rectangular_8x8_1-32-1_0.txt"
    29: "rectangular_8x8_1-40-1_0.txt"
    30: "rectangular_8x9_1-24-1_0.txt"
    31: "rectangular_8x9_1-32-1_0.txt"
    32: "rectangular_8x9_1-40-1_0.txt"
    33: "rochester_53_10_0_pABC.txt"
    34: "rochester_53_12_0_pABC.txt"
    35: "rochester_53_16_0_pABC.txt"
    36: "rochester_53_20_0_pABC.txt"
    37: "rochester_53_4_0_pABC.txt"
    38: "rochester_53_8_0_pABC.txt"
    39: "sycamore_53_10_0.txt"
    40: "sycamore_53_12_0.txt"
    41: "sycamore_53_14_0.txt"
    42: "sycamore_53_16_0.txt"
    43: "sycamore_53_18_0.txt"
    44: "sycamore_53_20_0.txt"
    45: "sycamore_53_4_0.txt"
    46: "sycamore_53_5_0.txt"
    47: "sycamore_53_6_0.txt"
    48: "sycamore_53_8_0.txt"
    49: "test.txt"
]
```

```julia
# check available circuits
readdir(joinpath(@__DIR__, "data", "circuits"))
```

We load the circuit to Julia with Yao (幺), a high performance quantum simulator.

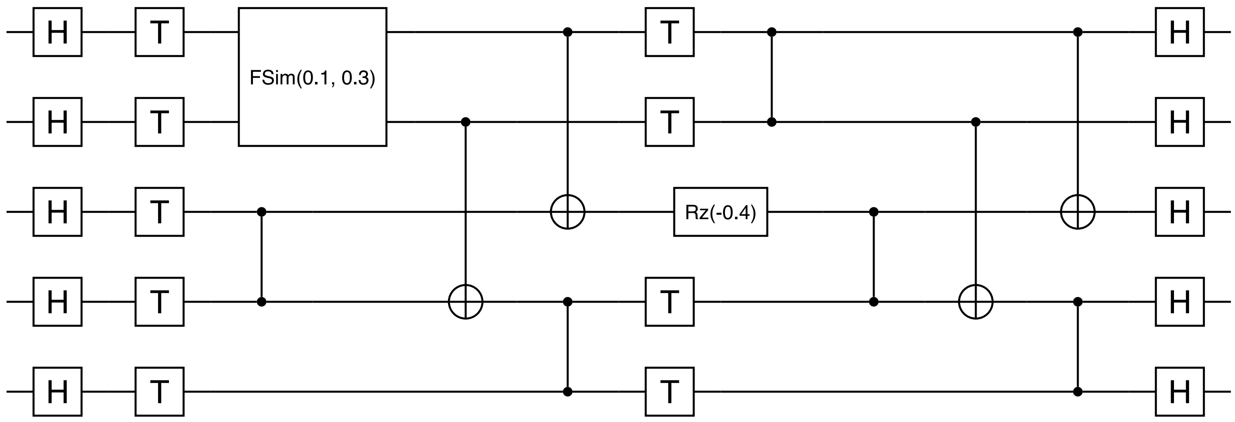
```
filename =
"/Users/liujinguo/jcode/tutorial-tensornetwork/examples/simulation/data/circuits/test.txt"
```

```julia
# Hint: please try replacing "test.txt" with "bristlecone_70_1-12-1_0.txt", a circuit with 70 qubits, 12 layers, see what happens
filename = joinpath(@__DIR__, "data", "circuits", "test.txt")
```

```julia
c = yaocircuit_from_file(filename);    # circuit in Yao's data-format
```

```
n = 5
```

```julia
n = nqubits(c)  # number of qubits
```




```julia
vizcircuit(c)
```

Step 2: construct tensor network

During the convertion, we also specify an optimizer to specify the contraction order.

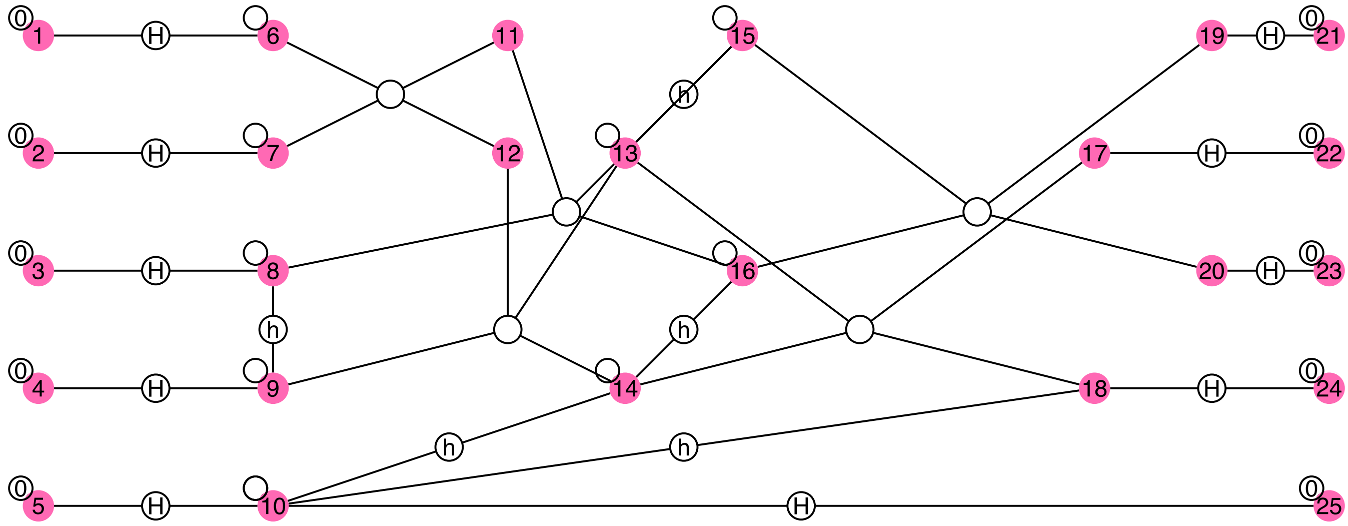
```
net = TensorNetwork
      Time complexity: 2^8.507794640198696
      Space complexity: 2^4.0
      Read-write complexity: 2^9.38586240064146
```

```julia
net = yao2einsum(c;
            initial_state= Dict(zip(1:n, zeros(Int,n))),
            final_state = Dict(zip(1:n, zeros(Int,n))),
            optimizer = TreeSA(ntrials=1)  # contraction order optimizer
        )
```

```
Time complexity: 2^8.507794640198696
Space complexity: 2^4.0
Read-write complexity: 2^9.38586240064146
```

```julia
contraction_complexity(net)
```

```julia
using LuxorGraphPlot  # Required by visualization extension
```




```
# red nodes are variables, transparent nodes are tensors
# h = [1 1; 1 -1] is the unnormalized version of Hadamard gate
# 0 tensor is defined as: [1, 0]
viznet(net; scale=60)
```

The space complexity is the number of elements in the largest itermediate tensor. For tensor network backend, it can be a much smaller number compared with the full amplitude simulation given the circuit is shallow enough. Learn more about contraction order optimizers: https://tensorbfs.github.io/OMEinsumContractionOrders.jl/dev/optimizers/

Step 3: contract the tensor network

If your circuit has space complexity less than 28, the tensor newtork is proababily contractable on your local device. Then please go ahead to check the following box.

contract_network = ☑

```
-0.044451061508327644 + 0.2223855135772236im
```

```
contract_network && Yao.contract(net)[]
```

The result should be consistent with the exact simulation.

exact_simulate = ☐

```
false
```

```
exact_simulate && apply(zero_state(n), c)' * zero_state(n)
```

Example 3: Construct tensor network for computing observables (channel simulation)

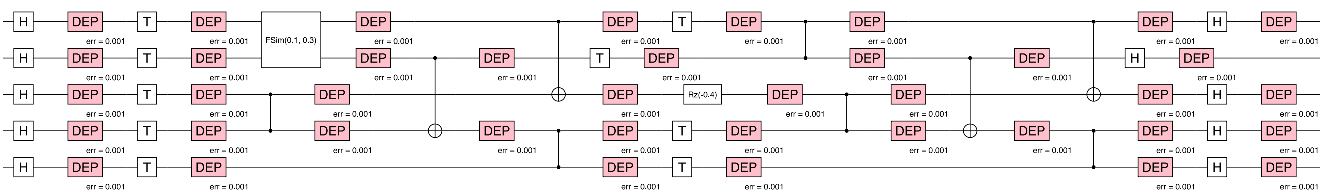
In this example, we show how to compute $\langle\psi|X_1X_2|\psi\rangle$ through quantum channel simulation, where $|\psi\rangle = U|0\rangle$, where U is the quantum circuit with interest. During the convertion, we also specify an optimizer to specify the contraction order.

```
add_depolarizing_noise (generic function with 1 method)
```

```julia
# add depolarizing noise
function add_depolarizing_noise(c::AbstractBlock, depolarizing)
    Optimise.replace_block(c) do blk
        if blk isa PutBlock || blk isa ControlBlock
            rep = chain(blk)
            for loc in occupied_locs(blk)
                push!(rep, put(nqubits(blk), loc=>DepolarizingChannel(1, depolarizing)))
            end
            return rep
        else
            return blk
        end
    end
end
```

Hint: please change the noise probability see how the result change with it.

```julia
noisy_c = add_depolarizing_noise(c, 0.001);
```




```julia
vizcircuit(noisy_c)
```

```
observable = nqubits: 5
             kron
             ├─ =>Z
             └─ =>Z
```

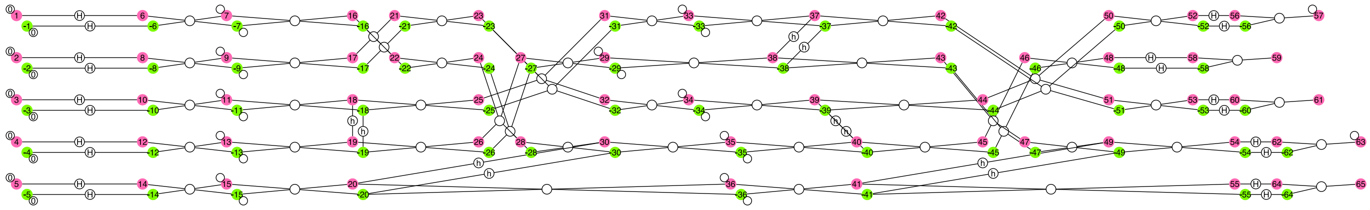
```julia
observable = kron(n, 1=>Z, 4=>Z)
```

```
noisy_net = TensorNetwork
            Time complexity: 2^13.023407843140218
            Space complexity: 2^8.0
            Read-write complexity: 2^12.588011853215086
```

```julia
noisy_net = yao2einsum(noisy_c;
                       initial_state=Dict(zip(1:n, zeros(Int,n))),
                       observable,
                       optimizer = TreeSA(ntrials=1),
                       mode=DensityMatrixMode())
```

```
Time complexity: 2^13.023407843140218
Space complexity: 2^8.0
Read-write complexity: 2^12.588011853215086
```

```julia
contraction_complexity(noisy_net)
```



```julia
# the green dots are dual variables
viznet(noisy_net; scale=60)
```

contract_noisy = ☑

```
0-dimensional Array{ComplexF64, 0}:
0.4464601777319247 + 4.163336342344336e-17im
```

```julia
contract_noisy && Yao.contract(noisy_net)
```

exact_noisy = ☑

```
0.446460177731925
```

```julia
exact_noisy && expect(observable, apply(density_matrix(zero_state(n)), noisy_c))
```

References

- **(qflex datasets)** B. Villalonga, et al., "A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware", NPJ Quantum Information 5, 86 (2019)
- **(Efficient simulation of noisy circuits)** Gao, Xun, and Luming Duan. "Efficient classical simulation of noisy quantum computation." arXiv preprint arXiv:1810.03176 (2018).
- **Tutorial page of YaoToEinsum**: https://docs.yaoquantum.org/dev/man/yao2einsum.html