Quantum circuit simulation with tensor network contraction

In this tutorial, we use Yao.jl as our default quantum simulation tool.

```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 v2.1.1 `~/.julia/dev/TensorQEC`
  [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
# `Yao` is a quantum simulator
# `OMEinsum` is a tensor network contraction engine
# `PlutoUI` is for control gadgets, e.g. the checkboxes
using Yao, OMEinsum, PlutoUI
```

Table of Contents

Quantum circuit simulation with tensor network contraction

- Example 1: GHZ state generation circuit
- Example 2: Simulate quantum supremacy experiments
  - Step 1: circuit loading
  - Step 2: construct tensor network
  - Step 3: contract the tensor network
- Example 3: Construct tensor network for computing observables (channel simulation)
- References

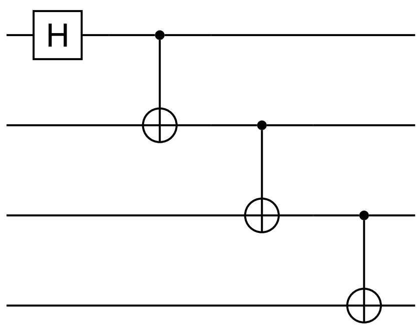

```julia
PlutoUI.TableOfContents(aside=false)
```

Example 1: GHZ state generation circuit

Let us first define a GHZ state.

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

```
# 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.

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

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

```
▸ (:code, :tensors, :label_to_qubit)
```

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

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

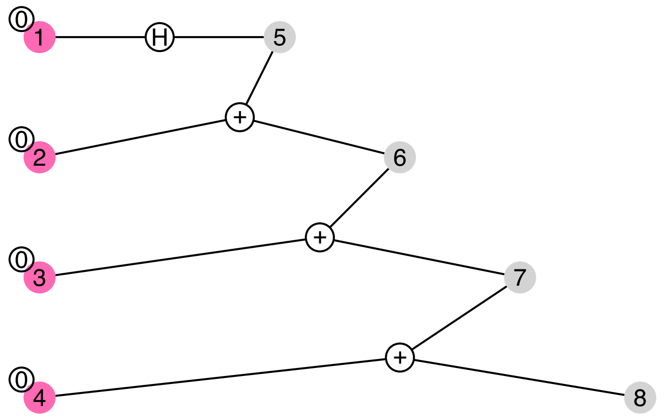
```
OMEinsum.getixsv(net_ghz.code)  # input tensor labels
```

```
8
```

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

```
[5, 6, 7, 8]
```

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




```julia
# 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
```

```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(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
```

```julia
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`:

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

```
["bristlecone_48_1-16-1_0.txt", "bristlecone_48_1-20-1_0.txt", "bristlecone_48_1-24-1_0.tx
```

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

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

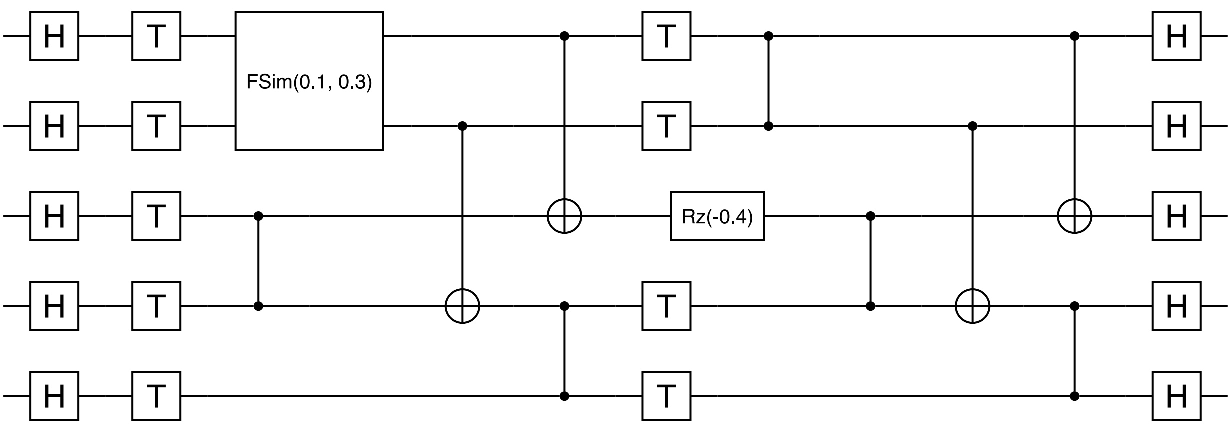
```
filename =
"/Users/liujinguo/jcode/tutorial-tensornetwork/examples/simulation/data/circuits/test.txt"
```

```
# 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")
```

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

```
n = 5
```

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




```
vizcircuit(c)
```

Step 2: construct tensor network

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

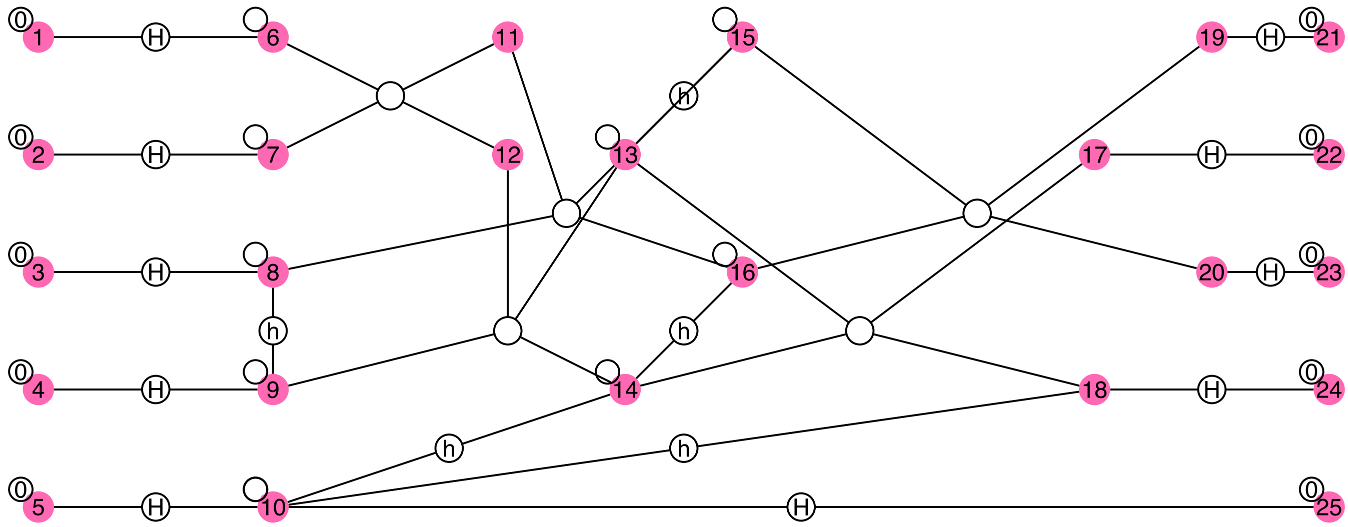
```
net = TensorNetwork
      Time complexity: 2^8.507794640198695
      Space complexity: 2^4.0
      Read-write complexity: 2^9.38586240064146
```

```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: 2^8.507794640198695
Space complexity: 2^4.0
Read-write complexity: 2^9.38586240064146
```

```julia
contraction_complexity(net)
```

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




```julia
# 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 proabably contractable on your local device. Then please go ahead to check the following box.

contract_network = ☑

```
-0.044451061508327686 + 0.22238551357722364im
```

```julia
contract_network && contract(net)[]
```

The result should be consistent with the exact simulation.

exact_simulate = ☑

```
-0.04445106150832767 - 0.2223855135772237im
```

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

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

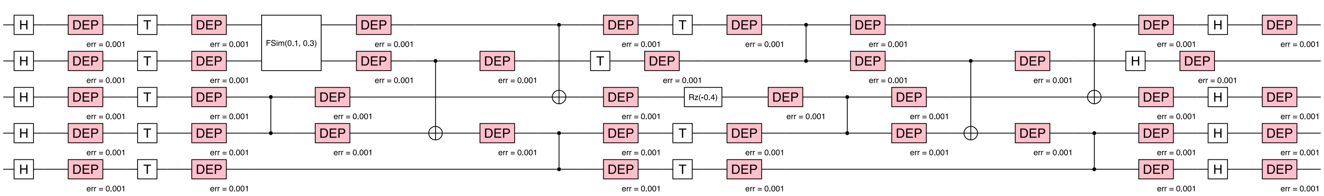
In this example, we show how to compute $\langle\psi|X_1X_2|\psi\rangle$ through quantum channel simulation, where $|\psi\rangle = U|0\rangle$, where U is the quantum circuit with interest. During the convertion, we also specify an optimizer to specify the contraction order.

```
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
```

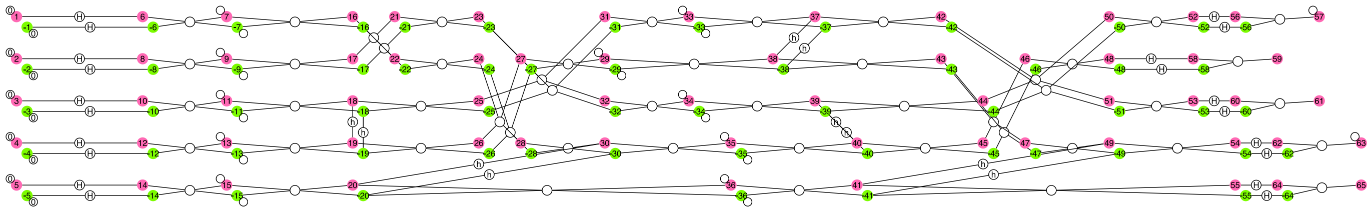
```julia
observable = kron(n, 1=>Z, 4=>Z)
```

```
noisy_net = TensorNetwork
            Time complexity: 2^13.026177596889228
            Space complexity: 2^8.0
            Read-write complexity: 2^12.595490606607642
```

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

```
Time complexity: 2^13.026177596889228
Space complexity: 2^8.0
Read-write complexity: 2^12.595490606607642
```

```
contraction_complexity(noisy_net)
```



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

```
contract_noisy = ☑
```

```
0-dimensional Array{ComplexF64, 0}:
0.4464601777319245 + 3.9217142730709714e-18im
```

```
contract_noisy && contract(noisy_net)
```

```
exact_noisy = ☑
```

```
0.446460177731925
```

```
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