```julia
using PlutoUI
```

About This notebook

This is a notebook about the quantum simulator Yao@v0.8, which could be download with the following link: https://raw.githubusercontent.com/GiggleLiu/YaoTutorial/master/notebooks/munich.jl

Contents

1. Overview of quantum simulation.
2. Why we create Yao.
3. A report on the current status of Yao.
4. A tutorial, covering latest Yao features.
5. Outlook.

Simulating quantum systems: The curse of dimensionality

Our world is probabilistic, the probability turns out to be complex valued.

1. If a computational system (or simulation) is deterministic, we say it is equivalent to a Turing Machine (TM).
2. If a computational system is parameterized by a classical real valued probability, we say it is a equivalent to a Probabilistic Turing Machine (PTM). Although we need $2^{\#\ of\ bits}$ numbers to parameterize such system, we believe they can be easily simulated with pseudo-random numbers on a TM.
3. If a computational system is parameterized by a complex valued probability, we say it is a equivalent to a Quantum Turing Machine (QTM). Simulating a QTM is NP-hard, and you need to store all $2^{\#\ of\ qubits}$ complex numbers.

Julia Quantum Ecosystem

A general truth about quantum simulation: general purposed v.s. larger scale

- Open quantum systems (a few subsystems)
  1. QuantumOptics.jl: Library for the numerical simulation of closed as well as open quantum systems.
- Circuit-based quantum simulation (~40 qubits)
  1. Yao.jl
  2. Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture.
  3. Braket.jl: a Julia implementation of the Amazon Braket SDK allowing customers to access Quantum Hardware and Simulators.
  4. PastaQ.jl: Package for Simulation, Tomography and Analysis of Quantum Computers
- Quantum many-body system (100 to ∞ spins)
  1. ITensors.jl: A Julia library for efficient tensor computations and tensor network calculations.
  2. QuantumLattices.jl: Julia package for the construction of quantum lattice systems.

Efficient circuit based simulation

ProjectQ: simulate up to 45 qubits, with 0.5 Petabyte memory (Steiger et al, 2018).

	JUQUEEN	K computer	Sunway TaihuLight	JURECA-CLUSTER	JUWELS
CPU	IBM PowerPC A2	eight-core SPARC64 VIIIfx	SW26010 manycore 64-bit RISC	Intel Xeon E5-2680 v3	Dual Intel Xeon Platinum 8168
clock frequency	1.6 GHz	2.0 Ghz	1.45 GHz	2.5 GHz	2.7 GHz
memory/node	16 GB	16 GB	32 GB	128 GB	96 GB
# threads/core used	1 – 2	8	1	1 – 2	1 – 2
# cores used	1 – 262144	2 – 65536	1 – 131072	1 – 6144	1 – 98304
# nodes used	1 – 16384	2 – 65536	1 – 32768	1 – 256	1 – 2048
# MPI processes used	1 – 524288	2 – 65536	1 – 131072	1 – 1024	1 – 2048
# qubits	46 (43)	48 (45)	48 (45)	43 (40)	46 (43)

References:

- Steiger D S, Häner T, Troyer M. ProjectQ: an open source software framework for quantum computing[J]. Quantum, 2018, 2: 49.
- De Raedt, Hans, Fengping Jin, Dennis Willsch, Madita Nocon, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. “Massively Parallel Quantum Computer Simulator, Eleven Years Later.” Computer Physics Communications 237 (April 2019): 47–61.

Yao.jl

An Extensible, Efficient Quantum Algorithm Design for Humans.

Yao is an open source framework that aims to empower quantum information research with software tools. It is designed with following in mind:

- quantum algorithm design;
- quantum software 2.0;
- quantum computation education.

Reference

- Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang and Lei Wang. Yao. jl: Extensible, efficient framework for quantum algorithm design. Quantum, 2020, 4: 341.

The Chinese character 幺 (Yāo) mean unitary.

Why yet another quantum simulator?

Differential programming a quantum circuit

- Variational quantum algorithms
- Inverse engineering, such as quantum optimal control

Variational Quantum Algorithms

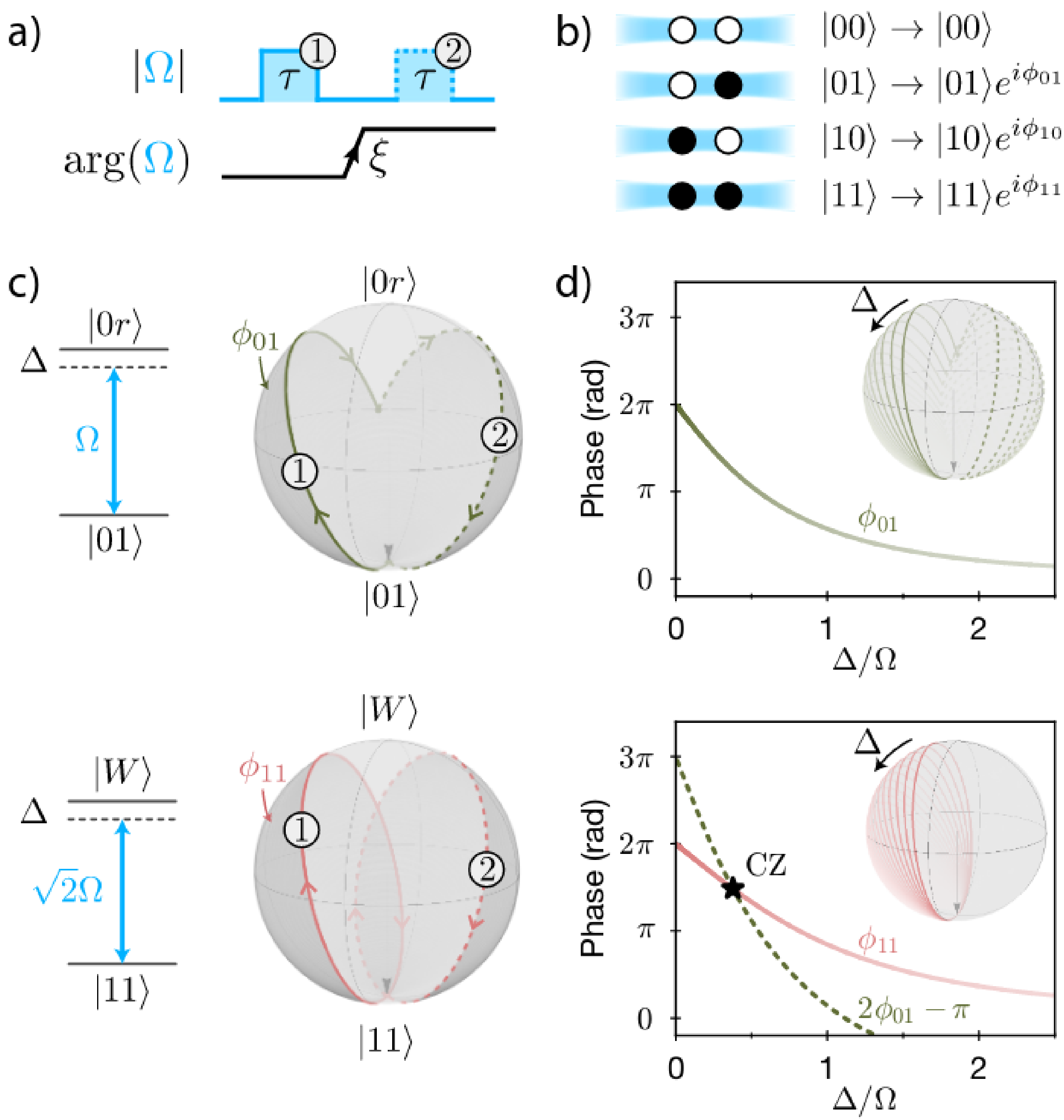
1. Quantum Circuit Born Machine: Using a quantum circuit as a probabilistic model $p(x) = |\langle x|\psi\rangle|^2$.
    - **Loss function**: maximum mean discrepancy between $|\langle x|\psi\rangle|^2$ and the target probability.
2. Variational Quantum Eigensolver: Solving the ground state of a quantum system.
    - **Loss function**: The energy expectation value.
3. Variational quantum optimization algorithms: Solving computational hard problems through variationally optimizing the annealing process.
    - **Loss function**: The success probability of finding an optimal solution.

References

- Kandala A, Mezzacapo A, Temme K, et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. nature, 2017, 549(7671): 242-246.
- Liu, Jin-Guo, and Lei Wang. "Differentiable Learning of Quantum Circuit Born Machines." Physical Review A 98, no. 6 (December 19, 2018): 062324.
- Liu, Jin-Guo, Yi-Hong Zhang, Yuan Wan, and Lei Wang. "Variational Quantum Eigensolver with Fewer Qubits." Physical Review Research 1, no.
- Ebadi, S., A. Keesling, M. Cain, T. T. Wang, H. Levine, D. Bluvstein, G. Semeghini, et al. "Quantum Optimization of Maximum Independent Set Using Rydberg Atom Arrays." Science 376, no. 6598 (June 10, 2022): 1209–15.

Quantum Control

- **Loss function**: the distance with the target gate.
- **Parameters**: the evolution times of each pulse.



References

- Levine, Harry, Alexander Keesling, Giulia Semeghini, Ahmed Omran, Tout T. Wang, Sepehr Ebadi, Hannes Bernien, et al. "Parallel Implementation of High-Fidelity Multiqubit Gates with Neutral Atoms." Physical Review Letters 123, no. 17 (2019): 1–16.

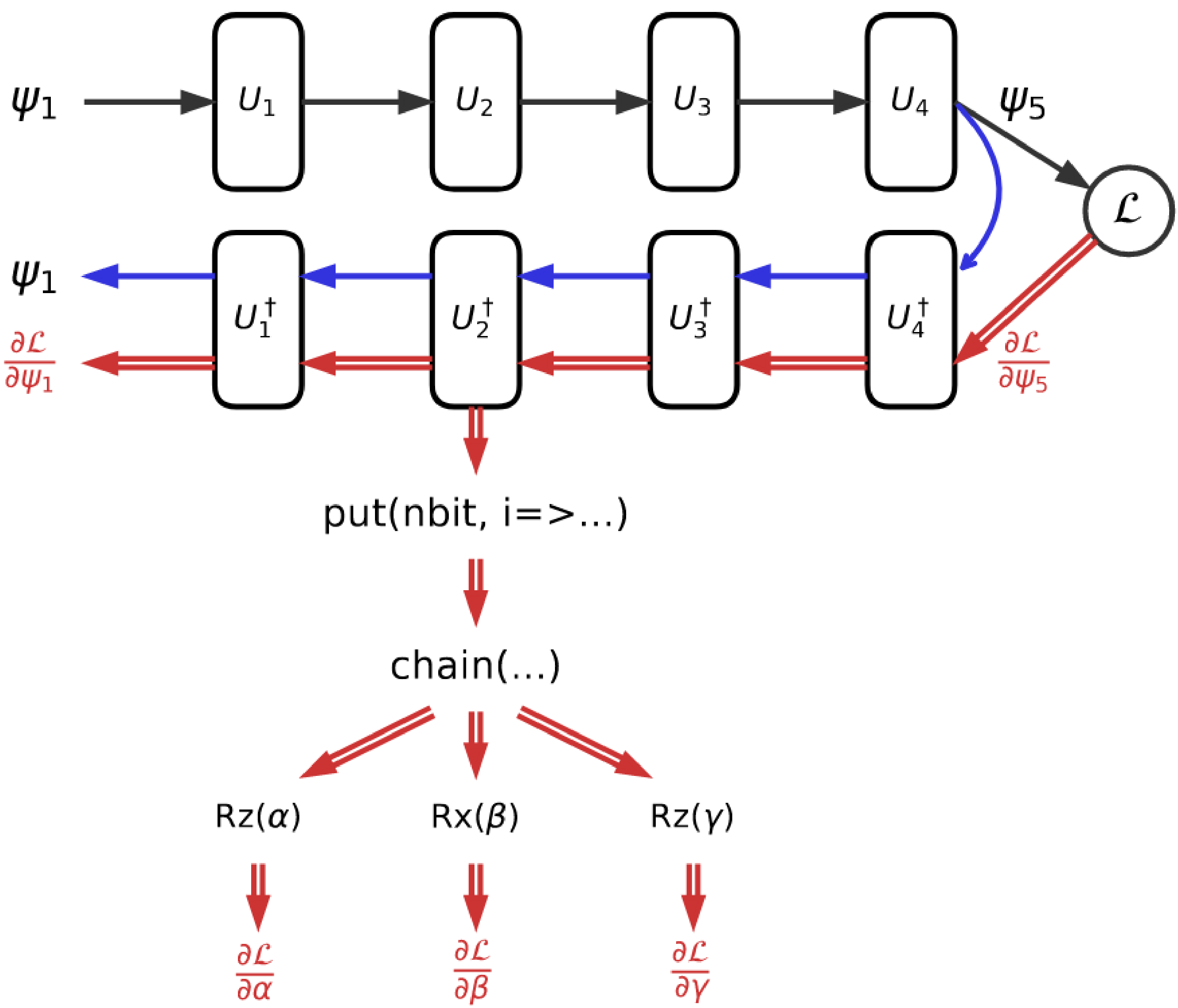
Challenges to differentiable programming quantum circuits

The memory wall problem.

- The back propagation technique requires knowing intermediate state for computing gradients.
- A quantum algorithm is memory consuming, caching all intermediate states is impossible.

We notice:

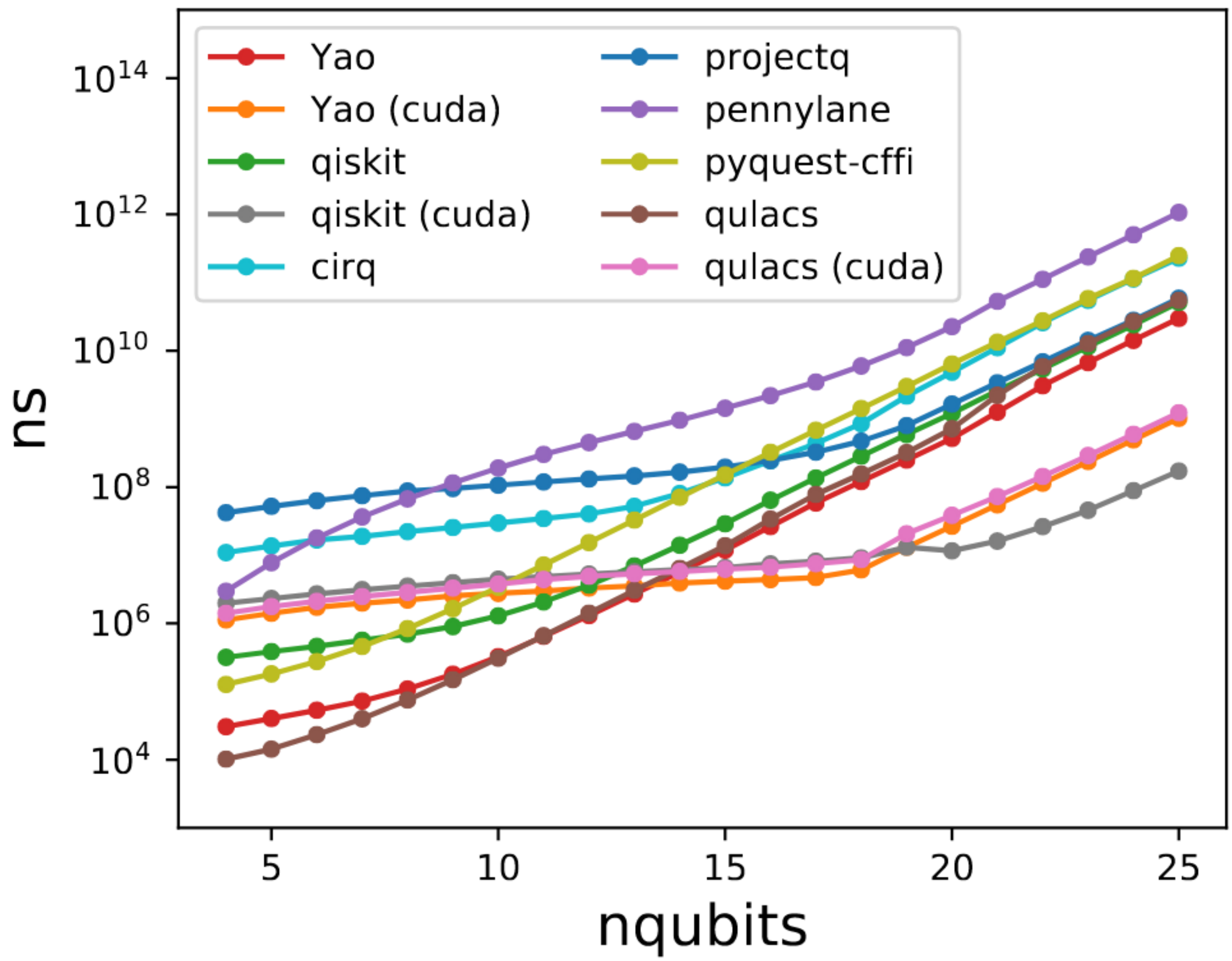
- quantum circuit simulation is reversible.



Performance

We benchmarked the simulation of a parameterized quantum circuit with single qubit rotation and CNOT gates. Please refer the Yao paper for details about the benchmark targets.

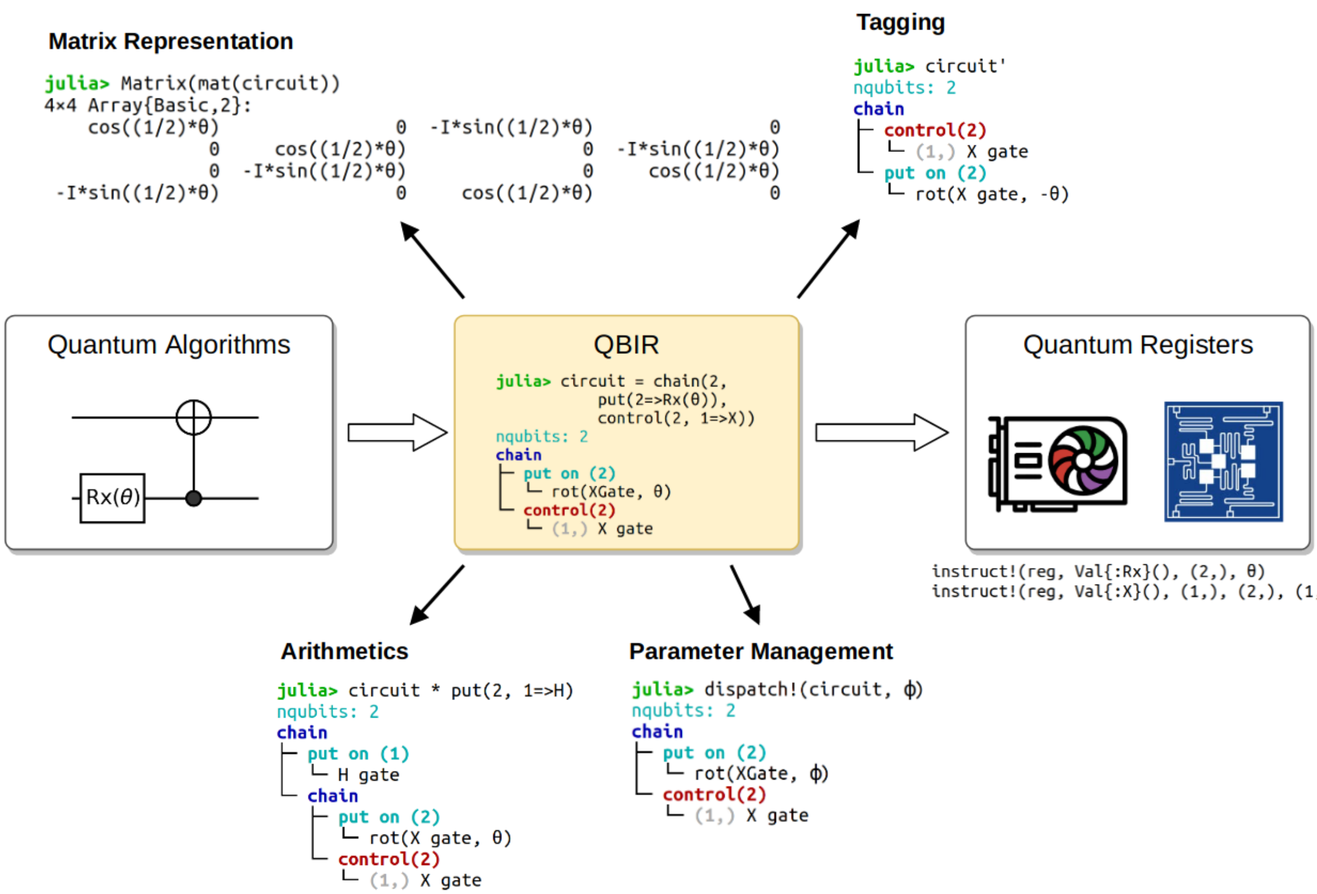
- Note: CuYao.jl is implemented with <600 lines of Julia code with CUDA.jl.



Overview of Yao features

Yao@v0.6

- Quantum simulation
- Matrix representation of quantum operators
    - Generate the sparse matrix representation of Hamiltonians with > 25 spins.
- Arithematic operations
- Parameter management and automatic differentiation
    - Differentiate a quantum circuit with depth > 10,000
- GPU backend



Current status of Yao.jl

Extra features in Yao@v0.8

- qudits: natively supported
    - 3-level Rydberg atom simulation is possible
- density matrix based simulation
    - only basic noisy channels supported
- operator indexing

Packages derived from Yao

- Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture (Roger Luo, QuEra Computing Inc. et al.)

- YaoToEinsum.jl: Convert Yao circuit to OMEinsum (tensor network) contraction (GiggleLiu et al.)
- YaoPlots.jl: plotting Yao circuit (GiggleLiu et al.)
- ZXCalculus.jl: An implementation of ZX-calculus in Julia (Chen Zhao, Roger Luo, Yusheng Zhao (through OSPP project) et al.)
- FLOYao.jl: A fermionic linear optics simulator backend for Yao.jl (Jan Lukas Bosse et al)
- QAOA.jl: This package implements the Quantum Approximate Optimization Algorithm and the Mean-Field Approximate Optimization Algorithm.
- QuantumNLDiffEq.jl

Papers citing Yao

Google scholar: up to Oct 7, 2023, Yao paper has 127 citations in total 🎉.

Related quantum simulators

- Pennylane: Automatic differentiation of hybrid quantum-classical computations
- Tensorflow quantum: A software framework for quantum machine learning
- Qulacs: a fast and versatile quantum circuit simulator for research purpose
- Qibo: a framework for quantum simulation with hardware acceleration
- Tequila: A platform for rapid development of quantum algorithms
- Qdnn: deep neural networks with quantum layers
- TenCirChem: An Efficient Quantum Computational Chemistry Package for the NISQ Era
- QuantumCumulants.jl: A Julia framework for generalized mean-field equations in open quantum systems
- Q^2Chemistry: A quantum computation platform for quantum chemistry
- QNet: A scalable and noise-resilient quantum neural network architecture for noisy intermediate-scale quantum computers
- Qforte: an efficient state simulator and quantum algorithms library for molecular electronic structure
- QuantNBody: a Python package for quantum chemistry and physics to build and manipulate many-body operators and wave functions.
- UniQ: a unified programming model for efficient quantum circuit simulation
- tqix.pis: A toolbox for quantum dynamics simulation of spin ensembles in Dicke basis
- BosonSampling.jl: A Julia package for quantum multi-photon interferometry
- QDNN: DNN with quantum neural network layers
- TeD-Q: a tensor network enhanced distributed hybrid quantum machine learning framework
- QXTools: A Julia framework for distributed quantum circuit simulation
- Heom.jl: An efficient Julia framework for hierarchical equations of motion in open quantum systems
- QUBO. jl: A Julia Ecosystem for Quadratic Unconstrained Binary Optimization
- HiQ-ProjectQ: Towards user-friendly and high-performance quantum computing on GPUs
- HyQuas: hybrid partitioner based quantum circuit simulation system on GPU

How do people use Yao?

- Variational quantum algorithms

    - Variational Quantum Eigensolvers, Quantum Neural networks, Quantum Approximate Optimization Algorithm, Solving differential equation, Quantum kernel method
- Fermionic simulation
    - Sketching phase diagrams using low-depth variational quantum algorithms
- Combinatorial optimization
    - Tropical Tensor Network for Ground States of Spin Glasses
- Measurement induced phase transition
    - Simulating a measurement-induced phase transition for trapped ion circuits
- Imaginary time evolution
    - Probabilistic nonunitary gate in imaginary time evolution
    - Efficient quantum imaginary time evolution by drifting real time evolution: an approach with low gate and measurement complexity
- Tensor network based simulation
    - Efficient and Portable Einstein Summation in SQL
    - Contracting Arbitrary Tensor Networks: General Approximate Algorithm and Applications in Graphical Models and Quantum Circuit Simulations
- Quantum Chebyshev Transform
    - Quantum Chebyshev Transform: Mapping, Embedding, Learning and Sampling Distributions
- Geologic fracture networks
    - Quantum algorithms for geologic fracture networks
- Teaching
    - Using the Julia framework to teach quantum entanglement
- Hamiltonian Operator Approximation
    - Hamiltonian Operator Approximation for Energy Measurement and Ground-State Preparation

Learn Yao

```
using Yao, YaoPlots; YaoPlots.darktheme!();
```

Part 1: Representing a quantum state

```
ArrayReg{2, ComplexF64, Array...}
    active qubits: 3/3
    nlevel: 2
```

```julia
# create a zero state |000⟩
zero_state(3)
```

```
ComplexF64[
    1:  1.0+0.0im
    2:  0.0+0.0im
    3:  0.0+0.0im
    4:  0.0+0.0im
    5:  0.0+0.0im
    6:  0.0+0.0im
    7:  0.0+0.0im
    8:  0.0+0.0im
]
```

```julia
# The quantum state is represented as a vector
statevec(zero_state(3))
```

```julia
print_table(zero_state(3))
```

```
000 ₍₂₎   1.0 + 0.0im
001 ₍₂₎   0.0 + 0.0im
010 ₍₂₎   0.0 + 0.0im
011 ₍₂₎   0.0 + 0.0im
100 ₍₂₎   0.0 + 0.0im
101 ₍₂₎   0.0 + 0.0im
110 ₍₂₎   0.0 + 0.0im
111 ₍₂₎   0.0 + 0.0im
```

```
ArrayReg{2, ComplexF32, Array...}
    active qubits: 3/3
    nlevel: 2
```

```julia
# Similarly, we can create a random state
# The element type is also configurable
rand_state(ComplexF32, 3)
```

```
ArrayReg{2, ComplexF64, Array...}
    active qubits: 3/3
    nlevel: 2
```

```julia
# A product state
product_state(bit"110")
```

```
1
```

```julia
# note the bit string representation is in the little endian format.
bit"110"[3]
```

```julia
# To print the elements with basis annotated
print_table(product_state(bit"110"))
```

```
000 ₍₂₎   0.0 + 0.0im
001 ₍₂₎   0.0 + 0.0im
010 ₍₂₎   0.0 + 0.0im
011 ₍₂₎   0.0 + 0.0im
100 ₍₂₎   0.0 + 0.0im
101 ₍₂₎   0.0 + 0.0im
110 ₍₂₎   1.0 + 0.0im
111 ₍₂₎   0.0 + 0.0im
```

```
ArrayReg{2, ComplexF64, Array...}
    active qubits: 3/3
    nlevel: 2
```

```julia
# A GHZ state
ghz_state(3)
```

```julia
print_table(ghz_state(3))
```

```
000 ₍₂₎   0.70711 - 0.0im
001 ₍₂₎   0.0 + 0.0im
010 ₍₂₎   0.0 + 0.0im
011 ₍₂₎   0.0 + 0.0im
100 ₍₂₎   0.0 + 0.0im
101 ₍₂₎   0.0 + 0.0im
110 ₍₂₎   0.0 + 0.0im
111 ₍₂₎   0.70711 - 0.0im
```

```
1.0000000000000229
```

```julia
# there is a single bit entanglement entropy between qubit sets (1, 3) and (2,)
von_neumann_entropy(ghz_state(3), (1, 3)) / log(2)
```

```
3.4638958368304884e-14
```

```julia
von_neumann_entropy(zero_state(3), (1, 3)) / log(2)
```

```
ArrayReg{3, ComplexF64, Array...}
    active qudits: 3/3
    nlevel: 3
```

```julia
# A random qutrit state
rand_state(3, nlevel=3)
```

```
ArrayReg{3, ComplexF64, Array...}
    active qudits: 3/3
    nlevel: 3
```

```julia
# A qudit product state, what follows ";" symbol denotes the number of levels
product_state(dit"120;3")
```

```
print_table(product_state(dit"120;3"))
```

```
000 ₍₃₎   0.0 + 0.0im
001 ₍₃₎   0.0 + 0.0im
002 ₍₃₎   0.0 + 0.0im
010 ₍₃₎   0.0 + 0.0im
011 ₍₃₎   0.0 + 0.0im
012 ₍₃₎   0.0 + 0.0im
020 ₍₃₎   0.0 + 0.0im
021 ₍₃₎   0.0 + 0.0im
022 ₍₃₎   0.0 + 0.0im
100 ₍₃₎   0.0 + 0.0im
101 ₍₃₎   0.0 + 0.0im
102 ₍₃₎   0.0 + 0.0im
110 ₍₃₎   0.0 + 0.0im
111 ₍₃₎   0.0 + 0.0im
112 ₍₃₎   0.0 + 0.0im
120 ₍₃₎   1.0 + 0.0im
121 ₍₃₎   0.0 + 0.0im
122 ₍₃₎   0.0 + 0.0im
200 ₍₃₎   0.0 + 0.0im
201 ₍₃₎   0.0 + 0.0im
202 ₍₃₎   0.0 + 0.0im
210 ₍₃₎   0.0 + 0.0im
211 ₍₃₎   0.0 + 0.0im
212 ₍₃₎   0.0 + 0.0im
220 ₍₃₎   0.0 + 0.0im
221 ₍₃₎   0.0 + 0.0im
222 ₍₃₎   0.0 + 0.0im
```

Part 2: representing a quantum circuit

A quantum operators such as Hamiltonians and quantum circuits, are represented as a matrix, or a Yao block. There are two types of blocks:

- primitive blocks: the basic building blocks of a quantum circuit.
- composite blocks: composing primitive blocks into Hamiltonians and circuits

Part 2.1: Primitive blocks

Primitive blocks are the basic building blocks of a quantum circuit.

X

```
# Pauli X gate
X
```

```julia
# visualize a gate
vizcircuit(X)
```

```julia
using SymEngine
```

```
(θ)
```

```julia
# create a symbolic variable θ
@vars θ
```

```
rot(X, θ)
```

```julia
# Rotation X gate
rot(X, θ)
```

```
[θ]
```

```julia
parameters(rot(X, θ))
```

```julia
vizcircuit(rot(X, θ))
```

```
2×2 SparseMatrixCSC{Basic, Int64} with 4 stored entries:
     cos((1/2)*θ)  -im*sin((1/2)*θ)
 -im*sin((1/2)*θ)      cos((1/2)*θ)
```

```julia
# The matrix representation
mat(rot(X, θ))
```

```
4×4 SparseMatrixCSC{Basic, Int64} with 6 stored entries:
 -im*sin((1/2)*θ) + cos((1/2)*θ)  …                                ⋅
                                 ⋅                                 ⋅
                                 ⋅                                 ⋅
                                 ⋅     -im*sin((1/2)*θ) + cos((1/2)*θ)
```

```julia
# The first argument of `rot` can be any reflexive operator, i.e. O² = 1
# Parameterized SWAP
mat(rot(SWAP, θ))
```

```
true
```

```julia
isreflexive(SWAP)
```

```
true
```

```julia
isreflexive(kron(SWAP, X))
```

```
2×2 Diagonal{Basic, Vector{Basic}}:
 exp(im*θ)          ⋅
         ⋅   exp(im*θ)
```

```
# Phase gate
mat(phase(θ))
```

```
vizcircuit(phase(θ))
```

```
2×2 Diagonal{Basic, Vector{Basic}}:
 1          ⋅
 ⋅   exp(im*θ)
```

```
# Shift gate
mat(shift(θ))
```

```
vizcircuit(shift(θ))
```

```
random_gate
```

```
# random single qubit matrix block
matblock(rand_unitary(2); tag="random_gate")
```

```
two 3-level
```

```
# random 2 qutrit matrix block
matblock(rand_unitary(9); nlevel=3, tag="two 3-level")
```

```
vizcircuit(matblock(rand_unitary(9); nlevel=3, tag="two 3-level"))
```

```
Measure(2)
```

```
# random 2 qutrit matrix block
Measure(2)
```

```
vizcircuit(Measure(2))
```

```
Time Evolution Δt = 0.3, tol = 1.0e-7
X
```

```
# Time evolution, the first argument can be any Hermitian operator
time_evolve(X, 0.3)
```

```
true
```

```
ishermitian(kron(X+Y, X))
```

Part 2.2: Composite blocks

```
nqubits: 3
put on (1)
└─ X
```

```
# Put a block at the first qubit of a 3-qubit register
put(3, 1=>X)
```

```
vizcircuit(put(3, 1=>X))
```

```
8×8 LuxurySparse.SDPermMatrix{ComplexF64, Int64, Vector{ComplexF64}, Vector{Int64}}:
 0.0+0.0im  1.0+0.0im  0.0+0.0im  0.0+0.0im  …  0.0+0.0im  0.0+0.0im  0.0+0.0im
 1.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  1.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  1.0+0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im     1.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im  …  0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  1.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im     0.0+0.0im  1.0+0.0im  0.0+0.0im
```

```
mat(put(3, 1=>X))
```

```
nqubits: 10
put on (5, 2, 1)
└─ Toffoli
```

```
# The target gate can be applied on any subset of qubits
put(10, (5, 2, 1) => ConstGate.Toffoli)
```

```
vizcircuit(put(10, (5, 2, 1) => ConstGate.Toffoli))
```

```
nqubits: 2
kron
├─ 1=>X
└─ 2=>X
```

```
# Kronecker product of two blocks
kron(X, X)
```

```julia
vizcircuit(kron(X, X))
```

```
nqubits: 10
kron
├─ 2=>X
└─ 3=>Y
```

```julia
# A more general form can be
kron(10, 2=>X, 3=>Y)
```

```julia
vizcircuit(kron(10, 2=>X, 3=>Y))
```

```
nqubits: 3
control(1)
└─ (2,) X
```

```
# Control the second qubit of a 3-qubit register with the first qubit
control(3, 1, 2=>X)
```

```
vizcircuit(control(3, 1, 2=>X))
```

```
nqubits: 10
control(1, ¬8)
└─ (7, 6) kron
   ├─ 1=>H
   └─ 2=>rot(Z, 0.7853981633974483)
```

```
# Multi-control and inverse control is supported
# Example: if and only if qubit 1 is 1 and qubit 8 is 0,
# apply 2-qubit gate `kron(H, Rz(π/4))` on position (7, 6).
control(10, (1, -8), (7, 6)=>kron(H, Rz(π/4)))
```

```
vizcircuit(control(10, (1, -8), (7, 6)=>kron(H, Rz(π/4))))
```

```
nqubits: 3
chain
├─ put on (1)
│  └─ X
└─ control(1)
   └─ (2,) X
```

```
# Chain two blocks into a circuit
chain(3, put(3, 1=>X), control(3, 1, 2=>X))
```

```julia
vizcircuit(chain(3, put(3, 1=>X), control(3, 1, 2=>X)))
```

```
nqubits: 3
chain
├─ put on (1)
│  └─ X
└─ control(1)
   └─ (2,) X
```

```julia
# It is equivalent to inverse ordered operator multiplication.
control(3, 1, 2=>X) * put(3, 1=>X)
```

```
[+im] X
```

```julia
# Scaling a block
im * X
```

```
nqubits: 3
+
├─ kron
│  ├─ 1=>X
│  └─ 2=>X
└─ kron
   ├─ 2=>X
   └─ 3=>X
```

```julia
# X₁X₂ + X₂X₃
sum([kron(3, 1=>X, 2=>X), kron(3, 2=>X, 3=>X)])
```

Example: quantum Fourier transformation simulation (QFT)

```
nqubits: 4
chain
├─ chain
│  ├─ put on (1)
│  │  └─ H
│  ├─ control(2)
│  │  └─ (1,) shift(1.5707963267948966)
│  ├─ control(3)
│  │  └─ (1,) shift(0.7853981633974483)
│  └─ control(4)
│     └─ (1,) shift(0.39269908169872414)
├─ chain
│  ├─ put on (2)
│  │  └─ H
│  ├─ control(3)
│  │  └─ (2,) shift(1.5707963267948966)
│  └─ control(4)
│     └─ (2,) shift(0.7853981633974483)
├─ chain
│  ├─ put on (3)
│  │  └─ H
│  └─ control(4)
│     └─ (3,) shift(1.5707963267948966)
└─ chain
   └─ put on (4)
      └─ H
```

```
# A QFT circuit is available in `Yao.EasyBuild` module.
EasyBuild.qft_circuit(4)
```

Step by step

```
# Let's first define the CPHASE gate
cphase(n, i, j) = control(n, i, j=> shift(2π/(2^(i-j+1))));
```

```
vizcircuit(cphase(5, 2, 1))
```

```
4×4 Diagonal{Basic, Vector{Basic}}:
 1  ⋅  ⋅          ⋅
 ⋅  1  ⋅          ⋅
 ⋅  ⋅  1          ⋅
 ⋅  ⋅  ⋅  exp(im*θ)
```

```julia
# A cphase is defined as
mat(control(2, 2, 1=> shift(θ)))
```

```julia
hcphases(n, i) = chain(n, i==j ? put(n, i=>H) : cphase(n, j, i) for j in i:n);
```

```julia
vizcircuit(hcphases(5, 2))
```

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

```julia
# with CPHASE gate, we have the qft circuit defined as
qft_circ(n::Int) = chain(n, hcphases(n, i) for i = 1:n)
```

```
qft = nqubits: 3
      chain
      ├─ chain
      │  ├─ put on (1)
      │  │  └─ H
      │  ├─ control(2)
      │  │  └─ (1,) shift(1.5707963267948966)
      │  └─ control(3)
      │     └─ (1,) shift(0.7853981633974483)
      ├─ chain
      │  ├─ put on (2)
      │  │  └─ H
      │  └─ control(3)
      │     └─ (2,) shift(1.5707963267948966)
      └─ chain
         └─ put on (3)
            └─ H
```

```julia
qft = qft_circ(3)
```

```julia
vizcircuit(qft)
```

```
8×8 Matrix{ComplexF64}:
 0.353553+0.0im   0.353553+0.0im      0.353553+0.0im       …      0.353553+0.0im
 0.353553+0.0im  -0.353553+0.0im   2.16489e-17+0.353553im            0.25-0.25im
 0.353553+0.0im   0.353553+0.0im     -0.353553+0.0im          -2.16489e-17-0.353553im
 0.353553+0.0im  -0.353553+0.0im  -2.16489e-17-0.353553im           -0.25-0.25im
 0.353553+0.0im   0.353553+0.0im      0.353553+0.0im             -0.353553+0.0im
 0.353553+0.0im  -0.353553+0.0im   2.16489e-17+0.353553im  …        -0.25+0.25im
 0.353553+0.0im   0.353553+0.0im     -0.353553+0.0im           2.16489e-17+0.353553im
 0.353553+0.0im  -0.353553+0.0im  -2.16489e-17-0.353553im            0.25+0.25im
```

```julia
# Let us check the matrix representation
mat(qft) |> Matrix
```

```
false
```

```julia
# Matrix properties
ishermitian(qft)
```

```
true
```

```julia
isunitary(qft)
```

```
false
```

```julia
isreflexive(qft)
```

```
iqft = nqubits: 3
       chain
       ├─ chain
       │  └─ put on (3)
       │     └─ H
       ├─ chain
       │  ├─ control(3)
       │  │  └─ (2,) shift(-1.5707963267948966)
       │  └─ put on (2)
       │     └─ H
       └─ chain
          ├─ control(3)
          │  └─ (1,) shift(-0.7853981633974483)
          ├─ control(2)
          │  └─ (1,) shift(-1.5707963267948966)
          └─ put on (1)
             └─ H
```

```julia
# The dagger of qft is the inverse-qft
iqft = qft'
```

```julia
vizcircuit(iqft)
```

```
reg3 = ArrayReg{2, ComplexF64, Array...}
           active qubits: 3/3
           nlevel: 2
```

```julia
# Run the circuit
reg3 = product_state(bit"011")
```

```
out = ArrayReg{2, ComplexF64, Array...}
          active qubits: 3/3
          nlevel: 2
```

```julia
out = copy(reg3) |> qft
```

```
true
```

```julia
copy(out) |> iqft ≈ reg3
```

```
resqft3 =
 [101 (2), 011 (2), 001 (2), 110 (2), 111 (2), 000 (2), 100 (2), 110 (2), 100 (2), 101 (2)]
```

```julia
# Measure the output (without collapsing state)
resqft3 = measure(out; nshots=10)
```

```
resqft3_inplace = 011 (2)
```

```julia
# Measure the output and collapsing state
resqft3_inplace = measure!(out)
```

```
1
```

```julia
# bit strings can be indexed (little endian)
resqft3_inplace[1][1]
```

```
reg20 = ArrayReg{2, ComplexF64, Array...}
            active qubits: 20/20
            nlevel: 2
```

```julia
# Run this quantum algorithm on a 20 qubit register at qubits (4,6,7)
reg20 = rand_state(20)
```

```
ArrayReg{2, ComplexF64, Array...}
    active qubits: 20/20
    nlevel: 2
```

```julia
apply(reg20, subroutine(20, qft, (4,6,7)))
```

Example: Simulate a variational quantum algorithms

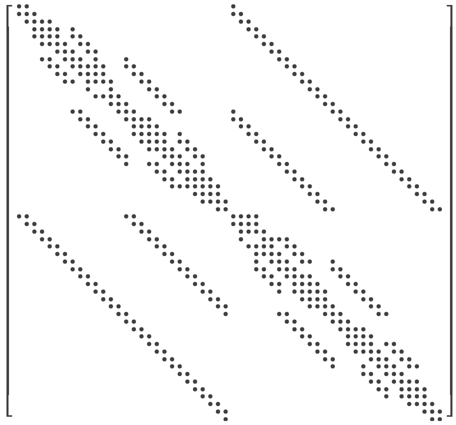
nbit = 10

```
nbit = 10
```

```
hami = nqubits: 10
      +
      ├─ +
      │  ├─ repeat on (1, 2)
      │  │  └─ X
      │  ├─ repeat on (1, 2)
      │  │  └─ Y
      │  └─ repeat on (1, 2)
      │     └─ Z
      ├─ +
      │  ├─ repeat on (2, 3)
      │  │  └─ X
      │  ├─ repeat on (2, 3)
      │  │  └─ Y
      │  └─ repeat on (2, 3)
      │     └─ Z
      ├─ +
      │  ├─ repeat on (3, 4)
      │  │  └─ X
      │  ├─ repeat on (3, 4)
      │  │  └─ Y
      │  └─ repeat on (3, 4)
      │     └─ Z
      ├─ +
      │  ├─ repeat on (4, 5)
      │  │  └─ X
      │  ├─ repeat on (4, 5)
      │  │  └─ Y
      │  └─ repeat on (4, 5)
      │     └─ Z
      ├─ +
      │  ├─ repeat on (5, 6)
      │  │  └─ X
```

```
# the hamiltonian
hami = EasyBuild.heisenberg(nbit)
```

```
hmat = 1024×1024 SparseMatrixCSC{ComplexF64, Int64} with 6144 stored entries:
```



```julia
# exact diagonalization
hmat = mat(hami)
```

```
YaoBlocks.EntryTable{BitBasis.DitStr{2, 10, Int64}, ComplexF64}:
  0001001110 ₍₂₎   2.0 + 0.0im
  0010001101 ₍₂₎   2.0 + 0.0im
  0010001110 ₍₂₎   2.0 + 0.0im
  0010010110 ₍₂₎   2.0 + 0.0im
  0100001110 ₍₂₎   2.0 + 0.0im
```

```julia
# If you only want to get one column, the following way is much faster
hami[:,bit"0010001110"]
```

```julia
using KrylovKit
```

```
(
    1:   [-18.0618]
    2:   [[-4.30165e-20-2.34279e-19im, 1.36979e-17+1.9506e-17im, -7.95635e-17+5.62738e-17
    3: ConvergenceInfo: one converged value after 2 iterations and 42 applications of the
       norms of residuals are given by (6.290937935392942e-14,).
)
```

```julia
# `eigsolve` is for solving dominant eigenvalue problem of the target Hamiltonian
# the second argument `1` means converging at least one eigenvectors.
# the thrid argument `:SR` means finding the eigenvalue with the smallest real part.
eg, vg = KrylovKit.eigsolve(hmat, 1, :SR)
```

```
vcirc = nqubits: 10
        chain
        ├─ chain
        │  ├─ chain
        │  │  ├─ put on (1)
        │  │  │  └─ rot(X, 0.12739253456281674)
        │  │  └─ put on (1)
        │  │     └─ rot(Z, 0.613875452373382)
        │  ├─ chain
        │  │  ├─ put on (2)
        │  │  │  └─ rot(X, 0.5570926552488839)
        │  │  └─ put on (2)
        │  │     └─ rot(Z, 0.47916700842395943)
        │  ├─ chain
        │  │  ├─ put on (3)
        │  │  │  └─ rot(X, 0.033334051639442l4)
        │  │  └─ put on (3)
        │  │     └─ rot(Z, 0.8930468262211081)
        │  ├─ chain
        │  │  ├─ put on (4)
        │  │  │  └─ rot(X, 0.7399524546910887)
        │  │  └─ put on (4)
        │  │     └─ rot(Z, 0.882622926676953)
        │  ├─ chain
        │  │  ├─ put on (5)
        │  │  │  └─ rot(X, 0.6824581127417725)
        │  │  └─ put on (5)
        │  │     └─ rot(Z, 0.4750318290323525)
        │  ├─ chain
        │  │  ├─ put on (6)
        │  │  │  └─ rot(X, 0.18033500901243282)
        │  │  └─ put on (6)
        │  │     └─ rot(Z, 0.5587565071148205)
```

```
# variational quantum circuit
vcirc = dispatch(EasyBuild.variational_circuit(nbit), :random)
```

```
vizcircuit(vcirc)
```

```
1.1754755102123555 - 5.204170427930421e-18im
```

```
# energy expectation value as the loss
expect(hami, zero_state(nbit) => vcirc)
```

```
ArrayReg{2, ComplexF64, Array...}
    active qubits: 10/10
    nlevel: 2                    ⇒   [-0.448835, -0.0290929, -0.277615, -0.135892, -0.042
```

```
# differentiate the energy function
# the return value is a pair: (gradient of initial state, gradient of circuit parameters)
expect'(hami, zero_state(nbit) => vcirc)
```

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

```julia
function train_vcirc(vcirc; nstep)
    for i = 1:nstep
        # compute gradient
        regδ, paramsδ = expect'(hami, zero_state(nbit)=>vcirc)
        # update parameters with gradient descent, check Optim.jl for advanced
        gradient based optimizers, such as BFGS.
        vcirc = dispatch(-, vcirc, 0.1*paramsδ)
        # show the loss
        energy = real(expect(hami, zero_state(nbit) => vcirc))
        @info "Mean energy at step $i is $energy"
    end
    return vcirc
end
```

```
│   │   └─ put on (3)
│   │      └─ rot(Z, 0.7161354697174853)
│   ├─ chain
│   │  ├─ put on (4)
│   │  │  └─ rot(X, 3.9834881609279596e-5)
│   │  └─ put on (4)
│   │     └─ rot(Z, 0.9549353114249319)
│   ├─ chain
│   │  ├─ put on (5)
│   │  │  └─ rot(X, 0.001085164740188195)
│   │  └─ put on (5)
│   │     └─ rot(Z, 0.6828022195941278)
│   ├─ chain
│   │  ├─ put on (6)
│   │  │  └─ rot(X, -0.00010073099691501174)
│   │  └─ put on (6)
│   │     └─ rot(Z, 0.5937211881461927)
│   ├─ chain
│   │  ├─ put on (7)
│   │  │  └─ rot(X, 0.01156426736557754)
│   │  └─ put on (7)
│   │     └─ rot(Z, 0.8012087403570878)
│   ├─ chain
│   │  ├─ put on (8)
│   │  │  └─ rot(X, 0.032108775999986766)
│   │  └─ put on (8)
│   │     └─ rot(Z, 0.8274085059504384)
│   ├─ chain
│   │  ├─ put on (9)
│   │  │  └─ rot(X, 0.2198964031925498)
│   │  └─ put on (9)
│   │     └─ rot(Z, 0.3544318664586506)
│   └─ chain
```

```
train_vcirc(vcirc; nstep=100)
```

```
Mean energy at step 100 is -14.847182887507067
```

CuYao: Speed up your quantum simulation with GPU

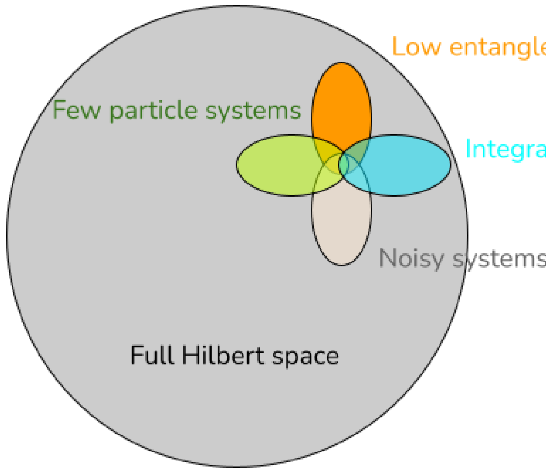
The following live coding simulates a QFT circuit on GPU. It requires

- A GPU with CUDA support
- Julia package `CuYao.jl`

Source code is available in file: `clips/yao-v0.8-cuda.jl`

Different circuit simulation tracks

There are special cases that quantum systems are efficiently simulatable by a classical computing device.



References

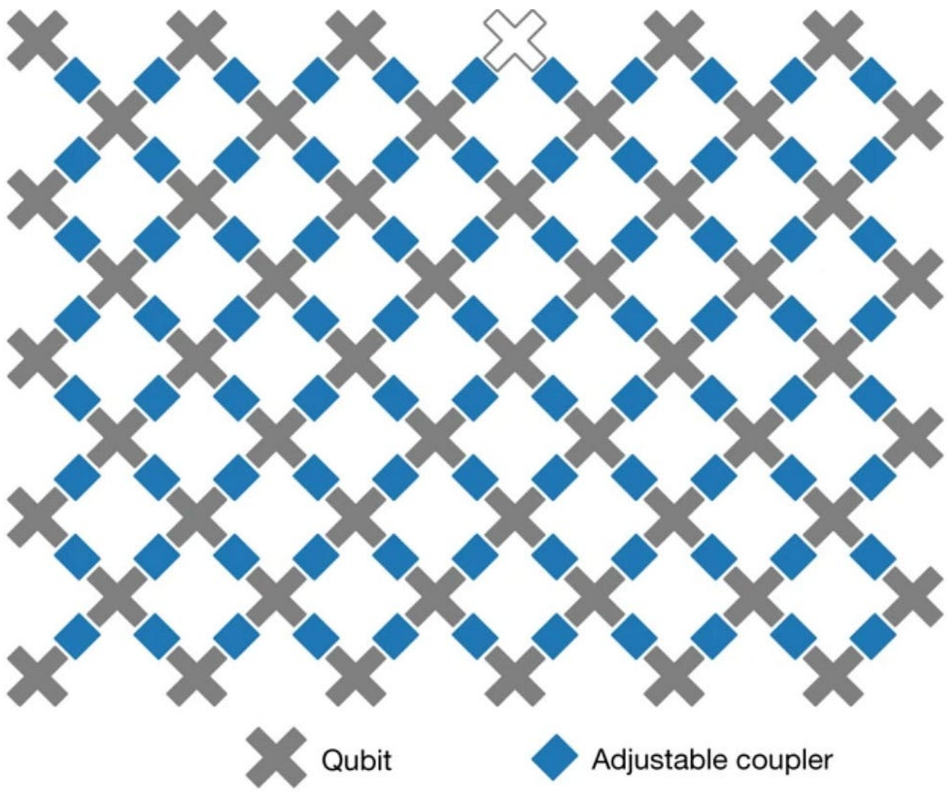
1. Low entangled state
    - Markov, Igor L., and Yaoyun Shi. “Simulating Quantum Computation by Contracting Tensor Networks.” SIAM Journal on Computing 38, no. 3 (January 2008): 963–81.
    - Kalachev, Gleb, Pavel Panteleev, and Man-Hong Yung. “Recursive Multi-Tensor Contraction for XEB Verification of Quantum Circuits,” 2021, 1–9.
2. Noisy limit
    - Gao, Xun, and Luming Duan. “Efficient Classical Simulation of Noisy Quantum Computation.” October 7, 2018. arXiv.1810.03176.
    - Shao, Yuguo, Fuchuan Wei, Song Cheng, and Zhengwei Liu. “Simulating Quantum Mean Values in Noisy Variational Quantum Algorithms: A Polynomial-Scale Approach.” July 20, 2023. arXiv.2306.05804.

Tensor network backend

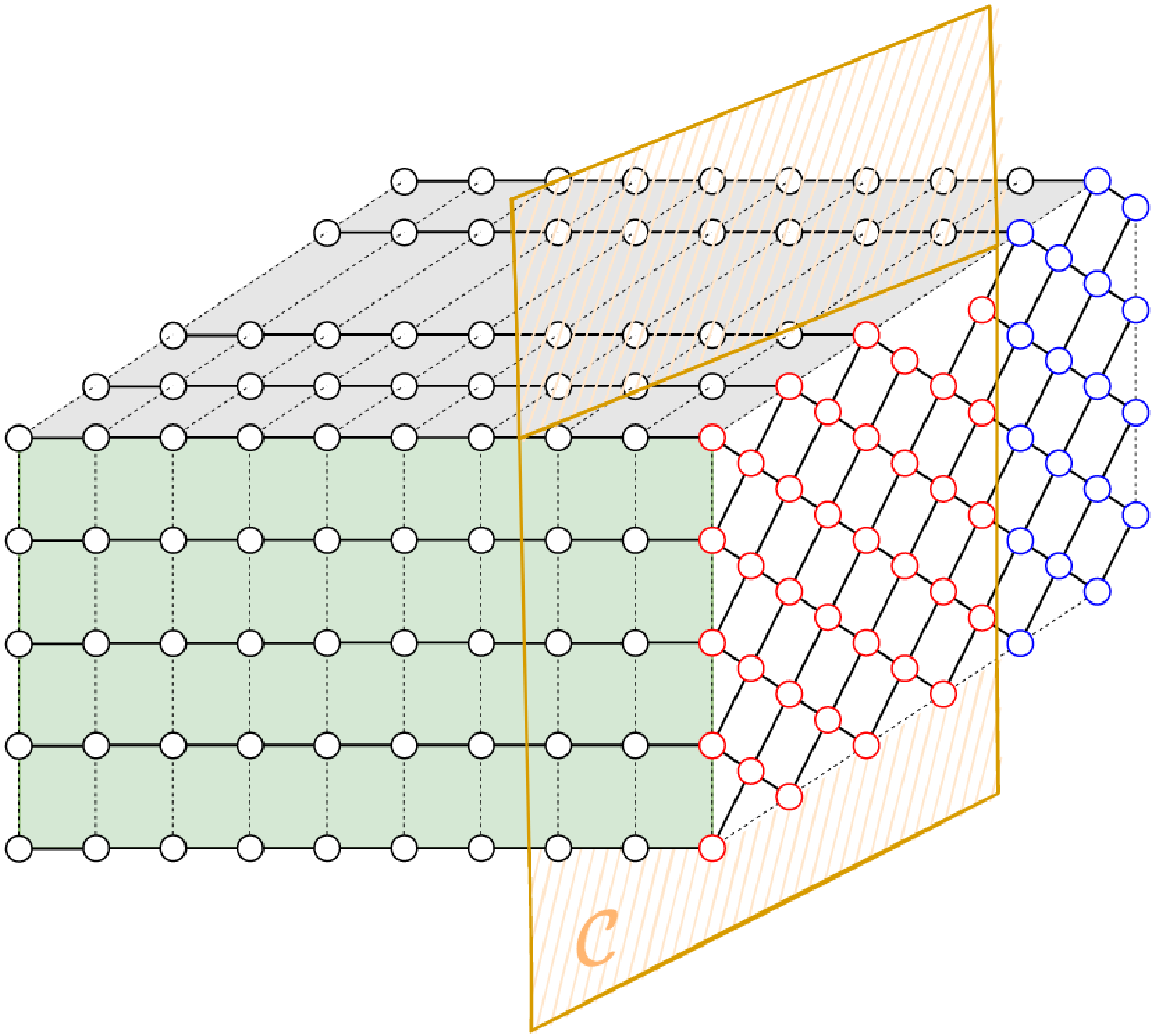
What is tensor network？ It is basically the same as `einsum` or sum-product network!

Tensor network is very good at simulating shallow circuit!

Google 53-qubit Sycamore processor: "The quantum system took approximately 200 seconds to execute a task that would have taken a classical computer around 10,000 years to complete."



Feng Pan et al.: "Nope! by contracting its tensor network representation on a single A100 GPU card, it costs only 149 Days.



References

- Arute, Frank, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, et al. “Quantum Supremacy Using a Programmable Superconducting Processor.” Nature 574, no. 7779 (October 2019): 505–10.
- Pan, Feng, and Pan Zhang. “Simulation of Quantum Circuits Using the Big-Batch Tensor Network Method.” Physical Review Letters 128, no. 3 (January 19, 2022): 030501.

YaoToEinsum

A state of the art tensor network backend for Yao, which supports

- Slicing (for reducing memory cost)
- GPU simulation

Including tests, there are 222 lines in total!

```
(base) ➜  YaoToEinsum git:(main) cloc .
      14 text files.
      14 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.02 s (513.3 files/s, 42217.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TOML                             2            103              1            406
Julia                            5             28             35            222
YAML                             3              0              0             71
TeX                              1              3              0             58
Markdown                         1             17              0             43
-------------------------------------------------------------------------------
SUM:                            12            151             36            800
-------------------------------------------------------------------------------
```

- It is based on OMEinsum (Andreas Peter, Google Summer of Code, 2019)
- `OMEinsum` is later extended with state of the art contraction order finding algorithms.
  - Recursive min-cut
  - Local search

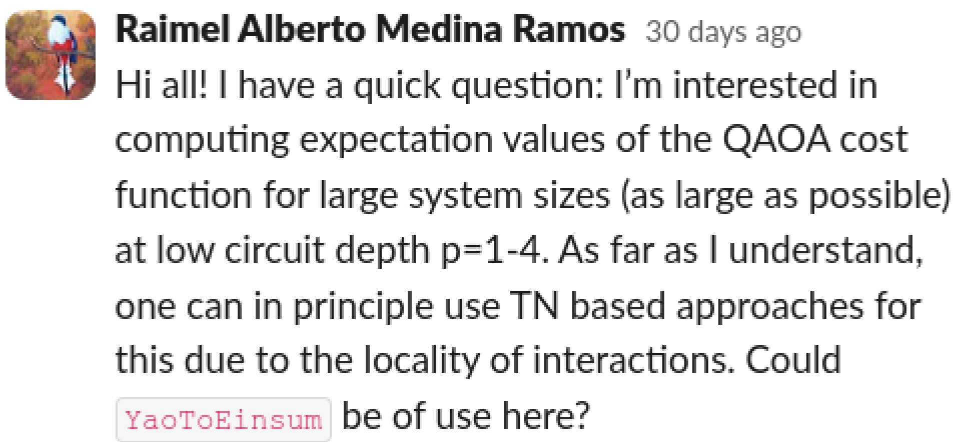
Please check Github repo OMEinsumContractionOrders.jl for more information.

References

- Gray, Johnnie, and Stefanos Kourtis. "Hyper-Optimized Tensor Network Contraction." ArXiv, 2020. https://doi.org/10.22331/q-2021-03-15-410.
- Kalachev, Gleb, Pavel Panteleev, and Man-Hong Yung. "Recursive Multi-Tensor Contraction for XEB Verification of Quantum Circuits," 2021, 1–9.

A demo using case

Julia slack > yao-dev



Raimel Alberto Medina Ramos 30 days ago

Hi all! I have a quick question: I'm interested in computing expectation values of the QAOA cost function for large system sizes (as large as possible) at low circuit depth p=1-4. As far as I understand, one can in principle use TN based approaches for this due to the locality of interactions. Could `YaoToEinsum` be of use here?

```
julia> @btime get_your_question_anwsered(
        "slack > yao-dev",
        "zulip > yao-dev",
        "discourse > quantum category"
    )
1 hour
```

Yao community call will be announced in the Julia slack.

Live coding

Source code available in file: `clips/yaotoeinsum.jl`

```
┌ Warning: target space complexity not found, got: 25.0, with time complexity 30.1
09897157269785, read-write complexity 27.016182724887575.
└ @ OMEinsumContractionOrders ~/.julia/packages/OMEinsumContractionOrders/WpwIz/sr
c/treesa.jl:229
TensorNetwork
Time complexity: 2^33.10989739632241
Space complexity: 2^25.0
Read-write complexity: 2^30.016188177563862

julia> # compute!

julia> contract(tensornetwork)
0-dimensional Array{ComplexF64, 0}:
1.078959321878884e-5 + 0.0im

julia> # Utilize the power of GPU

julia> using CUDA, BenchmarkTools

julia> # upload tensors to your CUDA device.

julia> cutensornetwork = cu(tensornetwork)
TensorNetwork
Time complexity: 2^33.10989739632241
Space complexity: 2^25.0
Read-write complexity: 2^30.016188177563862

julia> # the CPU version

julia> @btime contract($tensornetwork)
  46.989 s (296828 allocations: 16.20 GiB)
0-dimensional Array{ComplexF64, 0}:
1.078959321878884e-5 + 0.0im

julia> # the GPU version

julia> @btime CUDA.@sync contract($cutensornetwork)
  198.298 ms (1092329 allocations: 52.15 MiB)
0-dimensional CuArray{ComplexF64, 0, CUDA.Mem.DeviceBuffer}:
1.078959321878884e-5 + 0.0im

julia>
```

Density matrix based simulation

```
dm = DensityMatrix{2, ComplexF64, Array...}
        active qubits: 3/3
        nlevel: 2
```

```julia
# create a reduced density matrix on subsystem (1, 2, 3)
dm = density_matrix(ghz_state(5), 1:3)
```

```
8×8 Matrix{ComplexF64}:
 0.5+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im  …  0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0+0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0-0.0im  0.0+0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0-0.0im  0.0-0.0im     0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0-0.0im  0.0-0.0im  …  0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0-0.0im  0.0-0.0im     0.0-0.0im  0.0+0.0im  0.0+0.0im
 0.0-0.0im  0.0-0.0im  0.0-0.0im  0.0-0.0im     0.0-0.0im  0.0-0.0im  0.5+0.0im
```

```julia
# the density matrix is represented as a matrix - computational very inefficient
dm.state
```

```
0.6931471805599931
```

```julia
# the entanglement entropy
von_neumann_entropy(dm)
```

```
DensityMatrix{2, ComplexF64, Array...}
    active qubits: 3/3
    nlevel: 2
```

```julia
# apply a quantum gate X on the first qubit
apply(dm, put(3, 1=>X))
```

```
  BitBasis.DitStr{2, 3, Int64}[
     1:  000 ₍₂₎
     2:  111 ₍₂₎
     3:  000 ₍₂₎
     4:  000 ₍₂₎
     5:  000 ₍₂₎
]
```

```julia
# measure the density matrix
measure(dm; nshots=5)
```

```
  [110 ₍₂₎, 001 ₍₂₎, 001 ₍₂₎, 001 ₍₂₎, 001 ₍₂₎]
```

```julia
measure(apply(dm, put(3, 1=>X)); nshots=5)
```

```
1.0000000000000002 + 0.0im
```

```julia
# the expectation value of correlation Z₁Z₂
expect(kron(3, 1=>Z, 2=>Z), dm)
```

```
dpolarizing = nqubits: 1
              unitary_channel
              ├─ [0.925] I2
              ├─ [0.025] X
              ├─ [0.025] Y
              └─ [0.025] Z
```

```julia
# noises can be defined as a quantum unitary channel
dpolarizing = single_qubit_depolarizing_channel(0.1)
```

```
BitBasis.DitStr{2, 3, Int64}[
    1:  001 ₍₂₎
    2:  111 ₍₂₎
    3:  000 ₍₂₎
    4:  111 ₍₂₎
    5:  001 ₍₂₎
    6:  110 ₍₂₎
    7:  000 ₍₂₎
    8:  111 ₍₂₎
    9:  111 ₍₂₎
    10:  110 ₍₂₎
]
```

```
let
    dmk = dm
    # repeatedly apply depolarizing channel on the first qubit
    for i=1:100
        dmk = apply(dmk, put(3, 1=>dpolarizing))
    end
    # the first qubit becomes completely random
    measure(dmk; nshots=10)
end
```

More Examples

For more examples, please check QuAlgorithmZoo.

What is the next step?

1. Roger's research interest is more about quantum compling, he has written packages such as OpenQASM.jl. Roger Luo and Chen Zhao is working on YaoExpr.jl for the next generation of quantum compiling.
2. GiggleLiu will focus more on tensor network based simulation of density matrices.