Classical simulations of quantum circuits in the Pre-fault-tolerant era

Jed Pixley and Haining Pan

June 8, 2026

ICTS Lecture Series:
Quantum Dynamics in the pre-fault tolerant era

https://github.com/Pixley-Research-Group-in-CMT/QuantumCircuitsMPS.jl
https://github.com/Pixley-Research-Group-in-CMT/FSS

Lecture Series

I. Lecture 1: Classical and Quantum Chaos, from single particle to many-body

II. Lecture 2: Quantum platforms

III. Lecture 3: Monitored dynamics, interplay of unitary andprojective evolution

IV. Lecture 4: Adaptive dynamics, controlling quantum dynamics

V. Lecture 5: Numerical approaches to adaptive quantum dynamics

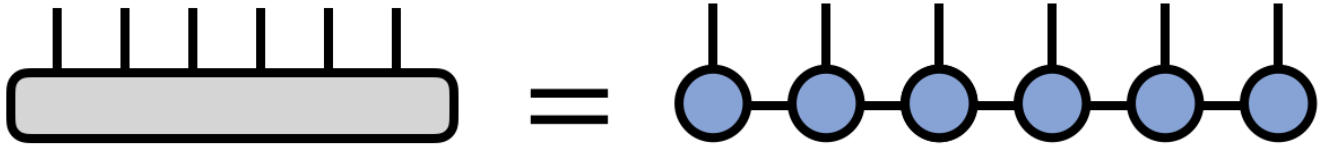
Learning Goal

- Understand matrix product states
- Numerically construct the circuits and simulate them
- Perform finite size scaling

https://github.com/Pixley-Research-Group-in-CMT/QuantumCircuitsMPS.jl
https://github.com/Pixley-Research-Group-in-CMT/FSS

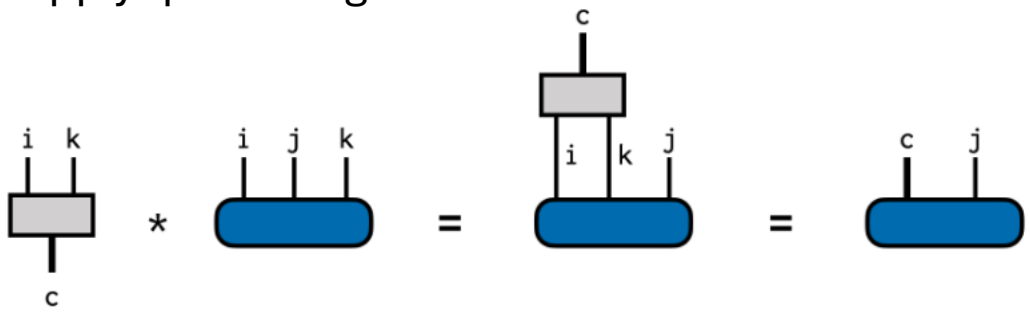
Matrix product states

Factorization of a tensor with N indices into a chain-like product of three-index tensors



$$T^{s_1 s_2 s_3 s_4 s_5 s_6} = \sum_{\{\alpha\}} A^{s_1}_{\alpha_1} A^{s_2}_{\alpha_1 \alpha_2} A^{s_3}_{\alpha_2 \alpha_3} A^{s_4}_{\alpha_3 \alpha_4} A^{s_5}_{\alpha_4 \alpha_5} A^{s_6}_{\alpha_5}$$

Apply quantum gates:



https://tensornetwork.org/
https://itensor.github.io

What is QuantumCircuitsMPS.jl

- Physicists code as we speak: focusing on physics without touching implementation details.
- Julia library for simulating quantum circuits using Matrix Product State (MPS) methods.
- For studying measurement-induced and control-induced phase transitions in monitored quantum systems.

Why use this package

- **Pure Julia MPS simulation**: Native Julia performance with ITensors.jl backend, scaling to 100+ qubits.
- **Physics-first API**: Write circuits using intuitive abstractions (Gates + Geometry) without managing MPS bond dimensions, index orderings, or truncation schemes. The library handles tensor network details internally.
- **Reproducible randomness**: Independent RNG streams for each source (:gates_spacetime, :gates_realization, :measurement_born, :state_init) enable reproducible trajectories. Useful in cross entropy benchmark and study quantum fluctuations.

Design Philosophy

Core logic:

```
1  apply!(state, gate, geometry)
```

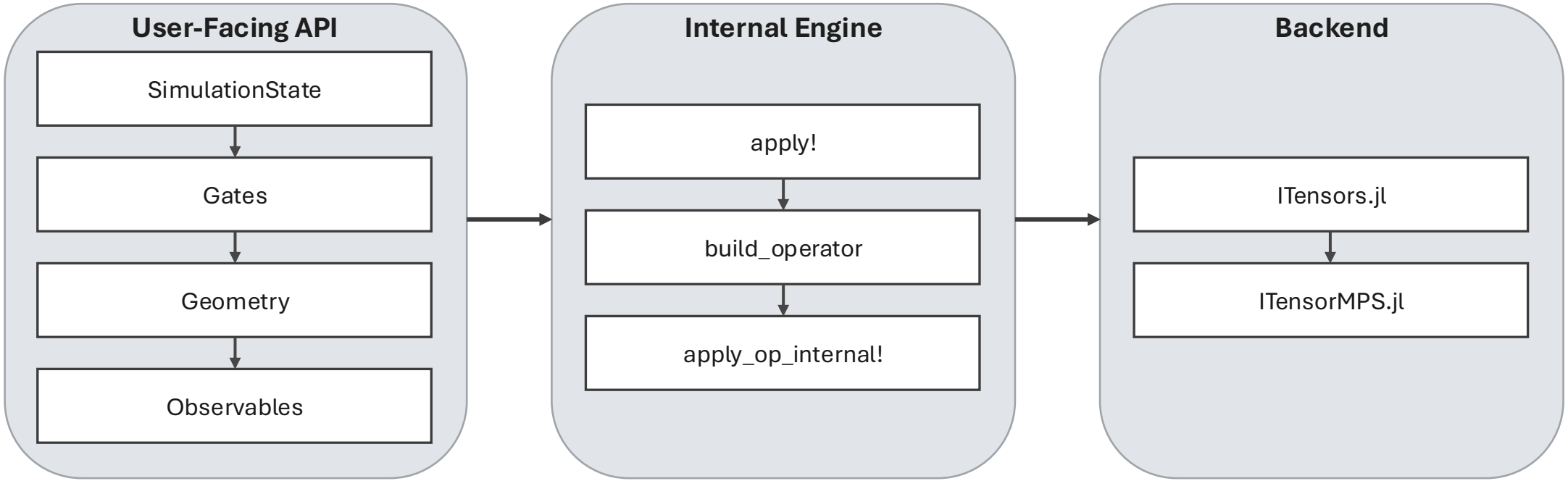
state : the MPS for the wave function
gate : quantum gates (e.g. HaarRandom())
geometry: geometry of the circuit (e.g., Bricklayer(:even), Bricklayer(:odd))

Users never see ITensor index objects, SVD calls, or orthogonalization centers.
The package manages the gap between high-level physics intent and low-level tensor manipulations.

Comparison with Existing Julia Quantum Libraries

Feature	ITensors.jl	PastaQ.jl	Yao.jl	Qiskit.jl	This Package
Primary Focus	Tensor networks	Tomography & benchmarking	Variational algorithms	Circuit construction	**MIPT/CIPT dynamics**
Backend	MPS/MPO (via ITensorMPS)	MPS/MPO	State vector (+ YaoToEinsum)	No simulation*	**MPS (via ITensors)**
MIPT/CIPT Support	Build from scratch	Manual logic	State vector limited	N/A	**First-class**
Scalability	N=100+	N=100+	~30 qubits	N/A	**N=100+**
API Level	Tensor-level	Circuit + Tomography	Block-level	Circuit construction	**Physics-level**
Learning Curve	Steep	Medium	Gentle	N/A	**Gentle**

Architecture



Physicists work with SimulationState, Gates, Geometry (Bricklayer, AllSites, StaircaseLeft), and Observables. No tensor network concepts exposed.

The apply! function **translates high-level physics operations into ITensor calls**. It manages physical-to-RAM index mappings, operator construction, and MPS updates. Users never interact with this layer.

ITensors.jl and **ITensorMPS**.jl handle tensor contractions, SVD truncations, and gauge management. All low-level optimizations (bond dimensions, cutoffs, orthogonality centers) are managed automatically.

Installation

QuantumCircuitsMPS.jl is not yet registered in the Julia General registry.

Install directly from GitHub:

```
using Pkg
Pkg.add(url="https://github.com/hainingpan/QuantumCircuitsMPS.jl")
```

MIPT Example: Measurement-Induced Phase Transition

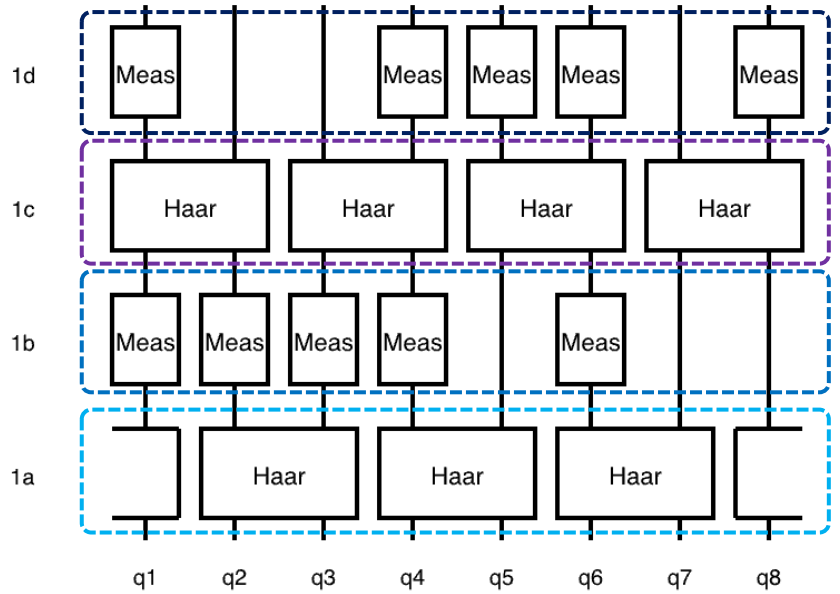
Define parameters

```
const L = 8                    # System size (number of qubits)
const bc = :periodic           # Boundary conditions
const n_steps = L              # Total timesteps for simulation (passed to simulate!(n_steps=n_steps))
const p = 0.5                  # Measurement probability (near critical p_c ≈ 0.16)
const cut = L ÷ 2              # Entanglement cut position

circuit = Circuit(L=L, bc=bc, p=p) do c
    apply!(c, HaarRandom(), Bricklayer(:even))
    apply_with_prob!(c; rng=:gates_spacetime, outcomes=[
        (probability=c.params[:p], gate=Measurement(:Z), geometry=AllSites())
    ])
    apply!(c, HaarRandom(), Bricklayer(:odd))
    apply_with_prob!(c; rng=:gates_spacetime, outcomes=[
        (probability=c.params[:p], gate=Measurement(:Z), geometry=AllSites())
    ])
end

plot_circuit(circuit; gates_spacetime=0, n_steps=1)
```

One time unit

Random seed for spacetime of gates

Time step: 1



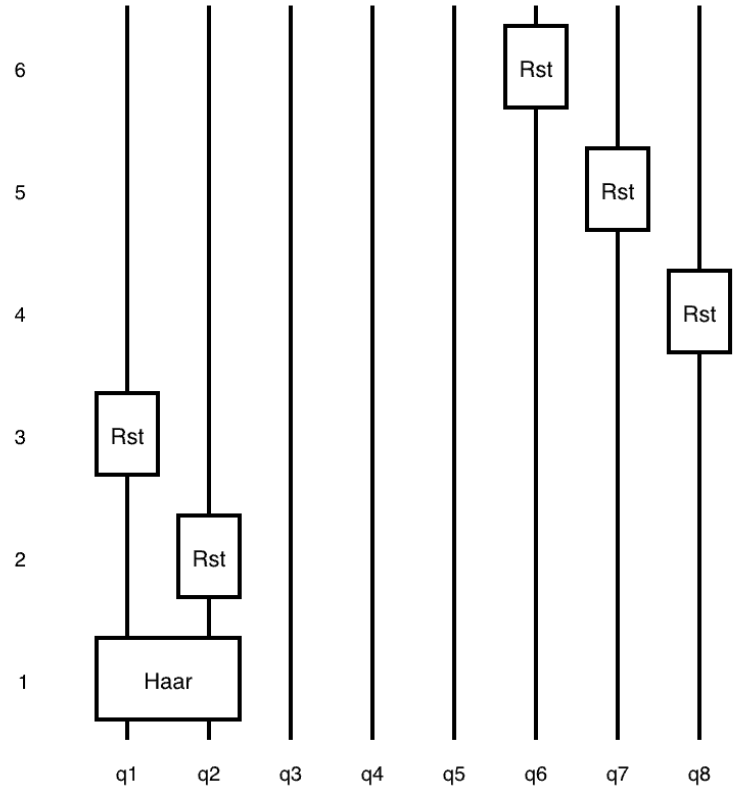
CIPT Example: Control-Induced Phase Transition

Define parameters

```julia
const L = 8                    # System size (number of qubits)
const bc = :periodic           # Boundary conditions
const n_steps = 2 * L^2        # Total timesteps (staircase sweeps)
const p_ctrl = 0.5             # Control probability

left = StaircaseLeft(1)
right = StaircaseRight(1)

circuit = Circuit(L=L, bc=bc, p_ctrl=p_ctrl) do c
    apply_with_prob!(c; rng=:gates_spacetime, outcomes=[
        (probability=c.params[:p_ctrl], gate=Reset(), geometry=left),
        (probability=1-c.params[:p_ctrl], gate=HaarRandom(), geometry=right)
    ])
end

plot_circuit(circuit; gates_spacetime=42, n_steps=6)
```

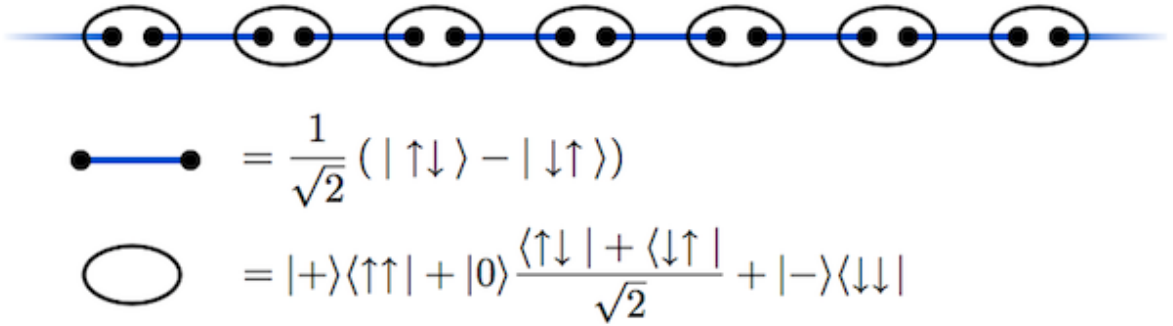
Choose Reset

Choose Unitary



Affleck–Kennedy–Lieb–Tasaki (AKLT) model

Spin-1 chain (i.e., $S = 1$ compared to the qubit $S = 1/2$)



Credit: https://en.wikipedia.org/wiki/AKLT_model

Protocol:
Measure the total spin of each nearest neighbor spin-1 pair (i.e., post-select the outcome of $S \neq 2$)

String order parameter

$$O^{\alpha}(i,j) = \langle S_i^{\alpha} \exp\left(i\pi \sum_{k=i+1}^{j-1} S_k^{\alpha}\right) S_j^{\alpha} \rangle = -\frac{4}{9}$$

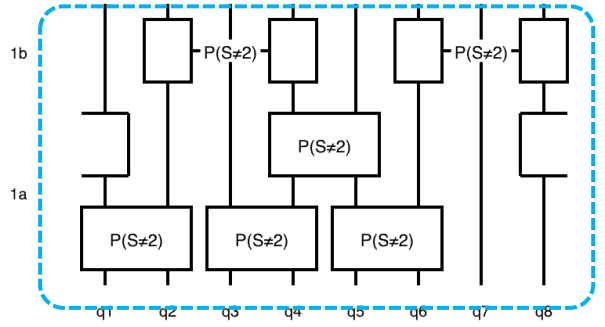
AKLT, PRL
Den Nijs & Rommelse, PRB

AKLT Example: Nearest Neighbor—to— next NN

Define parameters

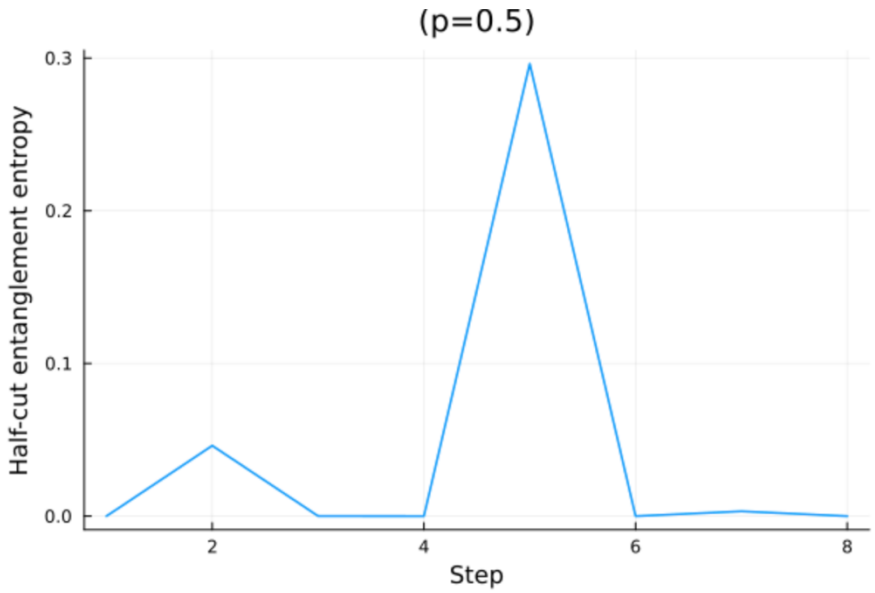
```julia
L = 8              # System size (spin-1 sites, divisible by 4 for NNN coverage)
bc = :periodic     # Boundary conditions
n_layers = L       # Number of projection layers
p_nn = 0.9         # Probability of NN projection (1 - p_nn for NNN)
maxdim = 128       # Maximum bond dimension

P0 = total_spin_projector(0)
P1 = total_spin_projector(1)
proj_gate = SpinSectorProjection(P0 + P1)

# One layer per do-block execution: NN bricklayer w.p. p_nn, NNN bricklayer w.p. 1-p_nn
circuit = Circuit(L=L, bc=bc, p_nn=p_nn, proj_gate=proj_gate) do c
    apply_with_prob!(c; rng=:gates_spacetime, outcomes=[
        (probability=c.params[:p_nn], gate=c.params[:proj_gate], geometry=Bricklayer(:nn)),
        (probability=1-c.params[:p_nn], gate=c.params[:proj_gate], geometry=Bricklayer(:nnn))
    ])
end

plot_circuit(circuit; gates_spacetime=3, n_steps=1)
```

Define forced measurement



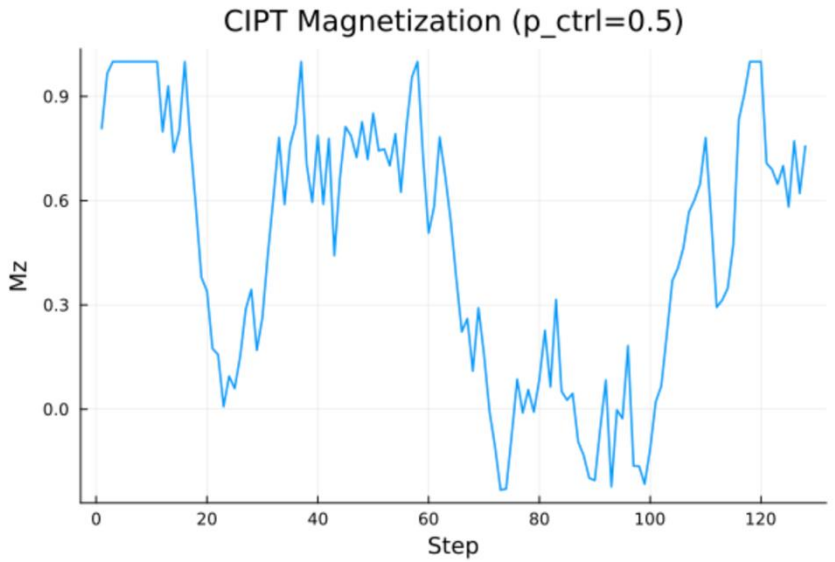
Simulation of MIPT – Single trajectory

```julia
state = SimulationState(
    L=L,
    bc=bc,
    rng=RNGRegistry(gates_spacetime=0, born_measurement=0, gates_realization=2)
)

# Initialize to product state |0⟩⊗L
initialize!(state, ProductState(binary_int=0))

# Track entanglement entropy at the central cut
track!(state, :entropy => EntanglementEntropy(; cut=cut))

# Run simulation: execute circuit n_steps times (n_steps=n_steps)
simulate!(circuit, state; n_steps=n_steps, record_when=:every_step)

# Extract entropy values from state
entropy_vals = state.observables[:entropy]

plot(entropy_vals, xlabel="Step", ylabel="Half-cut entanglement entropy",
     title="(p=$p)", legend=false, lw=1.5)
```



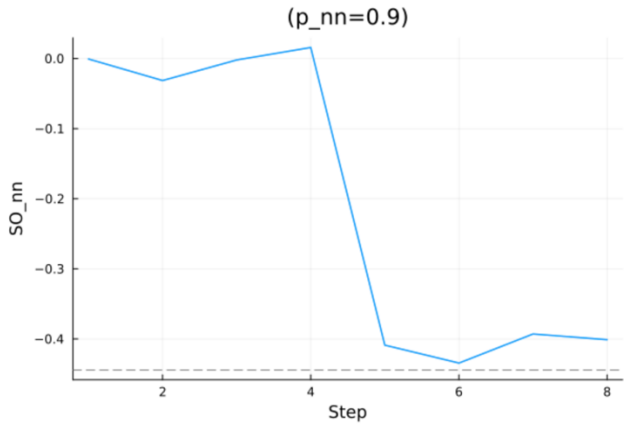
Simulation of CIPT – Single trajectory

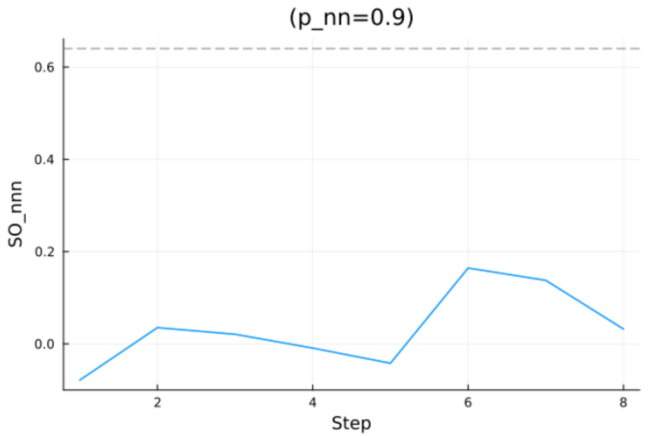
```julia
# Create simulation state with RNG registry
state = SimulationState(
    L=L,
    bc=bc,
    maxdim=64,
    rng=RNGRegistry(gates_spacetime=42, born_measurement=1, gates_realization=2)
)

# Initialize to product state |0>^L
initialize!(state, ProductState(binary_int=0))

# Track magnetization
track!(state, :Mz => Magnetization(:Z))

# Run simulation: n_steps controls how many times the circuit do-block runs
# record_when=:every_gate records after each gate (1 gate per step)
simulate!(circuit, state; n_steps=n_steps, record_when=:every_gate)

# Extract magnetization values
mz_vals = state.observables[:Mz]
```

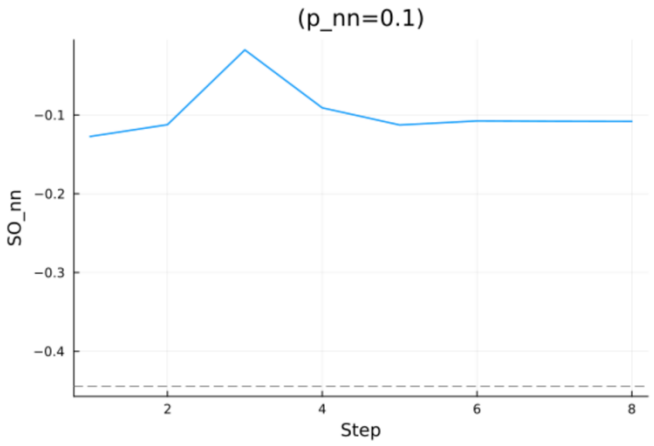


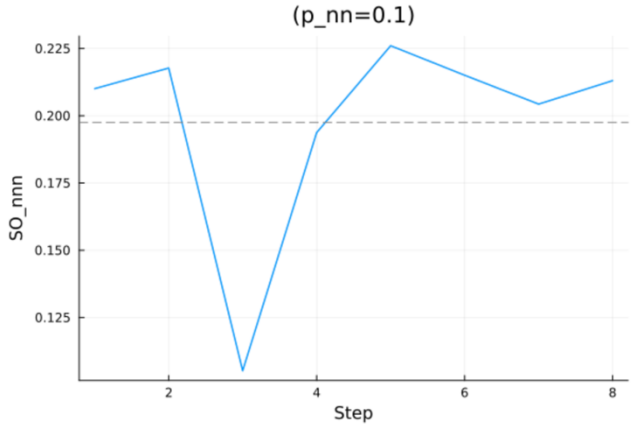
Simulation of AKLT – Single trajectory

```julia
state = SimulationState(
    L=L, bc=bc, site_type="S=1", maxdim=maxdim,
    rng=RNGRegistry(
        gates_spacetime=3,
        born_measurement=1,
        gates_realization=2
    )
)
initialize!(state, ProductState(spin_state="Z0"))

track!(state, :SO_nn => StringOrder(1, L÷2+1, order=1))
track!(state, :SO_nnn => StringOrder(1, L÷2+1, order=2))

simulate!(circuit, state; n_steps=n_layers, record_when=:every_step)
```









Phase diagram in CIPT

Create a wrapper function

```julia
function run_cipt(; L, p_ctrl, seed, bc=:periodic, n_steps=L^2, maxdim=2^20)
    left = StaircaseLeft(1)
    right = StaircaseRight(1)

    circuit = Circuit(L=L, bc=bc, p_ctrl=p_ctrl) do c
        apply_with_prob!(c; rng=:gates_spacetime, outcomes=[
            (probability=c.params[:p_ctrl], gate=Reset(), geometry=left),
            (probability=1-c.params[:p_ctrl], gate=HaarRandom(), geometry=right)
        ])
    end

    state = SimulationState(L=L, bc=bc, maxdim=maxdim,cutoff = 1e-6,
        rng=RNGRegistry(gates_spacetime=seed, born_measurement=seed+100, gates_realization=seed+200))
    initialize!(state, ProductState(binary_int=0))
    track!(state, :Mz => Magnetization(:Z))

    simulate!(circuit, state; n_steps=n_steps, record_when=:final_only)
    return state.observables[:Mz][end]
end
```

Define circuits

Simulate circuits

Phase diagram in CIPT

Define parameter lists

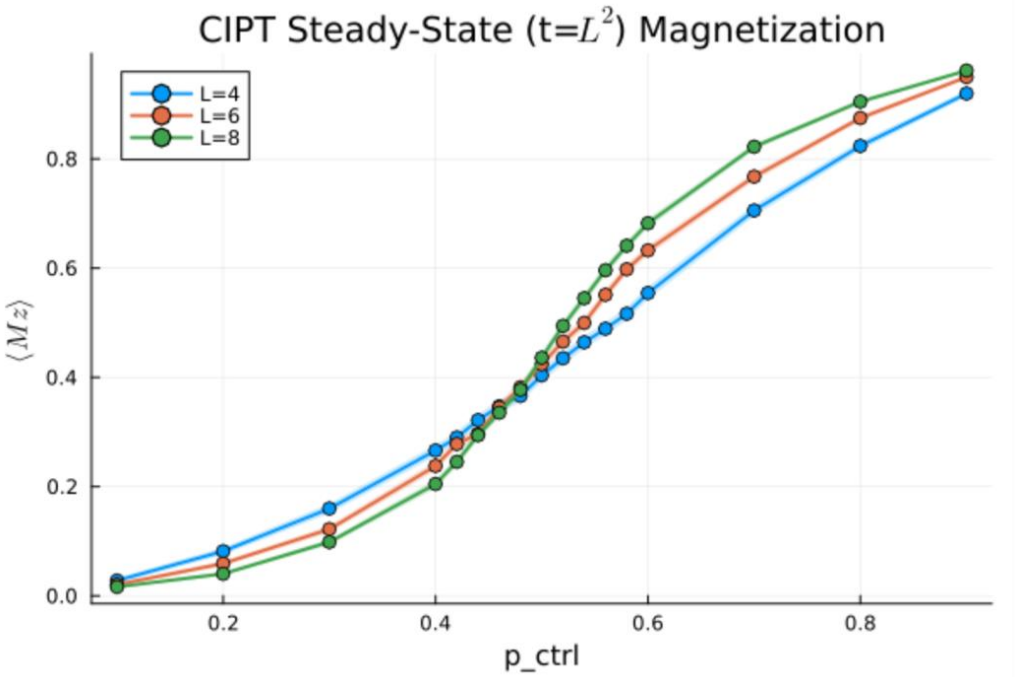
```julia
L_list = [4,6,8]
# Coarse grid over full range + fine grid near the critical point p_c = 0.5
p_list = sort(union(0.1:0.1:0.9, 0.4:0.02:0.6))
ensemble_size = 1000
```

```julia
configs = [(L=L, p=p, seed=s) for L in L_list for p in p_list for s in 1:ensemble_size]
raw = Vector{Float64}(undef, length(configs))

# Run with `julia -t auto` for multithreaded execution
println("Running $(length(configs)) configs on $(Threads.nthreads()) threads...")
prog = Progress(length(configs))
Threads.@threads for i in eachindex(configs)
    c = configs[i]
    raw[i] = run_cipt(L=c.L, p_ctrl=c.p, seed=c.seed)
    next!(prog)
end
finish!(prog)
```

Iteration

```julia
# Reshape to (seed, p, L) and average over seeds
ns, np, nL = ensemble_size, length(p_list), length(L_list)
Mz_raw = reshape(raw, ns, np, nL)
Mz_mean = dropdims(mean(Mz_raw, dims=1), dims=1)
Mz_sem  = dropdims(std(Mz_raw, dims=1), dims=1) ./ sqrt(size(Mz_raw, 1))
```

Reshape



Finite size scaling

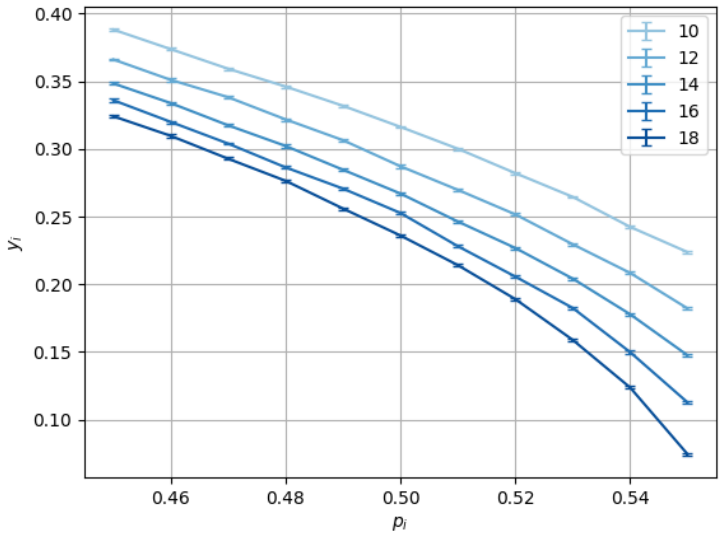
Installation

```
pip install git+https://github.com/hainingpan/FSS.git
```

$$y(p, L) \sim L^{-\beta/\nu} f((p - p_c)L^{1/\nu})$$

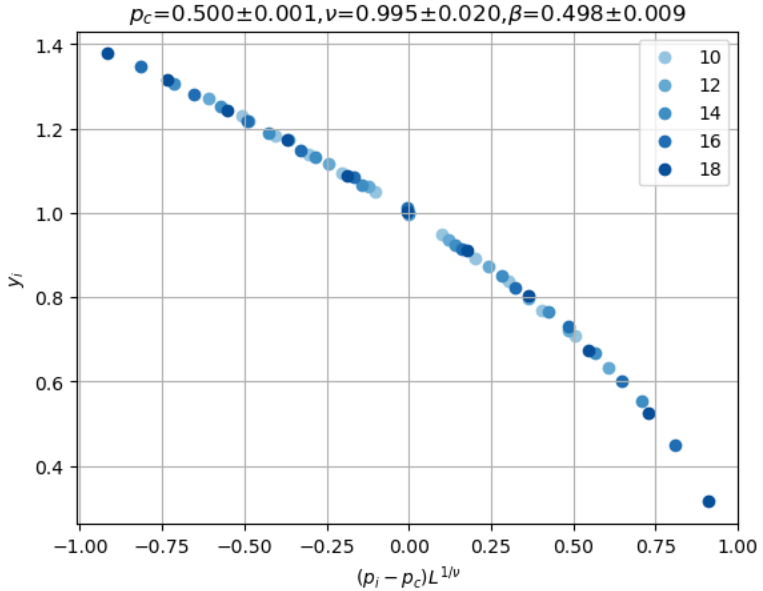
where:

- p: tuning parameter; L: system size; p_c: critical point
- ν: correlation-length exponent; β: scaling exponent of y
- $f(\cdot)$: unknown universal scaling function



Collapse rescales

$$x = (p - p_c)L^{1/\nu}, \quad y_{\text{scaled}} = yL^{\beta/\nu}$$




```
from fss import DataCollapse
dc = DataCollapse(df, p_='p', L_='L', params={}, p_range=[0.45, 0.55])
res = dc.datacollapse(p_c=0.501, nu=1.0, beta=0.0, p_c_vary=True, nu_vary=True, beta_vary=True)
```

Finite size scaling: CIPT

Load data

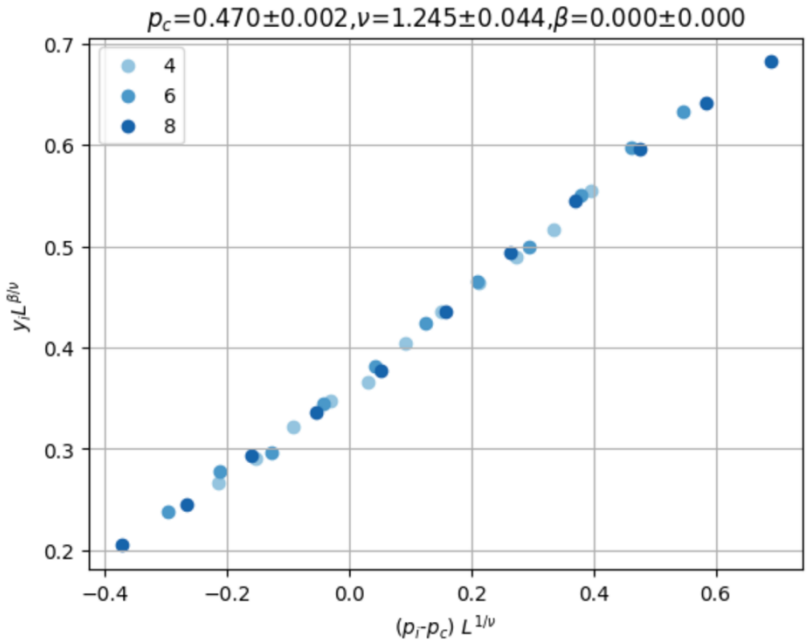
```python
import pandas as pd
from fss import DataCollapse

df = (pd.read_csv("cipt_Mz_data.csv", comment="#")
        .rename(columns={"Mz_mean": "estimator", "Mz_sem": "standard_error"})
        .set_index(["p", "L"])
        .sort_index())

dc = DataCollapse(df, p_="p", L_="L", p_range=[0.4, 0.6], estimator="manual")
```

Data collapse

```python
res = dc.datacollapse(p_c=0.491, nu=1.0, beta=0.0,
                      p_c_vary=True, nu_vary=True, beta_vary=False)
```

Visualization

```python
dc.plot_data_collapse()
```



Citation

```
@software{quantumcircuitsmps,
author = {Pan, Haining and Pixley, Jedediah H},
title = {QuantumCircuitsMPS.jl: MPS-based Quantum Circuit Simulation},
url = {https://github.com/hainingpan/QuantumCircuitsMPS.jl},
year = {2026}
}
```

```
@software{pan2025fss,
  author       = {Pan, Haining and Pixley, Jedediah H},
  title        = {FSS: Finite-Size Scaling Toolkit},
  year         = {2025},
  publisher    = {GitHub},
  journal      = {GitHub repository},
  url          = {https://github.com/hainingpan/FSS},
  version      = {0.0.6}
}
```

⚠ Packages under active development

Question/Bugs/Feature requests

Raise an issue at Github repos