Tensor network decoding for quantum error correction code

In this tutorial, we will use the tensor network to decode quantum error correction code with TensorQEC.jl, which is a package contains multiple error correction methods, including

- integer programming decoder
- tensor net work decoder
- belief propagation decoder (including BPOSD)
- ...

It could serve as a good starting point to benchmark different QEC decoding algorithms.

```
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
```

```
# `TensorQEC` utilizes the tensor network to study the properties of quantum error correction.
# `Yao` is a quantum simulator.
# `OMEinsum` is a tensor network contraction engine.
# `PlutoUI` is for control gadgets, e.g. the checkboxes
using TensorQEC, Yao, OMEinsum, Random, PlutoUI
```

Table of Contents

Tensor network decoding for surface code

Decoding is a process to extract the error pattern from the syndrome. So, we will start from generating a syndrome for a quantum code.

Step 1: define the code and error model

As a first example, we use the d=3 surface code, and the error model is independent depolarizing errors on each qubit.

```
surface3 = ▸ SurfaceCode(3, 3)
```

```
surface3 = SurfaceCode(3,3)
```

For covenience, we use the Tanner graph represetation of QEC codes in the following, as it offers a more convenient framework for syndrome extraction and decoding.

```
# `CSSTannerGraph` returns the tanner graph for a CSS quantum code.
tanner = CSSTannerGraph(surface3);
```

```
error_model =
▸ IndependentDepolarizingError([0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05], [0.05, 0.
```

```
# `iid_error` generates independent errors on each qubit
# The first three arguments are the error probabilities for X, Y, and Z errors, respectively.
# The last argument is the number of qubits.
error_model = iid_error(0.05, 0.05, 0.05, 9)
```

Then we generate a random error pattern

```
error_pattern = YIIIIIIII
```

```
# `random_error_pattern` generates a random error pattern from the error model.
error_pattern = (Random.seed!(2); random_error_pattern(error_model))
```

In practise, we will not see these error patterns, instead, we will get the error syndrome through measurements.

```
syndrome = ▸CSSSyndrome([1₂, 0₂, 0₂, 0₂], [0₂, 0₂, 1₂, 0₂])
```

```
# `syndrome_extraction` takes a error pattern and a taner graph, returns the syndrome.
syndrome = syndrome_extraction(error_pattern, tanner)
```

The goal is to infer the error pattern or its equivalent form from the above syndrome. Here, two error patterns are "equivalent", if and only if they can be reduced to each other by applying some stabilizers.

Step 2: tensor network representation

The tensor network representation can be generated with the `compile` function, with the `TNMMAP` as the first argument. TNMMAP means Tensor Network based Marginal Maximum A Posteriori decoder.

```
# - `TreeSA()` is the optimizer for optimizing the tensor network contraction order.
compiled_decoder = compile(
    TNMMAP(; optimizer=TreeSA()),  # tensor network based decoder (Piveteau2024)
    tanner,
    error_model
);
```

The contraction order of the tensor network is optimized by the optimizer `decoder.optimizer`, the default optimizer is `TreeSA()` and the optimal contraction order is stored in `compiled_decoder.code`.

```
1∘5∘9∘28, 1∘27∘5∘9 -> 27∘28
├─ 1∘5∘9∘28
└─ 1∘8∘27∘18∘5, 8∘18∘9 -> 1∘27∘5∘9
   ├─ 17∘18∘13∘1∘5, 5∘13∘8∘18∘27∘17 -> 1∘8∘27∘18∘5
   │  ├─ 14∘15∘17∘18, 13∘1∘15∘5∘14 -> 17∘18∘13∘1∘5
   │  │  ├─ 20, 14∘15∘17∘18∘20 -> 14∘15∘17∘18
   │  │  │  ├─ 20
   │  │  │  └─ 14∘15∘17∘18∘20
   │  │  └─ 2∘13∘14∘1, 15∘2∘14∘5 -> 13∘1∘15∘5∘14
   │  │     ├─ 1∘11∘2, 11∘13∘14∘1 -> 2∘13∘14∘1
   │  │     │  ⋮
   │  │     │
   │  │     └─ 15∘2∘5, 5∘14 -> 15∘2∘14∘5
   │  │        ⋮
   │  │
   │  └─ 17∘16∘5∘13∘8, 16∘17∘18∘27 -> 5∘13∘8∘18∘27∘17
   │     ├─ 8∘17, 16∘5∘8∘13 -> 17∘16∘5∘13∘8
   │     │  ├─ 8∘17
   │     │  └─ 13∘7∘16, 13∘5∘7∘8 -> 16∘5∘8∘13
   │     │     ⋮
   │     │
   │     └─ 16∘17∘18∘27
   └─ 8∘9, 9∘18 -> 8∘18∘9
      ├─ 26, 8∘9∘26 -> 8∘9
      │  ├─ 26
      │  └─ 8∘9∘26
      └─ 9∘18
```

```
# The tensor network topology
compiled_decoder.code
```

The output is associated with the open indices at the top level, which is [27, 28]. They correspond to the marginal probabilities of logical X and Z flip.

```
27
```

```
# The tensor network data, here we have 27 tensors
compiled_decoder.tensors |> length
```

```
Time complexity: 2^9.894817763307945
Space complexity: 2^6.0
Read-write complexity: 2^10.066089190457774
```

```
# 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(compiled_decoder)
```

Step 3: decode

```
decoding_result = DecodingResult(true, XZIZZIZIZ)
```

```
# `decode` function takes a compiled decoder and a syndrome, returns the decoding outcome. We will see what is actully happenes in this decode function.
# The decoder saves the deduced error pattern in `docoding_result.error_pattern`.
decoding_result = decode(compiled_decoder, syndrome)
```

```
XZIZZIZIZ
```

```
decoding_result.error_pattern
```

We can check whether the decoding result matches the syndrome.

```
true
```

```
syndrome == syndrome_extraction(decoding_result.error_pattern, tanner)
```

Tensor network decoder, explained

We firstly update the syndrome in the tensors of the tensor network and compute the probability of different logical sectors by tensor network contraction.

```
TensorQEC.update_syndrome!(compiled_decoder, syndrome);
```

```
marginal_probability = 2×2 Matrix{Float64}:
                       0.00174541  0.014687
                       4.50585e-5  0.000231682
```

```
# the contraction result is the marginal probabilities on lx and lz
# p(no logical flip) = 0.00174541
# p(logical Z flip) = 0.014687
# p(logical X flip) = 4.50585e-5
# p(logical X and Z flip) = 0.000231682
marginal_probability = compiled_decoder.code(compiled_decoder.tensors...)
```

Given this marginal probability, we can determine the logical information and further identify an error pattern corresponding to this logical sector.

```
(0.014687, CartesianIndex(1, 2))
```

```
# find the Cartesian coordinate of the most likely logical error
_, pos = findmax(marginal_probability)
```

```
XZIZZIZIZ
```

```
# Infer error pattern from the logical error.
# To correct errors on a physical device, just apply the same error pattern
TensorQEC.error_pattern(pos, compiled_decoder, syndrome)
```

Circuit-level Quanutm Error Correction Decoding Problem

In quantum error correction, circuit-level decoding refers to the challenge of accurately identifying and correcting errors that occur during the execution of a quantum circuit, where errors may arise from imperfect gates, measurements, or idle qubit storage. Unlike idealized noise models, circuit-level noise incorporates realistic spatial and temporal correlations, making decoding more complex due to the interplay of gate errors, leakage, and crosstalk.

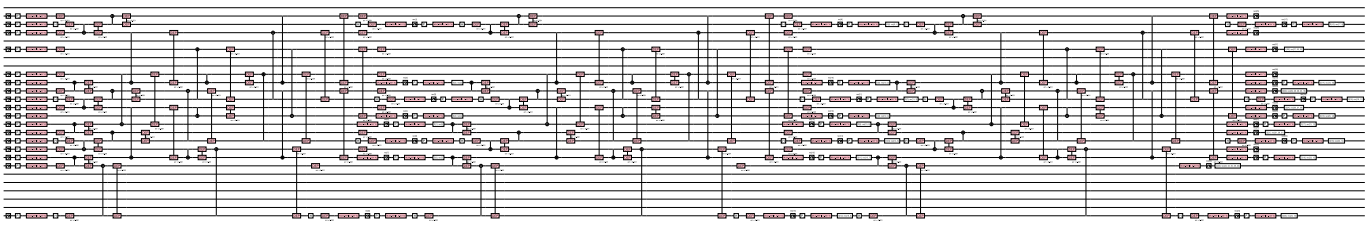
Load data

The quantum circuits and the corresponding detector error model is placed under the `data` folder. Here we load the syndrome measurement circuit of code distance `d=3`, error correction cycle `r=3` surface code and the corresponding detector error model. It is generated with stim package.

```
import stim

for d in [3, 5, 7, 9]:
    circuit = stim.Circuit.generated(
        "surface_code:rotated_memory_z",  # (rotated) surface code. z means the cir
cuit initializes and measures the logical Z basis observable.
        rounds=d,                         # number of measurements rounds
        distance=d,
        after_clifford_depolarization=0.001,  # depolarizing errors
        after_reset_flip_probability=0.001,
        before_measure_flip_probability=0.001,
        before_round_data_depolarization=0.001)  # before each round
    circuit.to_file(f"data/surface_code_d={d}_r={d}.stim") # stim file
    dem = circuit.detector_error_model(flatten_loops=True)
    dem.to_file(f"data/surface_code_d={d}_r={d}.dem")  # dem file
```

```
# stim file stores the circuit
qc = parse_stim_file(joinpath(@__DIR__, "data", "surface_code_d=3_r=3.stim"), 26);
```



```
# red boxes are error channels
# Hint: to zoom the circuit, please right click and open it in a new tab
vizcircuit(qc)
```

- The circuit first measure and reset the ancilla qubits to state 0. The unmeasured lines represents the data qubits.
- After each gate, depolarizing error is added.
- Intermediate measurement outcomes are stored into the recorder, annotated by `rec[k]`
- Detectors checks the syndromes, it checks the recorder and computes the XOR of the annotated records.

dem =

Error Index	Probability	Detectors	Logicals
1	0.00193118	[1]	Int64[]
2	0.00193118	[1, 2]	Int64[]
3	0.0025292	[1, 9]	Int64[]
4	0.00193118	[2, 3]	Int64[]
5	0.000400053	[2, 5, 6]	Int64[]
6	0.000133387	[2, 5, 9]	Int64[]
7	0.000799787	[2, 5]	[25]
8	0.00266191	[2, 6]	Int64[]
9	0.000400053	[2, 9]	Int64[]
10	0.00411959	[2]	[25]
⋮	⋮	⋮	⋮

209 rows omitted

```
# dem file stores the detector error model, which can be used to sample the errors and decode. This model is usually obtained from Clifford circuit simulation.
# col 1: error index
# col 2: error probabilities with i.i.d assumption
# col 3: which detectors are flipped
# col 4: which logical operators are flipped
dem = TensorQEC.parse_dem_file(joinpath(@__DIR__, "data", "surface_code_d=3_r=3.dem"))
```

Generate the tensor network

```
compiled_dem_decoder = compile(TNMMAP(; optimizer=TreeSA(ntrials=1)), dem);
```

```
Time complexity: 2^19.64533475781897
Space complexity: 2^13.0
Read-write complexity: 2^18.04375357632928
```

```
contraction_complexity(compiled_dem_decoder)
```

syndrome_dem = ▸SimpleSyndrome([0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, ⋯ more ,0_2])

```
# Generate an error pattern and the corresponding syndrome.
syndrome_dem = let
    Random.seed!(2)
    error_pattern = random_error_pattern(IndependentFlipError(dem.error_rates))
    syndrome_extraction(error_pattern, compiled_dem_decoder.tanner)
end
```

```
# update the tensor network
TensorQEC.update_syndrome!(compiled_dem_decoder, syndrome_dem);
```

▸[0.842622, 9.53539e-8]

```
compiled_dem_decoder.code(compiled_dem_decoder.tensors...)
```

Challenge: Tensor network decoder for [[144,12,12]] BB Code.

The circuit level decoder for BB code is notoriously hard problem for tensor network decoders. Here, we load the dem file from https://github.com/quantumlib/tesseract-decoder/tree/main/testdata/bivariatebicyclecodes. The belief propagation based approach is efficient, but the accuracy is not enough. The integer programming based approach is only efficient when the error rate is low enough (check below). For additional decoders applicable to this example, please refer to (Beni2025).

Goal: Develop a decoder that achieves both high accuracy and computational efficiency.

dem_bb =

Error Index	Probability	Detectors	Logicals
1	0.00385766	[1, 2, 57]	Int64[]
2	0.00385766	[1, 6, 56]	Int64[]
3	6.66978e-5	[1, 7, 16, 23, 29, 32, 103, 125]	[1731]
4	0.000233388	[1, 7, 16, 23, 29, 32, 103]	[1731]
5	6.66978e-5	[1, 7, 16, 23, 29, 32, 125]	[1731]
6	0.000233388	[1, 7, 16, 23, 29, 32]	[1731]
7	6.66978e-5	[1, 7, 16, 57, 63, 72, 87, 103]	[1737, 1739, 1740]
8	0.000233388	[1, 7, 16, 57, 63, 72, 87]	[1737, 1739, 1740]
9	6.66978e-5	[1, 7, 16, 57, 63, 72, 103]	[1737, 1739, 1740]
10	0.000233388	[1, 7, 16, 57, 63, 72]	[1737, 1739, 1740]
⋮	⋮	⋮	⋮

67814 rows omitted

```
dem_bb = TensorQEC.parse_dem_file(joinpath(@__DIR__, "data",
"r=12,d=12,p=0.001,noise=si1000,c=bivariate_bicycle_X,nkd=
[[144,12,12]],q=288,iscolored=True,A_poly=x^3+y+y^2,B_poly=y^3+x+x^2.dem"))
```

```
compiled_dem_decoder_bb = compile(TNMMAP(; optimizer=TensorQEC.NoOptimizer()),
dem_bb); # Here we use the NoOptimizer to avoid any optimization. Since the code is
too large, the default optimizer will be too slow.
```

585004

```
length(compiled_dem_decoder_bb.code.ixs)
```

It has ~585k tensors! Can you come up with a tensor network based decoder for it?

The performance of integer programming decoder

syndrome_bb = ▸SimpleSyndrome([0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, 0_2, ⋯ more ,0_2])

```
syndrome_bb = let
    Random.seed!(110)
    error_pattern = random_error_pattern(dem_bb)
    syndrome_extraction(error_pattern, compiled_dem_decoder_bb.tanner)
end
```

integer_prog_solve = ☑

true

```
if integer_prog_solve
    # decode with a integer programming decoder
    @time ip_res = decode(
        IPDecoder(),  # integer programming backend
        compiled_dem_decoder_bb.tanner,
        syndrome_bb
    )

    # test weather we get a same syndrome
    syndrome_bb == syndrome_extraction(
        ip_res.error_pattern,
        compiled_dem_decoder_bb.tanner
    )
end
```

```
35.282128 seconds (9.42 M allocations: 292.942 GiB, 8.77% gc time)
```

Integer programming decoder takes about 30 seconds. Can you beat it?

References

- **(Piveteau2024)** Piveteau, C.; Chubb, C. T.; Renes, J. M. Tensor Network Decoding Beyond 2D. PRX Quantum 2024, 5 (4), 040303. https://doi.org/10.1103/PRXQuantum.5.040303.
- **(Beni2025)** Beni, L. A.; Higgott, O.; Shutty, N. Tesseract: A Search-Based Decoder for Quantum Error Correction. arXiv March 14, 2025. https://doi.org/10.48550/arXiv.2503.10988.