Institute for Theoretical Physics
University of Cologne

Prof. Dr. S. Trebst
B. Han, Q. Preiss, M. Yutushui

Quantum Computational Physics

Exercise Sheet 5

Summer Term 2026

Due date: Monday, 22.06.2026 **Discussion**: Tuesday, 23.06.2026

Website: thp.uni-koeln.de/trebst/Lectures/2026-QuantCompPhys.shtml

This exercise sheet is all about **quantum error correction**. So, if you are interested in the computational details of practical quantum error correction implementations – and how classical computers can be crucial helpers for quantum computers – this exercise sheet is for you! If you are eager to learn more feel free to take a look at Quantum Error Correction For Dummies (https://arxiv.org/abs/2304.08678) and visit the Error Correction Zoo (https://errorcorrectionzoo.org) for even more quantum error correction stuff!

This exercise sheet involves some more advanced coding, for which we will be using the Julia programming language. If you are not familiar with Julia, you can find some helpful syntax tipps here (https://cheatsheets.quantecon.org).

Exercise 10: Quantum match making

Decoding a quantum error correction code is the process of using the results of stabilizer measurements to determine the errors that have occurred and correcting them. On the last exercise sheet we discussed how to correct errors in the repetition code. This was rather straightforward, however the code was not very ideal since it was only able to protect the logical qubit against one type of error. A better code that we already know about is the toric code. However, correcting isn't as trivial as for the repetition code. In this exercise we will discuss how to decode the toric code using an algorithm called **Minimum Weight Perfect Matching (MWPM)**.

First let's understand the idea of the problem we want to solve. Errors on the physical system qubits of the toric code appear as error strings which result in negative measurement outcomes of the stabilizers (syndromes) at both ends of the error string. Now given all the stabilizers with negative measurement outcomes we want to find a **matching** between pairs of stabilizers (using all, thus **perfect**) that best describes the error strings that occurred. Our ansatz is that the error strings are as short as possible (of **minimum weight**), which can be justified by the fact that the probability of an error string of length l occurring, decreases exponentially with l. Once we have such a *minimum weight perfect matching*, we want to correct the state by flipping all qubits along a path that connects the matches.

Let's start at the beginning and sample some physical errors following an error probability of p on every qubit. A value of 0 means the qubit is unaffected by errors, whereas a value of 1 means the qubits got flipped due to noise.

a) Write a function that returns `physical_errors::Vector{Int64}` of length $2d^2$ with zeros and ones based on a probability p and a code distance d. We want to use linear indexing for this (see fig. 1), as this will make the syndrome calculation quite neat!

Now that we sampled the physical errors, we want to calculate the syndromes (the measurement

outcomes of the stabilizers). Note that this is what we would actually see on a real quantum device, since we have no way of knowing the explicit errors on the physical qubits!

In this exercise we want to limit ourselves to bit flip noise, meaning that we only look at plaquettes of X-stabilizers (i.e. we neglect the vertex Z-stabilizers).

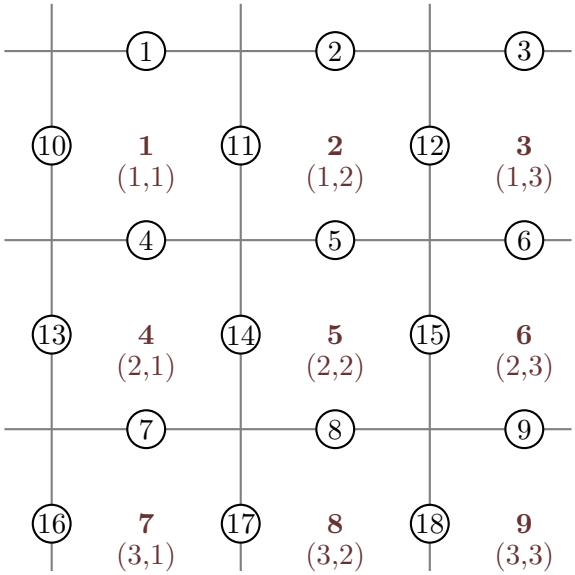

Figure 1 – Indexing of the physical qubits (black) and X stabilizers (red) in a $d = 3$ toric code.

b) Write a function that return `x_syndromes::Vector{Int64}` of length d^2 that contains zeros and ones based on if the X-stabilizer has a positive or negative measurement outcome. This function should take the `physical_errors` as input.
Hint: You can create a matrix A of size d^2 x $2d^2$ which stores the information of *which qubit is part of which stabilizer* as zeros and ones. Multiplying this matrix with the `physical_errors` will give you a vector of size d^2 with elements $\{0, 1, 2, 3, 4\}$ based on the number of flipped qubits on a plaquette. Running

```julia
mod.(A * physicalErrors, 2)
```

will give you the parity of every element in that vector, such that a 1 means that the stabilizer is active (negative measurement outcome) and a zero means it't not active (positive measurement outcome).
2. Hint: For calculating the correct indices of the matrix elements you might want to use the following julia functions:

```julia
# modulo function (you can also write i % n)
mod(i, n)

# one based modulo function (very useful due to 1 based indexing in julia)
mod1(i, n)
```

Great! Now we are able to sample stabilizer measurements based on a physical error rate on our qubits. Now we only need to find a way to match the syndromes to create paths that are as short as possible

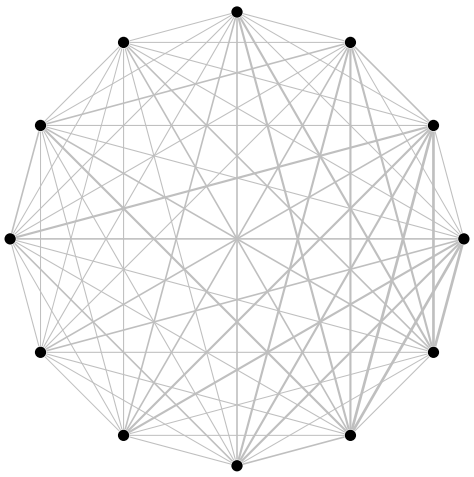

Figure 2 – Complete graph with edge weights visualized as line thickness.

and subsequently correct all qubits along such paths. That can't be too hard, ... right? (Spoiler: It can.)

Finding such matchings is a perfect candidate for a problem solvable using the language of graphs. Translating the problem to a graph problem can be done as follows: We represent the stabilizers with negative measurement outcomes as nodes in a complete undirected weighted graph, where the weights on the edges correspond to the distance between the stabilizers in the code lattice.

Now we want to write a function that given the `x_syndromes` returns a graph representation of the problem. We can use the following function for that:

julia

```julia
using Graphs

# construct complete graph (and weights + syndrom positions)
# given an x_syndrom configuration
function to_complete_graph(x_syndroms::Vector{Int64})
    d = Int64(sqrt(length(x_syndroms)))
    syndrom_pos = [to_2d_index(i, d) for i in findall(x_syndroms .== 1)]
    num_nodes = length(syndrom_pos)

    # construct graph and Dict to store weights
    graph = SimpleGraph(num_nodes)
    weights = Dict{edgetype(graph),Int64}()

    for (i, pos1) in enumerate(syndrom_pos[1:end-1])
        for (j, pos2) in enumerate(syndrom_pos[i+1:end])
            e = Edge(i, j + i)
            add_edge!(graph, e)
            weights[e] = taxicab_metric(pos1, pos2, d)
        end
    end

    return graph, weights, syndrom_pos
end

# linear index to 2d index
function to_2d_index(i::Int64, d::Int64)
    return ((i - 1) ÷ d + 1, mod1(i, d))
end
```

Somehow the function `taxicab_metric` got lost due to noise on the classical bits! Can you help recovering it?

c) Write a function `taxicab_metric` that calculates the number of qubits that you need to flip to connect the two stabilizers given by the positions `p` and `q`. The return value should be an integer.

julia

```julia
# calculate the distance between two qubits
function taxicab_metric(
    p::Tuple{Int64,Int64},
    q::Tuple{Int64,Int64},
    d::Int64
)
    ####################
    ### do something ###
    ####################

    return distance
end
```

Hint: We are on a torus, so the shortest path may be through the boundaries!

The next step is to find a perfect match. Since we have a complete graph and an even number of nodes, this is always possible. Finding **a** perfect match (not the minimum weight) is also quite simple.

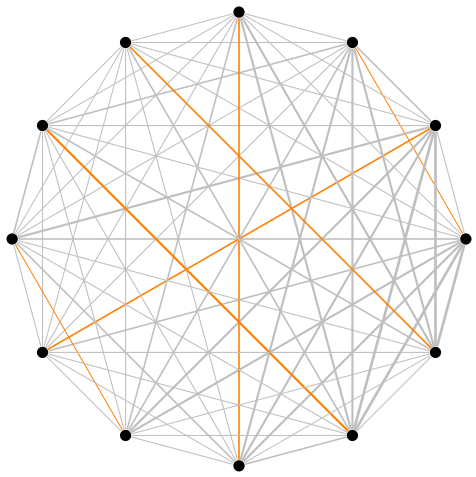

Figure 3 – Non-minimum weight perfect matching. Matched edges are visualized in orange.

d) Define a function in julia that - given the graph representation of the problem - , returns a perfect match. Ideally, the return value should be `mate::Vector{Int64}` such that the matched mate of node `i` is `mate[i]` and the matched mate of node `j` is `mate[j]`.

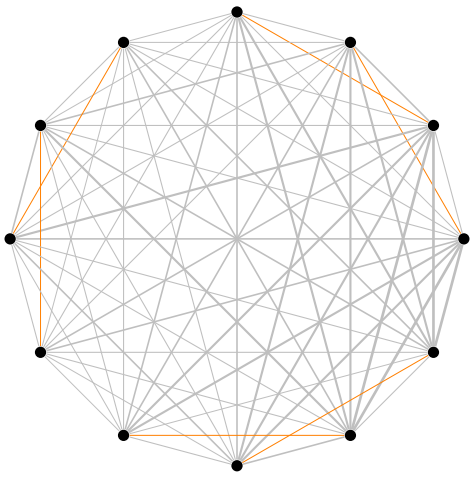

Figure 4 – Minimum weight perfect matching. Matched edges are visualized in orange.

Now there is of course the problem of finding the **minimum weight** perfect matching. Naively, we could just construct all possible perfect matchings and take the one with the minimum weight, that can't be too bad, ... can it?

e) Using brute force (comparing the weight sum of all perfect matchings explicitly), what would be the largest code distance d that could be decoded in reasonable time?
Hint: How many perfect matchings are there in a complete graph with n (even) nodes?

f) Implement the brute force method for finding the minimum weight perfect matching in julia.
Hint: This can be done using recursion.

Since decoding should happen at real time in parallel to the quantum computations, which currently have short coherence times, we need to find a faster way to solve the problem. Especially for larger code distances.

An algorithm that gives super-exponential speed up over the brute force method is the **Blossom algorithm**. Originally we wanted to create an exercise that would guide you to your own implementation of this algorithm. However, as we did our research and learned about the algorithm ourselves we noticed this would be a bit too much. Let's just cite a blog article by Abraham Flaxman that we found while reading about the implementation:

> " *..., finding minimum-weight perfect matchings in general graphs is not a reasonable assignment for a programming class, even a very advanced one. If you are reading this page trying to get homework answers, you should double check what exactly was assigned. Either you've misunderstood the problem, or your professor is a cruel, cruel mathematician.*"

We decided we don't want to be cruel mathematicians, so we will use a library instead! :-) However, we still want to give you an idea of how the algorithm works as this is quite interesting!

The idea of the **Blossom** algorithm goes like this: starting from a not perfect matching configuration, find a way to improve it by adding one more pair at a time. To make this possible we might need to break up and rematch some matches. Such an improvement step works as follows:

- Find a path in the graph that starts and ends at unmatched nodes and has alternating matched and unmatched edges (augmenting path). Furthermore, the edges in the path should all be *tight* (more on that later).

- Invert the matching of the edges on the path, i.e. if an edge was in the matching before it is not in the matching after and vice versa.

This process can be repeated until it is no longer possible to find such a path, at which point we have found a perfect matching.

Clearly the difficult part is finding such an augmenting path. One might think that finding such a path should be possible using a simple breadth first search (BFS) algorithm. However, for a non-bipartite graph there can be cycles with an odd number of edges (and nodes), called Blossoms, which make this more difficult. The idea is to contract these Blossoms into a single node, which can then be treated as a single node in the search for the path. The contraction can be undone after the path has been found. Furthermore, to ensure we find the matching of minimum weight, we need to make sure that all the edges in the path are tight. Let's define what we mean by that. We define a value (dual variable) for every node (and Blossom) in the graph $\{v_i\}$. For an edge (i, j), we define the slack of the edge as

$$s_{ij} = v_i + v_j - w_{ij}, \tag{1}$$

where w_{ij} is the weight of this edge. An edge is tight if $s_{ij} = 0$. If we think about the weights w_{ij} as the length of the edges, and about the dual variables $\{v_i\}$ as the radius of a circle, an edge is tight if the two circles touch each other. By updating the dual variables in a smart way, we vary the radiuses of the circles and that way 'search' the graph for the best matches.

Going into more detail would be a bit too much, but if you're interested you can read up on this algorithm here: Efficient Algorithms for Finding Maximum Matching in Graphs.

g) Use the `minimum_weight_perfect_matching(graph, weights)` function from `GraphsMatching.jl` instead of the brute force method to find the minimum weight perfect matching for the graph you defined in the previous parts of this exercise. By using the following small wrapper function, the input and return types should be exactly the same as for the brute force method.

julia
```julia
using GraphsMatching

function mwpm(graph, weights)
    res = minimum_weight_perfect_matching(graph, weights)
    return res.mate
end
```

Awesome! Now we are able to find the minimum perfect matchings between our syndromes in a reasonable runtime! All that is left now is to correct the physical qubits and check whether the logical information is still there. To get the `error_string` that connects two syndromes you can use the function `get_error_string(pair::Tuple{Tuple{Int64,Int64},Tuple{Int64,Int64}}, d)` that we provide to you in the ILIAS folder - Sheet05.

h) Write a function that loops through all matchings and given the positions of the syndromes `syndrome_pos` based on their index in the graph representation calculates the error string that connects a pair of syndromes (using the `get_error_string` function). Now flip all qubits along that error string. Note that the function `get_error_string` returns a `Vector{Int64}` where each element is the linear index of a physical qubit which is part of such an error string. The output of this function should be the `physical_errors::Vector{Int64}` which we created in a), but of course with some corrections!

At this point we corrected our qubits and we restored the logical information, ... or ... have we? Let's check! We can check whether the logical information is still there, by calculating Z-string operators

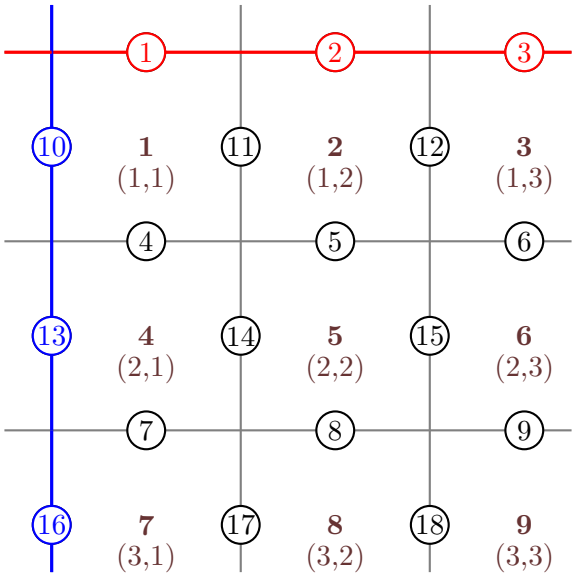

Figure 5 – Visualization of the logical Z-operators that span the torus vertically (blue) and horizontally (red).

(since we are only considering the X-stabilizers) that loop vertically and horizontally through the torus.

i) Write a function, that given the (corrected) `physical_errors::Vector{Int64}` calculates whether the logical information is lost. The function should return 1 if the information is lost, and 0 if it is still there. You can do this by calculating the parity of all physical qubits in a given logical loop operator. The logical information is lost, if any of the two logical operators has a value of -1, i.e. if the parity of its containing qubits is odd.

Now we want to check for which code distances d and which physical error rates p the logical information is preserved / lost.

j) Write a simulation that calculates the logical error for some values of p (e.g. between 0 and 0.2) and some code distances d (e.g. $3, 5, 7, 9$) for many different error configurations (e.g. 5000) and average over them. Plot the logical error as a function of the physical error probability p for different code distances d. If you are not familiar with plotting in julia, you can use the following code:

julia

```julia
using CairoMakie
using ColorSchemes

# get some nice colors
colors = [color for color in ColorSchemes.Spectral_6]

# plot
fig = Figure()
ax1 = Axis(fig[1, 1];
    xlabel=L"physical errorrate $p$",
    ylabel=L"logical errorrate $p_L$"
)
for (i, d) in enumerate(ds)
    scatterlines!(ax1, ps, log_err[i, :], color=colors[i])
end
lines!(ax1, ps, ps; color=:black, linestyle=:dash)
axislegend(ax1,
    [
        [
            MarkerElement(marker=:circle, color=c),
            LineElement(color=c)
        ] for c in reverse(colors)
    ],
    [L"%$d" for d in reverse(ds)],
    L"code distance $d$",
    position=:lt
)
fig
```

What do you observe? In the above code we also plotted a straight line. What could be the meaning of that?

In the plot you should already see a nice crossing point and - just by looking at it - you should be able to tell where the threshold lies. However, we can do even better than that by performing a finite size scaling analysis. Without going into too much detail (see the computational many-body physics lecture for that) you can exploit the behavior of finite systems around a critical point to extract information about the nature of the transition, as well as the exact transition point. We can do this analysis by rescaling the x and y data based on a *scaling Ansatz* or scaling function. If done correctly the data should collapse onto a single line (at least in the vicinity of the critical point). In order to get the best parameters we can do an automated finite size scaling analysis using e.g. the `ScalingCollapse.jl` package.

k) Perform a finite size scaling analysis using the following code to extract the critical point p_{th} and the critical exponent ν.

julia

```julia
using ScalingCollapse

# perform finite size scaling analysis
sp = ScalingProblem(ps, log_err, ds;
    sf=ScalingFunction(:x; p_names=["x_c", "nu"]),
    dx=[-0.5, 0.5],                        # set the optimization interval
    p_space=[0.01:0.01:0.2, 1.0:0.1:2.0], # set the allowed parameter space
)
```

Running the above code should print you the optimal scaling parameters (here called `x_c` and `nu`). Let's plot the collapsed data! You can use the following code:

julia

```julia
# get the rescaled data
sx, sy, se, sd = scaled_data(sp);

fig = Figure()
ax1 = Axis(fig[1, 1];
    xlabel=L"physical errorrate $p$",
    ylabel=L"logical errorrate $p_L$",
)
ax2 = Axis(fig[1, 2];
    limits=((-2.3, 2.3), nothing),
    xlabel=L"(p-p_c)/p_c \cdot d^{1 / \nu}"
)


for (i, d) in enumerate(ds)
    scatterlines!(ax1, ps, log_err, color=colors[i])
    scatterlines!(ax2, sx[i], sy[i])
end
axislegend(ax1,
    [
        [
            MarkerElement(marker=:circle, color=c),
            LineElement(color=c)
        ] for c in reverse(colors)
    ],
    [L"%$d" for d in reverse(ds)],
    L"code distance $d$",
    position=:lt
)
colsize!(fig.layout, 2, Relative(0.3))
fig
```

You should find that the data point in the right plot (roughly) collapse onto one line! The critical exponent ν tells us something about the nature of the phase transition. Do you expect it to change if we use a different algorithm for decoding? What about the critical point / threshold?

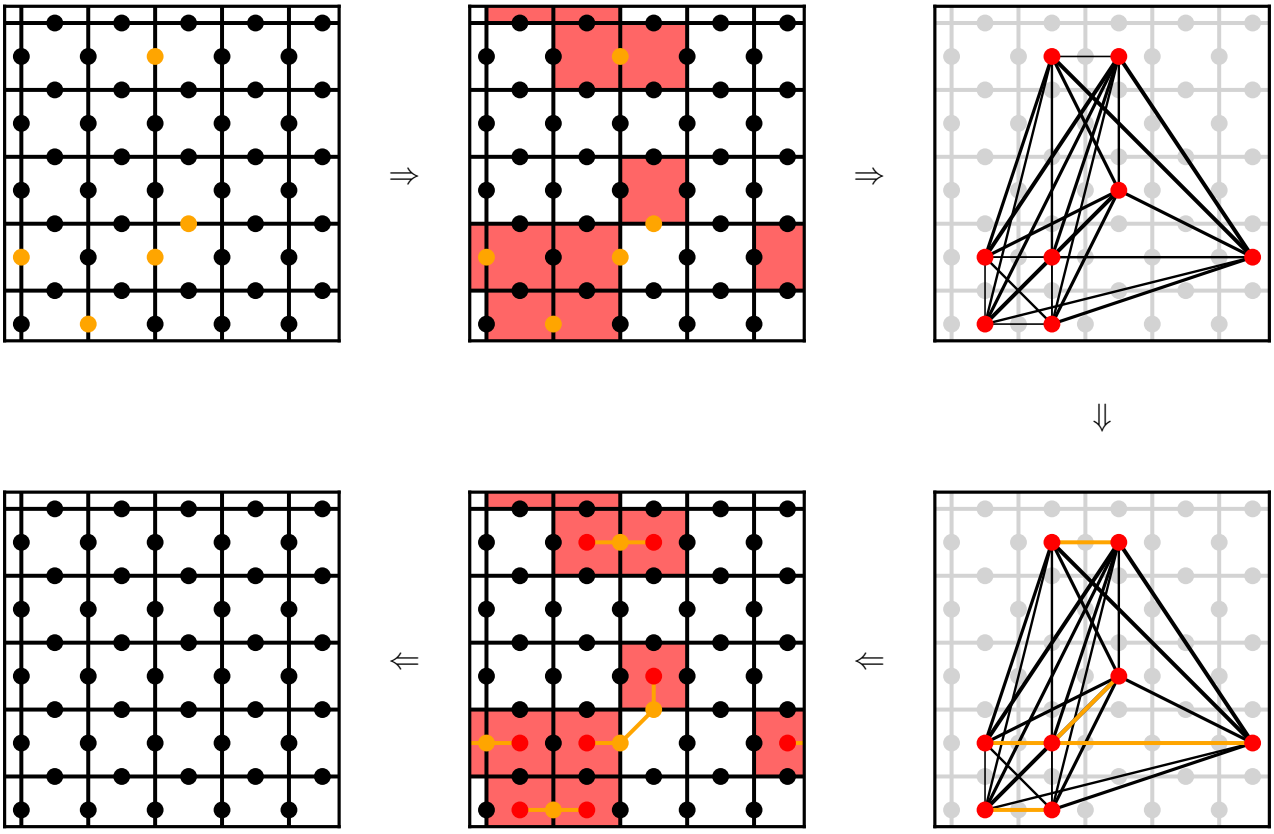

Figure 6 – Summary of the decoding pipeline for the toric code. Step 1: Sample physical errors. Step 2: Calculate syndromes. Step 3: Generate a complete graph with weights. Step 4: Find a minimum weight perfect matching. Step 5: Determine the error strings. Step 6: Correct the errors.