Repetition code on IQM - solutions

November 6, 2025

1 Quantum Error Correction with the Repetition Code

The notebook follows a hands-on approach:

- You will **build and run** repetition code circuits using Qiskit,
- **Analyze** syndrome patterns from real device data,
- **Implement** and use a **Minimum Weight Perfect Matching (MWPM)** decoder with PyMatching
- **Simulate the repetition code threshold** efficiently using Stim

This material was written and developed by **Moritz Lange**, **Vidar Petersson**, and **Mats Granath**.

[13]:
```
!pip install stim pymatching
```

```
Requirement already satisfied: stim in ./.venv/lib/python3.12/site-packages
(1.15.0)
Requirement already satisfied: pymatching in ./.venv/lib/python3.12/site-
packages (2.3.1)
Requirement already satisfied: numpy in ./.venv/lib/python3.12/site-packages
(from stim) (2.3.4)
Requirement already satisfied: scipy in ./.venv/lib/python3.12/site-packages
(from pymatching) (1.15.3)
Requirement already satisfied: networkx in ./.venv/lib/python3.12/site-packages
(from pymatching) (3.5)
Requirement already satisfied: matplotlib in ./.venv/lib/python3.12/site-
packages (from pymatching) (3.10.7)
Requirement already satisfied: contourpy>=1.0.1 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (1.3.3)
Requirement already satisfied: cycler>=0.10 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (4.60.1)
Requirement already satisfied: kiwisolver>=1.3.1 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (1.4.9)
Requirement already satisfied: packaging>=20.0 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (24.1)
Requirement already satisfied: pillow>=8 in ./.venv/lib/python3.12/site-packages
```

```
(from matplotlib->pymatching) (12.0.0)
Requirement already satisfied: pyparsing>=3 in ./.venv/lib/python3.12/site-
packages (from matplotlib->pymatching) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in
./.venv/lib/python3.12/site-packages (from matplotlib->pymatching) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in ./.venv/lib/python3.12/site-packages
(from python-dateutil>=2.7->matplotlib->pymatching) (1.16.0)
```

[32]:
```python
from qiskit import transpile, ClassicalRegister, QuantumCircuit, QuantumRegister
import matplotlib.pyplot as plt
from collections import defaultdict
import math
import os
import json
import numpy as np
import pymatching
import stim
from iqm.qiskit_iqm import IQMProvider
from pathlib import Path
from getpass import getpass
import uuid
from scipy.optimize import curve_fit
```

1.1 RepetitionCodeCircuit

This class builds a **repetition code circuit**, a minimal example of a quantum error-correcting code.
It encodes a single logical qubit into `d` physical qubits and performs `T` rounds of stabilizer (syndrome) measurements to detect errors.

Overview: - The circuit consists of `d` **code qubits** (holding the logical information) and `d-1` **measure qubits** used to measure parities between neighboring code qubits.
- Each **syndrome measurement round** entangles the code and measure qubits via CNOT gates, then measures the measure qubits to extract error information without collapsing the logical state.
- After the final round, all code qubits are measured in the chosen basis (`Z` or `X`) to perform the logical readout and final syndrome extraction.

Parameters: - `d`: number of code qubits (code distance).
- `T`: number of syndrome measurement rounds.
- `xbasis`: if `True`, encodes and measures in the X basis instead of Z.

[100]:
```python
class RepetitionCodeCircuit:
    """RepetitionCodeCircuit class."""

    def __init__(
        self,
        d: int,
        T: int,
        xbasis: bool = False,
```

```
    ):
        """
        Creates the circuits corresponding to a logical 0 (or logical +1, if␣
↪xbasis=True)
        using a repetition code.

        Implementation of a distance d repetition code, implemented over
        T syndrome measurement rounds.

        Args:
            d (int): Number of code qubits (and hence repetitions) used.
            T (int): Number of rounds of measure-assisted syndrome measurement.
            xbasis (bool): Whether to use the X basis to use for encoding (Z␣
↪basis used by default).

        """

        super().__init__()

        self.n = d
        self.d = d
        self.T = 0

        self.code_qubit = QuantumRegister(d, "code_qubit")
        self.measure_qubit = QuantumRegister((d - 1), "measure_qubit")
        self.qubit_registers = {"code_qubit", "measure_qubit"}

        self.measure_bits = []
        self.code_bit = ClassicalRegister(d, "code_bit")

        self.circuit = QuantumCircuit(self.measure_qubit, self.code_qubit)

        self._xbasis = xbasis

        # state preparation
        if self._xbasis:
            self.circuit.h(self.code_qubit)

        for _ in range(T - 1):
            self.syndrome_measurement()

        if T != 0:
            self.syndrome_measurement()
            self.readout()

    def syndrome_measurement(self):
        """Application of a syndrome measurement round.
```

```
        """

        self.measure_bits.append(ClassicalRegister((self.d - 1), "round_" +␣
↪str(self.T) + "_measure_bit"))

        self.circuit.add_register(self.measure_bits[-1])

        # entangling gates
        self.circuit.barrier()
        if self._xbasis:
            self.circuit.h(self.measure_qubit)
        for j in range(self.d - 1):
            if self._xbasis:
                self.circuit.cx(self.measure_qubit[j], self.code_qubit[j])
            else:
                self.circuit.cx(self.code_qubit[j], self.measure_qubit[j])
        for j in range(self.d - 1):
            if self._xbasis:
                self.circuit.cx(self.measure_qubit[j], self.code_qubit[j + 1])
            else:
                self.circuit.cx(self.code_qubit[j + 1], self.measure_qubit[j])
        if self._xbasis:
            self.circuit.h(self.measure_qubit)
        # measurement
        self.circuit.barrier()
        for j in range(self.d - 1):
            self.circuit.measure(self.measure_qubit[j], self.measure_bits[self.
↪T][j])

        self.T += 1


    def readout(self):
        """
        Readout of all code qubits, which corresponds to a logical measurement
        as well as allowing for a measurement of the syndrome to be inferred.
        """
        if self._xbasis:
            self.circuit.h(self.code_qubit)
        self.circuit.add_register(self.code_bit)
        self.circuit.measure(self.code_qubit, self.code_bit)
```

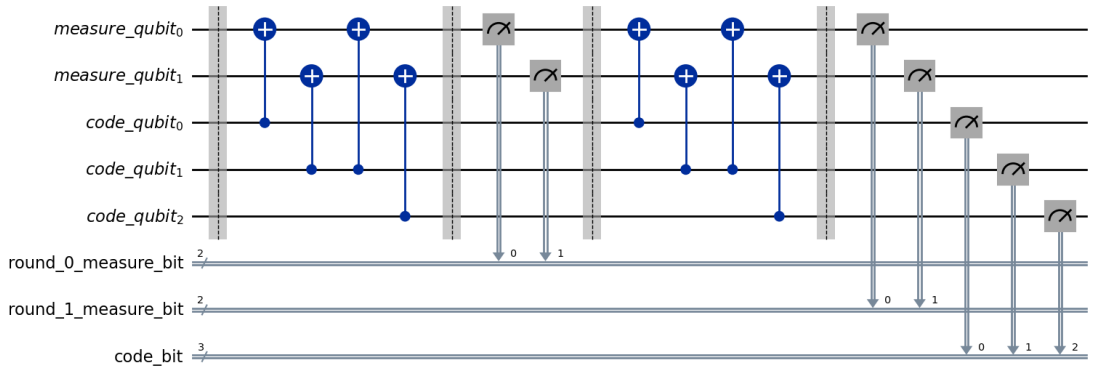
1.2 Task: Explore Basis Choice in the Repetition Code

This task demonstrates how the choice of encoding basis affects the structure of the repetition code circuit.

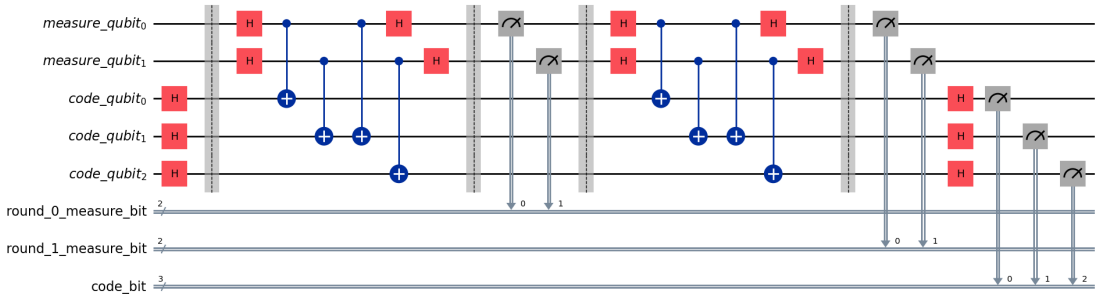
A distance-3 code is used with two rounds of syndrome measurements. When `xbasis=True`, the logical qubit is encoded in the X basis, protecting against **phase-flip errors**.
When `xbasis=False`, the code is encoded in the Z basis, protecting against **bit-flip errors**.

Your task: - Switch between `xbasis=True` and `xbasis=False`.
- Observe how the pattern of CNOT gates changes (who controls whom).
- Identify how the measurement basis and the type of stabilizers differ between the two cases.
- Think about why the X-basis version protects against phase errors while the Z-basis version protects against bit-flip errors.

[144]:
```
d, d_t = 3, 2
circuit_class_instance = RepetitionCodeCircuit(d, d_t, xbasis=False)
circuit = circuit_class_instance.circuit
circuit.draw(output="mpl")
```

[144]:


[145]:
```
d, d_t = 3, 2
circuit_class_instance = RepetitionCodeCircuit(d, d_t, xbasis=True)
circuit = circuit_class_instance.circuit
circuit.draw(output="mpl")
```

[145]:


1.3 Connecting and Transpiling the Circuit for Hardware Execution

This code connects to the **IQM Resonance** platform, selects a real quantum processor, and transpiles the repetition code circuit so it can be executed on that device.

Step-by-step explanation:

1. **Authenticate and connect to IQM**
   The script checks for an existing authentication token (`token.txt`).
   If no token is found, it prompts you to enter the access token generated from the IQM Resonance dashboard.
   The token is stored locally and used to authenticate future connections.
   After authentication, an **IQMProvider** instance is created and used to connect to the selected backend, here `garnet`.

2. **Select the backend**
   The provider connects to the specified backend URL (`https://cocos.resonance.meetiqm.com/garnet`) and loads the corresponding hardware configuration.
   This backend represents a real superconducting quantum processor of IQM.

3. **Transpile the circuit**
   The `transpile()` function adapts the repetition code circuit to the constraints of the selected IQM backend:

   - `backend=backend` ensures the circuit uses the correct gate set and qubit connectivity.
   - `initial_layout=layout` (optional) specifies which physical qubits to use for reproducibility.
   - `routing_method="none"` disables automatic SWAP insertion, keeping the logical qubit mapping intact.
   - `optimization_level=1` performs light optimization without altering the circuit structure.
   - `seed_transpiler=42` makes the transpilation deterministic and repeatable.

4. **Visualize the transpiled circuit**
   The final hardware-mapped circuit is displayed using
   `draw(output="mpl", idle_wires=False)`
   This shows the actual physical qubit layout and gate mapping while hiding unused qubits for clarity.

After these steps, the repetition code circuit is ready for execution on the selected IQM device.

Note:
The transpiler or backend may rename registers internally (for example, using names like `ancilla` or `q` instead of `measure_qubit`).
Don't worry about the exact naming, simply identify the **measurement qubits** as those that are actually measured in each round of the circuit.

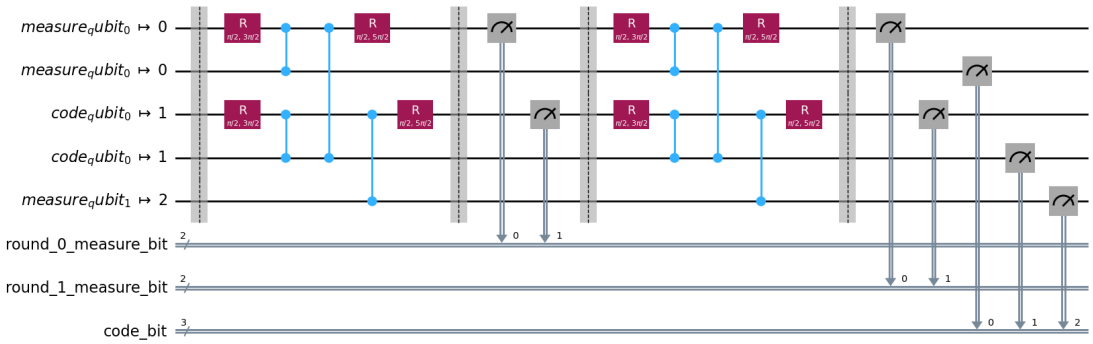
[146]:
```
token_file = Path('token.txt')

if not token_file.exists():
    token = ""
    while token == "":
        token = getpass("Please enter the token from that you generated from␣
 ↪IQM resonance")
        token_file.write_text(token)
else:
    print("Reusing existing token.txt")
token = token_file.read_text()
os.environ["IQM_TOKEN"] = token
```

```
Reusing existing token.txt
```

[147]:
```
backend_name = 'garnet'
provider = IQMProvider(url = "https://cocos.resonance.meetiqm.com/" +␣
 ↪backend_name)
backend = provider.get_backend()
```

[148]:
```
# Transpile the circuit for the target backend without routing swaps
layout = None  # Set to fixed physical layout for reproducibility, if desired
transpiled_circuit = transpile(
            RepetitionCodeCircuit(d, d_t, xbasis=False).circuit,
            backend=backend,
            initial_layout=layout,
            routing_method="none",  # Disable SWAP-based routing
            optimization_level=1,
            seed_transpiler=42,
        )
transpiled_circuit.draw(output="mpl", idle_wires=False)
```

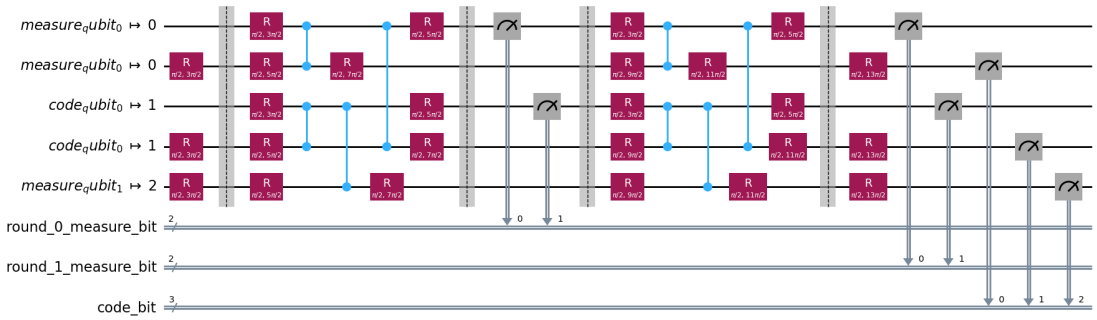
[148]:



[149]:
```
# Transpile the circuit for the target backend without routing swaps
layout = None  # Set to fixed physical layout for reproducibility, if desired
```

```
transpiled_circuit = transpile(
            RepetitionCodeCircuit(d, d_t, xbasis=True).circuit,
            backend=backend,
            initial_layout=layout,
            routing_method="none",  # Disable SWAP-based routing
            optimization_level=1,
            seed_transpiler=42,
        )
transpiled_circuit.draw(output="mpl", idle_wires=False)
```

[149]:



1.4 Task: Understanding Basis Choice and IQM's Native Gate Decomposition

1. **Start in the Z basis.**
   Run and visualize the repetition code with `xbasis=False`.
   In the transpiled circuit, each logical **CNOT** appears as a native **CZ** sandwiched by **Hadamards on the target**, but IQM implements H as single-qubit via:

   - pre-CZ: `R( /2, n /2)`
     These realize a Hadamard up to a global phase, which is why you see `R( , )` instead of `H`.

2. **Verify the CNOT construction.**
   Explicitly check that $(I \otimes H)\, CZ\, (I \otimes H) = CNOT$.

3. **Switch to the X basis.**
   Set `xbasis=True` and re-run the transpilation. Compare the resulting native circuit to the Z-basis version you analyzed before.

From now on, we'll work with the phase flip repetition code (X basis).

1.5 Sampling from an IQM device

We now execute the transpiled repetition code circuit on the selected **IQM backend** using Qiskit's IQM provider. The IQM backend is responsible for submitting the circuit to the hardware, running it, and collecting the measurement results.

1. **Set the number of shots.**
   Each circuit is executed `shots` times on the device.

2. **Submit the circuit.**
   The circuit is sent to the backend with `backend.run(...)`.
   The backend queues the execution and returns a job ID that can be used to later retrieve the job. You can find your job_id on the IQM Resonance platform webpage under your account's job history.

3. **Retrieve the results.**
   Once the job has finished, the results can be accessed through the job ID.
   The backend returns bitstring counts and metadata.

4. **Store the results.**
   The results are saved locally as a JSON file containing the job ID, code distance `d`, number of rounds `d_t` and number of shots.
   This provides a consistent record for later analysis.

Note: Avoid calling backend.run([…]) too often! Each execution consumes part of your IQM runtime quota.
If you've already run a job, you can retrieve its results later instead of re-running the circuit. > python > job = backend.retrieve_job(job_id)

result = job.result()

[159]:
```
shots = 10_000
# Each execution of the following line consumes part of your IQM runtime quota!
# job = backend.run(transpiled_circuit, shots=shots)

job = backend.retrieve_job("019a589d-abb5-71a0-ae86-41a8e19c6cdb")
job_result = job.result()
counts = job_result.get_counts()
print(counts)
```

```
{'110 11 01': 1, '000 00 11': 99, '010 01 11': 14, '000 00 01': 27, '001 10 00':
16, '000 01 11': 20, '010 11 11': 32, '100 10 11': 5, '000 10 01': 13, '001 01
11': 18, '101 11 11': 1, '001 00 10': 6, '100 00 01': 7, '000 10 11': 16, '110
10 11': 3, '010 10 11': 15, '101 00 01': 1, '000 01 10': 31, '010 10 10': 12,
'011 10 11': 10, '010 01 01': 6, '011 00 01': 1, '110 11 11': 2, '110 01 11': 2,
'001 00 11': 13, '010 00 11': 20, '001 11 11': 7, '110 00 11': 1, '000 00 10':
25, '001 01 10': 7, '001 10 11': 7, '100 11 10': 5, '011 00 11': 6, '101 01 01':
2, '001 10 01': 5, '000 11 00': 33, '001 10 10': 11, '011 11 11': 6, '000 10
10': 98, '010 00 10': 10, '010 00 00': 49, '010 01 10': 44, '000 11 10': 17,
'011 10 10': 2, '010 11 10': 12, '010 00 01': 9, '000 01 01': 79, '000 00 00':
3702, '011 01 00': 13, '000 11 11': 20, '001 00 00': 211, '000 01 00': 108, '101
01 00': 9, '011 11 00': 111, '011 00 00': 15, '010 11 01': 19, '100 00 11': 3,
'001 11 01': 3, '001 11 10': 25, '000 10 00': 78, '001 11 00': 15, '011 11 01':
22, '010 01 00': 49, '010 10 00': 55, '011 00 10': 15, '110 11 10': 25, '010 11
00': 1827, '100 11 01': 13, '011 11 10': 6, '100 10 10': 8, '011 10 00': 535,
'000 11 01': 8, '110 00 01': 4, '001 01 00': 1080, '100 10 00': 426, '001 01
01': 22, '111 10 10': 9, '010 10 01': 30, '100 00 10': 42, '100 01 10': 3, '110
10 10': 2, '100 11 00': 17, '101 00 10': 3, '001 00 01': 53, '110 01 00': 217,
'111 11 01': 1, '110 11 00': 14, '100 01 00': 4, '110 00 00': 9, '110 10 00': 5,
```

```
'100 00 00': 30, '101 11 00': 122, '110 01 01': 3, '101 00 00': 3, '111 00 00':
73, '100 01 11': 3, '111 00 01': 2, '101 10 00': 23, '111 00 10': 1, '111 11
10': 2, '111 01 00': 9, '011 10 01': 6, '111 10 00': 5, '101 11 10': 2, '110 01
10': 4, '101 11 01': 2, '111 10 01': 1, '101 01 10': 18, '011 01 11': 8, '011 01
10': 4, '111 01 01': 3, '011 01 01': 2, '100 10 01': 2, '101 10 01': 4, '101 01
11': 1, '111 11 00': 2, '101 10 11': 1, '100 01 01': 1, '100 11 11': 1}
```

[151]:
```python
res_filename_real_device = os.path.join(
    "./jobdata/",
    f"{backend_name}_{job.job_id()}_d_{d}_d_t_{d_t}_shots_{shots}.json"
)
os.makedirs(os.path.dirname(res_filename_real_device), exist_ok=True)

def serialize(obj):
    if isinstance(obj, uuid.UUID):
        return str(obj)
    raise TypeError(f"Type {type(obj)} not serializable")

job_result = job.result().to_dict()
with open(res_filename_real_device, "w") as f:
    json.dump(job_result, f, default=serialize, indent=2)

print(f"Measurement saved as '{res_filename_real_device}'")
```

```
Measurement saved as './jobdata/garnet_019a589d-abb5-71a0-ae86-
41a8e19c6cdb_d_3_d_t_2_shots_10000.json'
```

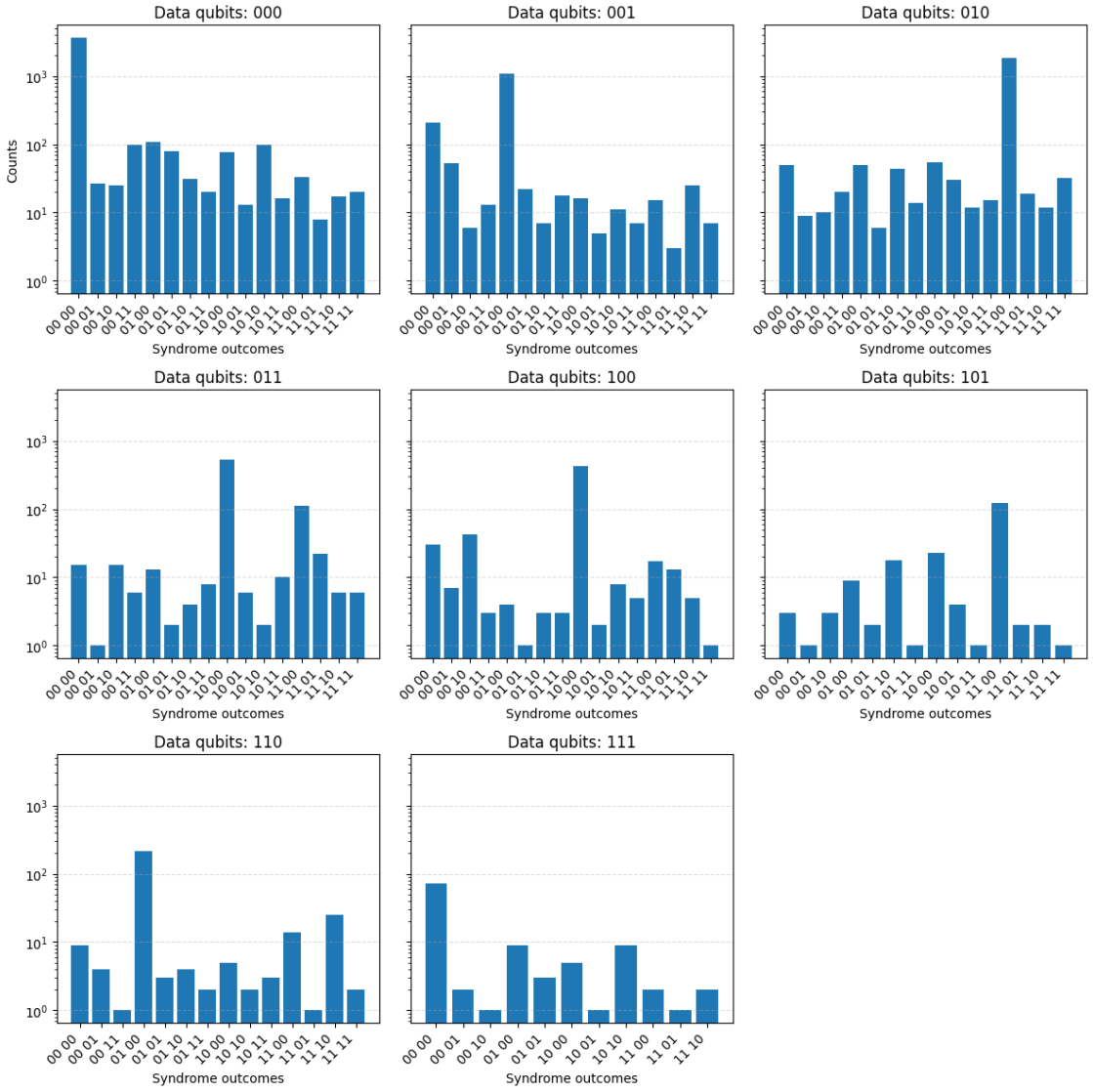
[152]:
```python
with open(res_filename_real_device) as f:
    data = json.load(f)
counts = data['results'][0]['data']['counts']
```

1.6 Visualizing measurement outcomes

Measurement results are grouped by the final data qubit states. Each subplot shows the distribution of syndrome outcomes for one data state: the x-axis lists syndrome bitstrings, and the y-axis (log scale) shows their counts. This reveals which syndrome patterns dominate for each logical outcome.

[153]:
```python
grouped_counts = defaultdict(dict)
for bitstring, count in counts.items():
    data, *syndrome = bitstring.split()
    grouped_counts[data][" ".join(syndrome)] = count

n_groups = len(grouped_counts)
n_cols = math.ceil(math.sqrt(n_groups))
n_rows = math.ceil(n_groups / n_cols)

fig, axes = plt.subplots(n_rows, n_cols, figsize=(4 * n_cols, 4 * n_rows),␣
 ↪sharey=True)
```

```
axes = axes.flatten()

for ax, (data_state, subcounts) in zip(axes, sorted(grouped_counts.items())):
    keys = sorted(subcounts.keys())
    values = [subcounts[k] for k in keys]

    ax.bar(range(len(keys)), values)
    ax.set_xticks(range(len(keys)))
    ax.set_xticklabels(keys, rotation=45, ha='right')
    ax.set_title(f"Data qubits: {data_state}")
    ax.set_xlabel("Syndrome outcomes")
    ax.set_yscale('log')
    ax.grid(axis='y', linestyle='--', alpha=0.4)

for ax in axes[len(grouped_counts):]:
    ax.axis('off')

axes[0].set_ylabel("Counts")
plt.tight_layout()
plt.show()
```



1.7 Task: Understanding Syndrome Distributions

0. **Noiseless execution.**
   Think about the circuit we implemented. In an ideal world without any noise, what outcome(s) would you expect? Next, focus on the data above:

1. **Identify the dominant syndrome.**
   For each logical data state, find the syndrome outcome with the highest count.
   Does this dominant syndrome make sense given what you expect for that logical state?

2. **Interpret deviations.**
   Look at the remaining, less frequent syndrome outcomes.
   What might these correspond to - single-qubit phase flips, measurement errors, or other noise events introduced by the circuit?

3. **Relate to decoding.**
   Discuss how a decoder could use these syndrome patterns to infer the most likely logical state. Which of the bins will go undetected and lead to a logical error?

Hint 1:
Because we don't reset the measure qubits after measurement, a persistent error on a data qubit will cause the corresponding measure to *alternate between 1 and 0* in subsequent rounds.

Hint 2:
Qiskit lists classical registers in *reverse* order of how they were added.
Within each register, bits appear from most- to least-significant.

So for an outcome like `100 01 00` (with `d=3`, `T=2`):
- `100` → data qubit readout (added last, shown first)
- `01` → measure bits from the **last** syndrome round, where the **leftmost** bit is the **last measure qubit**
- `00` → measure bits from the **first** round

You can verify this order by checking `circuit.cregs`.

1.8 From Syndrome Patterns to Decoding

In the previous task, you examined how specific syndromes correspond to logical states and how noise alters these distributions.
That exercise is the **core idea behind decoding**: given a noisy measurement record (the syndrome), infer the most probable underlying logical state or error configuration that caused it.

Loosely speaking, the **decoding problem** in quantum error correction is to determine the most likely set of physical errors that explain the measured outcomes.
An ideal decoder assigns a correction minimizing the probability of a logical error.

We now move from qualitative reasoning about syndrome patterns to a quantitative decoding framework.
In the next section, you'll implement a **Minimum Weight Perfect Matching (MWPM)** decoder, which formalizes this inference process.

1.9 MWPM Decoder

The `MWPMDecoder` class implements a **Minimum Weight Perfect Matching (MWPM)** decoder for the repetition code. It uses the **PyMatching** library (Higgott & Gidney, *PyMatching v2*, 2022) to decode detection events obtained from either hardware or simulator runs. The edges in the matching graph correspond to probabilities of error events, directly obtained from the detection events via the *p_ij method* (Spitz et al.)

1. **Initialization**
   The decoder loads the measurement results, extracting the code and measure qubit readouts. The measurement outcomes are then converted to the form expected if measure qubits were reset after each round.

2. **Detection event extraction**
   The syndrome time series is converted into *detection events* - binary indicators marking

changes between consecutive measurement rounds. These correspond to nodes in the decoding graph.

3. **Logical outcome mapping**
   Logical flips are determined by comparing the final data qubit state to the expected logical state.

4. **Correlation-based edge weights**
   A full correlation matrix between detection events is computed using the p_ij method. Edge weights are set as `-log(p_ij)`, linking detection nodes both **spatially** (within a round) and **temporally** (across rounds).

5. **Graph construction and decoding**
   The matching graph is built with PyMatching, containing both space-like and time-like edges. MWPM then identifies the most likely set of error chains consistent with observed syndromes.

6. **Performance evaluation**
   The decoder compares predicted and actual logical outcomes to estimate logical accuracy and its statistical uncertainty.

[154]:
```python
class MWPMDecoder:
    """
    Minimum Weight Perfect Matching (MWPM) decoder for rep code syndrome data.

    This decoder uses PyMatching to construct a matching graph based on observed
    detection events from a quantum circuit, and predicts logical errors
    by decoding these using MWPM. Weights can be computed from pairwise␣
↪detection correlations,
    """

    def __init__(self, distance, t, counts, shots) -> None:
        """
        Initialize the MWPMDecoder.

        Parameters
        ----------
        args : Args
            Configuration object containing code parameters and backend␣
↪settings.
        """
        self.distance = distance
        self.n_measures = distance - 1
        self.t = t
        self.matcher = pymatching.Matching()
        self.shots = shots
        self.counts = counts
        self._load_job_data()

    def _load_job_data(self) -> None:
```

```
        """
        Load syndrome and logical flip data.
        """
        syndromes, final_state = [], []
        for bitstring, freq in self.counts.items():
            bits = bitstring.replace(" ", "")
            final = [b == "1" for b in bits[:self.distance]]
            syndrome = [b == "1" for b in bits[self.distance:]]
            syndromes.extend([syndrome] * freq)
            # t...0
            final_state.extend([final] * freq)
        final_state = np.array(final_state, dtype=np.uint8)
        syndromes = np.array(syndromes, dtype=np.uint8)
        # reverse time order to start from round 0:
        syndromes = syndromes[:, ::-1]
        # no reset used, so take diff in subsequent measurements
        # reshape to (n_shots, n_rounds, n_anc)
        syndromes_reshaped = syndromes.reshape(-1, self.t, self.n_measures)

        # compute diff along time axis (rounds)
        diff = (syndromes_reshaped[:, 1:, :] != syndromes_reshaped[:, :-1, :]).
↪astype(np.uint8)

        # prepend first measurement
        first = syndromes_reshaped[:, :1, :].astype(np.uint8)
        syndrome = np.concatenate([first, diff], axis=1)

        # flatten back
        syndrome = syndrome.reshape(syndromes.shape)

        # Reverse bit order back to match IBM's convention
        syndrome = syndrome[:, ::-1]

        initial_syndrome = np.full((self.shots, self.n_measures), int(0),␣
↪dtype=np.uint8)

        final_syndrome = final_state[:, :-1] ^ final_state[:, 1:]
        syndrome_matrix = np.concatenate([initial_syndrome, syndrome,␣
↪final_syndrome], axis=1)
        T = syndrome_matrix.shape[1] // self.n_measures
        reshaped = syndrome_matrix.reshape(-1, T, self.n_measures)
        flips = np.diff(reshaped, axis=1).astype(bool)
        self.detections = flips.reshape(flips.shape[0], -1)

        # get equivalence class defined by last data qubit
        self.logical_flips = (final_state[:, 0] == 1)
```

```
    def _error_correlation_matrix_full(self) -> np.ndarray:
        """
        Compute the full correlation matrix from the observed detections.

        Returns
        -------
        pij_matrix : np.ndarray
            A symmetric matrix of error-pairing probabilities between detector␣
↪events.
        """
        x = self.detections.astype(np.float64)  # shape (shots, N)

        # Compute means
        mean_i = x.mean(axis=0)  # shape (N,)
        mean_ij = (x.T @ x) / x.shape[0]  # shape (N, N)

        # Numerator and denominator
        numerator = mean_ij - np.outer(mean_i, mean_i)
        denominator = 1 - 2 * mean_i[:, None] - 2 * mean_i[None, :] + 4 *␣
↪mean_ij

        with np.errstate(divide='ignore', invalid='ignore'):
            sqrt_term = np.sqrt(1 - 4 * numerator / denominator)
            pij = 0.5 - 0.5 * sqrt_term

        pij = np.where(np.isfinite(pij), pij, 0.0)  # Replace NaNs and infs␣
↪with 0.0
        np.fill_diagonal(pij, 0.0)  # set diagonal to 0 for clarity

        return pij


    def _get_edges(self) -> None:
        """
        Build the matching graph with edges weighted according to the selected␣
↪weight scheme.

        Constructs both space-like (within a time slice) and time-like (between␣
↪time slices) edges.
        Edge weights are derived from the negative log of correlation␣
↪coefficients.
        """
        row_len = self.distance - 1
        error_correlation = self._error_correlation_matrix_full()
```

```
        error_correlation[error_correlation <= 0] = 1e-7  # Avoid log(0) or␣
↪negative weights

        weights = -np.log(error_correlation)

        # Add space-like edges (horizontal, within each time slice)
        for t_index in range(self.t + 1):
            row_start = t_index * row_len
            row_end = row_start + row_len

            for i in range(row_start, row_end - 1):
                self.matcher.add_edge(
                    i, i + 1,
                    weight=weights[i][i + 1],
                    fault_ids={i % row_len + 1},
                    merge_strategy='replace'
                )

            self.matcher.add_boundary_edge(
                row_start,
                weight=weights[row_start][row_start + 1],
                fault_ids={0},
                merge_strategy='replace'
            )

            self.matcher.add_boundary_edge(
                row_end - 1,
                weight=weights[row_end - 2][row_end - 1],
                fault_ids={row_len},
                merge_strategy='replace'
            )
        # Add time-like edges (vertical, across time slices)
        for t_index in range(self.t):
            for offset in range(row_len):
                i = t_index * row_len + offset
                j = i + row_len
                self.matcher.add_edge(
                    i, j,
                    weight=weights[i][j],
                    merge_strategy='replace'
                )

    def _evaluate_predictions(self) -> float:
        """
        Evaluate decoder accuracy using.

        Returns
```

```
        -------
        logical_accuracy : float
            Logical decoding accuracy, including both trivial and non-trivial␣
↪shots.
        """

        # Filter out trivial syndromes
        nontrivial = np.any(self.detections, axis=1)
        detections_nt = self.detections[nontrivial]
        flips_nt = self.logical_flips[nontrivial]
        # Decode predictions using MWPM
        predictions = self.matcher.decode_batch(detections_nt)
        predicted = predictions[:, 0]
        correct = np.sum(flips_nt == predicted)
        trivial_count = np.sum(~nontrivial)

        logical_accuracy = (correct + trivial_count) / self.detections.shape[0]
        logical_accuracy_err = np.sqrt(logical_accuracy * (1- logical_accuracy
                                                           ) / self.detections.
↪shape[0])

        return logical_accuracy, logical_accuracy_err, trivial_count

    def decode(self) -> float:
        """
        Full decoding pipeline: load data, construct the graph, run decoding,␣
↪and return accuracy.

        Returns
        -------
        logical_accuracy : float
            Logical accuracy on the validation set.
        """

        self._get_edges()
        pdet_mean = self.detections.mean()
        logical_accuracy, logical_accuracy_err, trivial_count = self.
↪_evaluate_predictions()
        return logical_accuracy, logical_accuracy_err, pdet_mean, trivial_count
```

1.10 Task: Investigating Error Correction by Increasing Code Distance

In this task, you will run the repetition code circuit on an IQM backend for **code distances 3 and 5**.
The goal is to see whether increasing the code distance improves the **logical accuracy** of the encoded qubit. 1. **Run both circuits.**
Execute the repetition code for `d=5` on the same IQM backend using the same number of rounds

and shots.
Save the resulting measurement data.

2. **Decode each dataset.**
   Apply the `MWPMDecoder` to the results from both distances.
   Compute the logical accuracy or failure rate for each code distance.

3. **Compare the outcomes.**
   Examine how the logical failure rate changes when increasing the code distance from 3 to 5. Does the larger code show better suppression of errors, as expected from quantum error correction?

[155]:
```
decoder = MWPMDecoder(distance=d, t=d_t, counts = counts, shots = shots)
```

[156]:
```
res_filename_real_device = 'jobdata/
 ↪garnet_019a589d-abb5-71a0-ae86-41a8e19c6cdb_d_3_d_t_2_shots_10000.json'
with open(res_filename_real_device) as f:
    data = json.load(f)
counts = data['results'][0]['data']['counts']
decoder = MWPMDecoder(distance=3, t=d_t, counts = counts, shots = shots)
decoder.decode()
```

[156]:
```
(np.float64(0.8668),
 np.float64(0.0033979075914450647),
 np.float64(0.40205),
 np.int64(3775))
```

[158]:
```
res_filename_real_device = 'jobdata/
 ↪garnet_019a589c-e8e7-7240-8058-61660edfdcca_d_5_d_t_2_shots_10000.json'
with open(res_filename_real_device) as f:
    data = json.load(f)
counts = data['results'][0]['data']['counts']
decoder = MWPMDecoder(distance=5, t=d_t, counts = counts, shots = shots)
decoder.decode()
```

[158]:
```
(np.float64(0.9185),
 np.float64(0.002736014437096413),
 np.float64(0.3786333333333333),
 np.int64(1963))
```

1.11 Threshold Estimation with Stim

Estimating the **logical error threshold** requires running the repetition code at many noise levels and code distances to see where increasing distance begins to suppress logical errors.
Doing this directly with Qiskit, either on simulators or real hardware, would be prohibitively expensive, since each data point requires thousands of circuit runs.

To make this feasible, we switch to **Stim** (Gidney, 2021) @gidney2021stim, a high-performance stabilizer simulator.
Stim models the circuits but operates directly in the stabilizer formalism and tracks *detection events*

rather than full quantum states.
This makes it orders of magnitude faster and allows simulation of millions of shots per second.

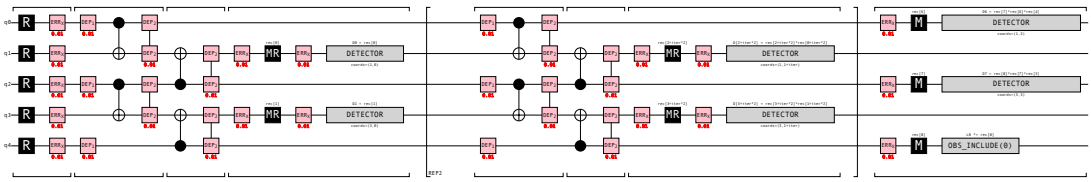
Circuit generation
We use the `stim.Circuit.generated(...)` method to define a repetition-code memory experiment. The parameters include code distance `d`, number of rounds `rounds=d`, and a noise rate `noise` which sets depolarization after Clifford gates, flip probability after resets, flip probability before measurement, and data-qubit depolarization before each round. This produces a stabilizer circuit that models repeated syndrome extraction under **circuit-level noise**.

Logical error counting & decoding
We define a `count_logical_errors(circuit, num_shots)` routine which
(1) compiles the circuit into a detector sampler via `circuit.compile_detector_sampler()`
(2) samples `num_shots` shots producing two outputs: `detection_events` (space–time patterns of syndrome flips) and `observable_flips` (whether the logical observable flipped)
(3) extracts the detector error model via `circuit.detector_error_model(decompose_errors=True)`
(4) builds a decoder graph with PyMatching (`pymatching.Matching.from_detector_error_model(...)`)
(5) decodes the detection events in batch (`matcher.decode_batch(detection_events)`)
and finally (6) counts how many decoded logical outcomes differ from the true flips, yielding the number of logical errors.

[28]:
```
def count_logical_errors(circuit, num_shots):
    sampler = circuit.compile_detector_sampler()
    detection_events, observable_flips = sampler.sample(num_shots,␣
 ↪separate_observables=True)

    detector_error_model = circuit.detector_error_model(decompose_errors=True)
    matcher = pymatching.Matching.
 ↪from_detector_error_model(detector_error_model)
    predictions = matcher.decode_batch(detection_events)

    num_errors = (predictions != observable_flips).sum()
    return num_errors
```

[30]:
```
noise = 0.01
circuit = stim.Circuit.generated(
            "repetition_code:memory",
            rounds=d,
            distance=d,
            after_clifford_depolarization=noise,
            after_reset_flip_probability=noise,
            before_measure_flip_probability=noise,
            before_round_data_depolarization=noise)
circuit.diagram('timeline-svg')
```

[30]:



1.12 Task: Estimating the Logical Error Threshold

In this task, you will use **Stim** and **PyMatching** to estimate the **logical error threshold** of the repetition code.

1. **Vary the noise strength and code distance.**
   Run the logical error counting procedure (`count_logical_errors`) for multiple values of the physical noise rate, e.g. p = 0.001, ..., 0.15 and for several code distances d = 3, ... 21.

2. **Compute logical error rates.**
   For each combination of (d, p), estimate the logical failure probability p_L using the number of logical errors returned by your decoding function.

3. **Plot the threshold curve.**
   On a log-log plot, show p_L vs. p for each code distance d.
   The **threshold** is the crossing point of these curves - where increasing d begins to *reduce* the logical error rate.

4. **Interpretation.**

   - What approximate threshold value $ p_th $ do you observe?

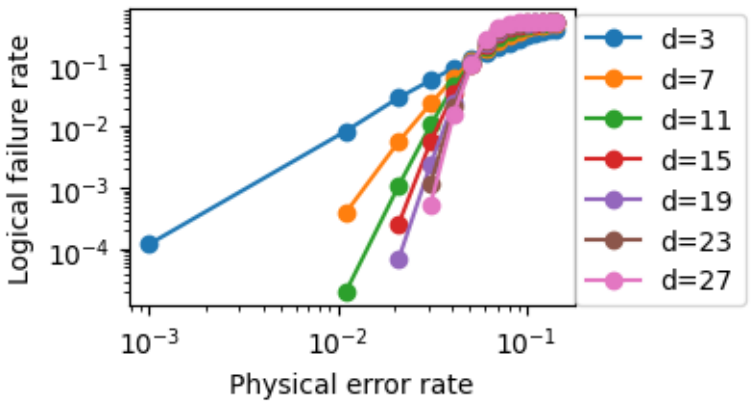
   - How does it compare to theoretical expectations for a phase-flip repetition code under circuit-level noise?

   - How does the slope below the threshold reflect the code's scaling behavior (hint: what is the shortest chain of errors that leads to a logical error?)?

This task connects the repetition code's performance under noise to the concept of **fault tolerance**: below the threshold, increasing code size exponentially suppresses logical errors, while above it, error correction fails to keep up with noise.

[ ]:
```
noises = np.arange(0.001, 0.151, 0.01)
N = 10_000
ds = np.arange(3, 31, 4)

fig, ax = plt.subplots(figsize=(3,2))
P_Ls = {}

for d in ds:
    P_L = []
    for noise in noises:
```

```
        circuit = stim.Circuit.generated(
            "repetition_code:memory",
            rounds=d,
            distance=d,
            after_clifford_depolarization=noise,
            after_reset_flip_probability=noise,
            before_measure_flip_probability=noise,
            before_round_data_depolarization=noise)
        num_errors = count_logical_errors(circuit, N)
        logical_failure_rate = num_errors / N
        P_L.append(logical_failure_rate)

    P_L = np.array(P_L)
    mask = P_L > 0
    P_Ls[int(d)] = P_L
    ax.plot(noises[mask], P_L[mask], 'o-', label=f'd={d}')

ax.legend(loc=(1, 0))
ax.set_ylabel('Logical failure rate')
ax.set_xlabel('Physical error rate')
ax.set_yscale('log')
ax.set_xscale('log')
```



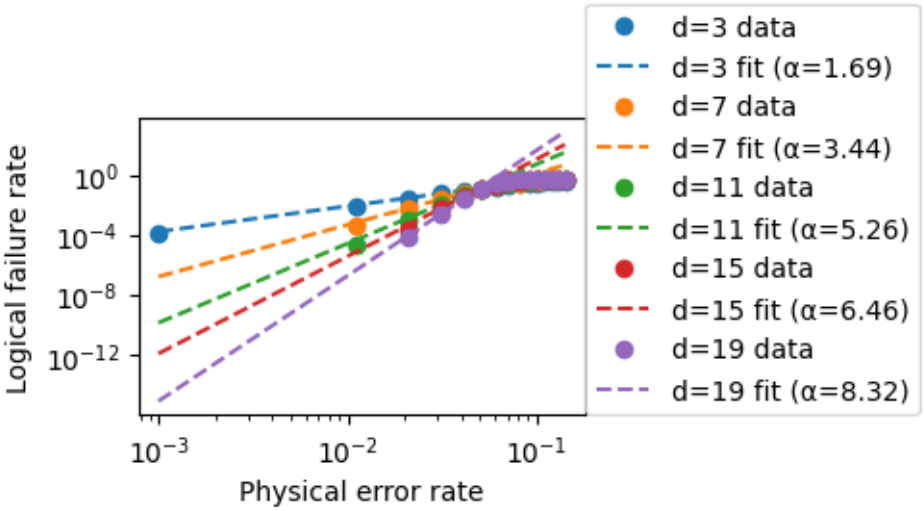
[48]:
```
def model(p, A, alpha):
    return A * p**alpha

fit_results = {}
for d in ds[:6]:
    p = noises
    P_L = P_Ls[int(d)]
    mask = (p < 0.05) & (P_L > 0)    # only low-noise and nonzero data
    p_fit = p[mask]
```

```
    P_L_fit = P_L[mask]

    def model(p, A, alpha):
        return A * p**alpha

    # log-space guess: slope ~ (d+1)/2
    A0 = P_L_fit[0] / p_fit[0]**((d+1)/2)
    alpha0 = (d+1)/2
    popt, _ = curve_fit(model, p_fit, P_L_fit, p0=[A0, alpha0], maxfev=10000)
    A, alpha = popt
    fit_results[int(d)] = (A, alpha)
    print(f"d={d}:  ={alpha:.2f}, expected={(d+1)/2:.1f}")
```

```
d=3:  =1.69, expected=2.0
d=7:  =3.44, expected=4.0
d=11:  =5.26, expected=6.0
d=15:  =6.46, expected=8.0
d=19:  =8.32, expected=10.0
d=23:  =10.16, expected=12.0

/var/folders/dg/80j736xj4kbcb971_5qyx1xr0000gn/T/ipykernel_1073/3952436229.py:18
: OptimizeWarning: Covariance of the parameters could not be estimated
  popt, _ = curve_fit(model, p_fit, P_L_fit, p0=[A0, alpha0], maxfev=10000)
```

[157]:
```
fig, ax = plt.subplots(figsize=(3,2))

for d in ds[:5]:
    p = noises
    P_L = P_Ls[int(d)]
    A, alpha = fit_results[int(d)]

    # plot data and grab its color
    line, = ax.plot(p, P_L, 'o', label=f'd={d} data')
    color = line.get_color()

    # plot fit with same color
    ax.plot(p, A * p**alpha, '--', color=color,
            label=f'd={d} fit ( ={alpha:.2f})')

ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel('Physical error rate')
ax.set_ylabel('Logical failure rate')
ax.legend(loc=(1, 0))
```

[157]:
```
<matplotlib.legend.Legend at 0x363580170>
```


Logical failure rate
10^{0}
10^{-4}
10^{-8}
10^{-12}
10^{-3}
10^{-2}
10^{-1}
Physical error rate
d=3 data
d=3 fit (α=1.69)
d=7 data
d=7 fit (α=3.44)
d=11 data
d=11 fit (α=5.26)
d=15 data
d=15 fit (α=6.46)
d=19 data
d=19 fit (α=8.32)

[ ]: