PRD 24: Topological Quantum Error Correction

Pure Thought AI Challenge 24

Pure Thought AI Challenges Project

January 18, 2026

Abstract

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

Contents

Domain: Quantum Information Theory Topology
Timeline: 6-9 months
Difficulty: High
Prerequisites: Quantum mechanics, algebraic topology, graph theory, stabilizer formalism

---

0.1 1. Problem Statement

0.1.1 Scientific Context

Topological quantum error correction represents one of the most promising approaches to building fault-tolerant quantum computers. Unlike conventional error correction codes that encode logical qubits into many physical qubits arranged in arbitrary ways, topological codes exploit the geometry and topology of lattices to provide robust protection against local errors. The key insight is that quantum information is encoded non-locally in **topological degrees of freedom** that cannot be disturbed by local perturbations.

The **surface code** (also known as the Kitaev toric code on periodic boundaries) is the leading candidate for near-term quantum error correction. It has several remarkable properties:

- **High threshold**: 10-15
- **Local stabilizer measurements**: Only nearest-neighbor interactions required
- **Homology interpretation**: Errors are 1-chains on lattice, syndromes measure boundaries
- **Anyonic excitations**: Violations of stabilizers create quasiparticle excitations that must be paired

Color codes extend surface codes by using 3-colorable lattices (triangular, hexagonal), enabling **transversal gate implementations** for Clifford gates—a crucial advantage for fault-tolerant quantum computation.

0.1.2 Core Question

Can we rigorously implement topological quantum codes, develop efficient decoders, and certify their distance and threshold properties using pure algebraic topology and graph algorithms?

Key challenges:

- **Code construction**: Build stabilizer groups from lattice homology H(), H()
- **Syndrome decoding**: Given syndrome (violated stabilizers), find minimum-weight error
- **Threshold estimation**: Determine critical physical error rate $p_thforfaulttolerance$

- **Logical operators**: Construct non-trivial homology cycles (logical X, Z)
- **Certificate generation**: Prove code distance d via chain complex computations

0.1.3 Why This Matters

- **Quantum computing**: Leading approach to fault-tolerant QC (Google, IBM, IonQ roadmaps)
- **Topological quantum memory**: Robust storage against local decoherence
- **Anyonic physics**: Connection to topological phases of matter (fractional quantum Hall effect)
- **Algebraic topology**: Deep link between homology theory and error correction
- **Complexity theory**: NP-hardness of optimal decoding, approximation algorithms

0.1.4 Pure Thought Advantages

Topological codes are **ideal for pure thought investigation**:

- Based on **algebraic topology** (homology groups, chain complexes)
- Stabilizers computable via **linear algebra over GF(2)**
- Decoding reducible to **graph algorithms** (minimum-weight perfect matching)
- Threshold estimable via **percolation theory** and **statistical mechanics**
- All results **certified via algebraic certificates** (homology computations)
- NO physical quantum hardware needed until experimental verification
- NO numerical optimization or heuristics

0.2 2. Mathematical Formulation

0.2.1 Stabilizer Formalism

Stabilizer code: Quantum error-correcting code defined by stabilizer group S = ⟨g,...,g⟩ where:

- g P are Pauli operators on n qubits: P = ±1, ±i × I, X, Y, Z^n
- All g commute: [g, g] = 0
- Code space: $V_code = |> : g|> = |> forall i$

Code parameters [[n, k, d]]:

- n: number of physical qubits
- k: number of encoded logical qubits (k = n - rank(S))
- d: code distance (minimum weight of non-trivial logical operator)

Syndrome measurement: For error E P, syndrome s = (s,...,s) where:

```
s   = 0 if [E,  g  ] = 0 (stabilizer satisfied)
s   = 1 if {E,  g  } = 0 (stabilizer violated)
```

0.2.2 Toric Code

Setup: Place qubits on edges of L×L square lattice with periodic boundary conditions (torus topology).

Stabilizers:

- **Star (vertex) operators**: Av = estar(v) $X_e(4X'saroundvertexv)$
- **Plaquette (face) operators**: Bp = ep $Z_e(4Z'saroundplaquettep)$

Properties:

- n = $2L^2$ qubits (edges)
- k = 2 logical qubits
- d = L (code distance)
- Threshold $p_th10.9$

Logical operators:

- X: product of X along horizontal non-contractible loop
- Z: product of Z along vertical non-contractible loop
- X, Z: similar for dual lattice

0.2.3 Homology Interpretation

Chain complex: C → C → C where:

- C: vector space of edges (qubits)
- C: vector space of faces (plaquettes)
- C: vector space of vertices

Boundary operators:

- : face → sum of edges around face
- : edge → sum of endpoints

Homology groups:

- H() = ker() / im() = Z^2 for torus (2 independent cycles)
- Elements of H are logical operators (non-contractible loops)

Error correction:

- Errors: 1-chains e C
- Syndromes: (e) = boundary of error chain
- Decoding: Find minimum-weight e' such that (e') = (e)

0.2.4 Surface Code with Boundaries

Setup: L×L square lattice with open boundaries (disk topology).
Properties:

- n = L^2 + (L-1)^2 qubits
- k = 1 logical qubit
- d = L
- Stabilizers: similar star/plaquette but modified at boundaries

Advantages:

- Simpler implementation (no periodic boundaries)
- Logical operators: paths connecting opposite boundaries

0.2.5 Certificates

All results must come with **machine-checkable certificates**:

- **Stabilizer certificate**: Verify all generators commute and are independent
- **Distance certificate**: Prove minimum weight of non-trivial logical operators
- **Decoder certificate**: Verify syndrome decoding produces valid error correction
- **Threshold certificate**: Statistical analysis of logical error rate vs physical error rate

Export format: JSON with stabilizer generators and homology basis:

```
{
  "code_type": "toric",
  "lattice_size": 5,
  "n_qubits": 50,
  "n_stabilizers": 50,
  "k_logical": 2,
  "distance": 5,
  "stabilizers_X": [[0,1,2,3], ...],
  "stabilizers_Z": [[4,5,6,7], ...],
  "logical_operators": {
    "X1": [0, 5, 10, 15, 20],
    "Z1": [0, 1, 2, 3, 4]
  },
  "threshold_estimate": 0.109,
  "threshold_std_error": 0.002
}
```

0.3 3. Implementation Approach

0.3.1 Phase 1 (Months 1-2): Toric Code Construction

Goal: Implement toric code on L×L lattice with stabilizers and logical operators.

```python
import numpy as np
import networkx as nx
from itertools import product
from typing import List, Tuple

class ToricCode:
    """
    Toric code on L L square lattice with periodic boundaries.

    Qubits live on edges, stabilizers on vertices (stars) and faces
       (plaquettes).
    """

    def __init__(self, L: int):
        """
        Initialize L L toric code.

        Args:
            L: Linear dimension of lattice
        """
        self.L = L
        self.n_qubits = 2 * L**2  # Horizontal + vertical edges
        self.n_stabilizers = 2 * L**2  # L   stars + L   plaquettes
        self.k_logical = 2  # 2 independent homology cycles
        self.distance = L

        # Build lattice
        self.edges_h, self.edges_v = self._construct_edges()
        self.vertices = list(product(range(L), range(L)))
        self.faces = list(product(range(L), range(L)))

        # Build stabilizers
        self.stabilizers_X = self._construct_star_operators()
        self.stabilizers_Z = self._construct_plaquette_operators()

        # Build logical operators
        self.logicals = self._construct_logical_operators()


    def _construct_edges(self) -> Tuple[dict, dict]:
        """
        Construct horizontal and vertical edges.

        Returns:
            (edges_h, edges_v) where each is a dict {(i,j): qubit_index}
        """
        L = self.L

        # Horizontal edges: connect (i,j) to (i,j+1)
        edges_h = {}
        idx = 0
```

```
        for i in range(L):
            for j in range(L):
                edges_h[(i, j)] = idx
                idx += 1

        # Vertical edges: connect (i,j) to (i+1,j)
        edges_v = {}
        for i in range(L):
            for j in range(L):
                edges_v[(i, j)] = idx
                idx += 1

        return edges_h, edges_v


    def _construct_star_operators(self) -> List[List[int]]:
        """
        Construct star (vertex) stabilizers A_v = X_e1 X_e2 X_e3 X_e4.

        Each star operator acts on 4 edges around vertex (i,j).
        """
        L = self.L
        stars = []

        for i, j in self.vertices:
            # 4 edges around vertex (i,j):
            # Horizontal: incoming from left (i,j-1), outgoing to right (i,j)
            # Vertical: incoming from below (i-1,j), outgoing upward (i,j)

            edge_indices = [
                self.edges_h[(i, (j-1) % L)],  # Left horizontal
                self.edges_h[(i, j)],           # Right horizontal
                self.edges_v[((i-1) % L, j)],  # Bottom vertical
                self.edges_v[(i, j)]            # Top vertical
            ]

            stars.append(sorted(edge_indices))

        return stars


    def _construct_plaquette_operators(self) -> List[List[int]]:
        """
        Construct plaquette (face) stabilizers B_p = Z_e1 Z_e2 Z_e3 Z_e4.

        Each plaquette operator acts on 4 edges around face (i,j).
        """
        L = self.L
        plaquettes = []

        for i, j in self.faces:
            # 4 edges around face (i,j):
            # Bottom, right, top, left (counterclockwise)
```

```
104
105             edge_indices = [
106                 self.edges_h[(i, j)],              # Bottom
107                 self.edges_v[(i, (j+1) % L)],      # Right
108                 self.edges_h[((i+1) % L, j)],      # Top
109                 self.edges_v[(i, j)]               # Left
110             ]
111
112             plaquettes.append(sorted(edge_indices))
113
114         return plaquettes
115
116
117     def _construct_logical_operators(self) -> dict:
118         """
119         Construct logical X and Z operators.
120
121         Logical operators correspond to non-contractible loops on torus:
122         - X    : horizontal loop (top row horizontal edges)
123         - Z    : vertical loop (left column vertical edges)
124         - X    : vertical loop
125         - Z    : horizontal loop
126         """
127         L = self.L
128
129         # Logical  X  : horizontal non-contractible loop (all
               horizontal edges in row 0)
130         X1 = [self.edges_h[(0, j)] for j in range(L)]
131
132         # Logical  Z  : vertical non-contractible loop (all vertical
               edges in column 0)
133         Z1 = [self.edges_v[(i, 0)] for i in range(L)]
134
135         # Logical  X  : vertical loop (all vertical edges in row 0)
136         X2 = [self.edges_v[(0, j)] for j in range(L)]
137
138         # Logical  Z  : horizontal loop (all horizontal edges in column
               0)
139         Z2 = [self.edges_h[(i, 0)] for i in range(L)]
140
141         return {
142             'X1': X1, 'Z1': Z1,
143             'X2': X2, 'Z2': Z2
144         }
145
146
147     def verify_stabilizer_commutation(self) -> bool:
148         """
149         Verify all stabilizers commute.
150
151         Returns True if all [ A  ,  A  ] = [ B  ,  B  ] = [ A  ,  B  ]
               = 0.
152         """
153         # Two Pauli operators commute if they overlap on even number of
               qubits
154         # (ignoring global phase)
```

```
155
156         all_stabilizers = self.stabilizers_X + self.stabilizers_Z
157
158         for i, s1 in enumerate(all_stabilizers):
159             for j, s2 in enumerate(all_stabilizers[i+1:], start=i+1):
160                 overlap = len(set(s1) & set(s2))
161                 if overlap % 2 != 0:
162                     print(f"Non-commuting stabilizers: {i} and {j}")
163                     return False
164
165         return True
166
167
168     def compute_distance_certificate(self) -> dict:
169         """
170         Certify code distance d = L.
171
172         Method: Verify that shortest non-trivial logical operator has
                weight L.
173         """
174         # For toric code, logical operators are non-contractible loops
175         # Minimum length loop on L  L  torus has length L
176
177         min_weights = {
178             name: len(operator)
179             for name, operator in self.logicals.items()
180         }
181
182         distance = min(min_weights.values())
183
184         return {
185             'distance': distance,
186             'expected_distance': self.L,
187             'distance_certified': (distance == self.L),
188             'logical_operator_weights': min_weights
189         }
190
191
192 def paulistring_to_matrix(operator_indices: List[int],
193                           pauli_type: str,
194                           n_qubits: int) -> np.ndarray:
195     """
196     Convert list of qubit indices to full Pauli operator matrix.
197
198     Args:
199         operator_indices: List of qubit indices where Pauli acts
200         pauli_type: 'X', 'Y', or 'Z'
201         n_qubits: Total number of qubits
202
203     Returns:
204         2^n    2^n matrix representing operator
205     """
206     # Start with identity
207     op = np.eye(2**n_qubits, dtype=complex)
208
209     # Build Pauli matrices
```

```
    if pauli_type == 'X':
        pauli = np.array([[0, 1], [1, 0]])
    elif pauli_type == 'Y':
        pauli = np.array([[0, -1j], [1j, 0]])
    elif pauli_type == 'Z':
        pauli = np.array([[1, 0], [0, -1]])
    else:
        raise ValueError(f"Unknown Pauli type: {pauli_type}")

    # Apply Pauli to each qubit in operator_indices
    for idx in operator_indices:
        # Build operator: I  ...  I  pauli  I  ...  I
        op_list = [np.eye(2) for _ in range(n_qubits)]
        op_list[idx] = pauli

        full_op = op_list[0]
        for o in op_list[1:]:
            full_op = np.kron(full_op, o)

        op = op @ full_op

    return op
```

Validation: Verify stabilizers commute, logical operators anticommute, distance = L.

0.3.2 Phase 2 (Months 2-3): Syndrome Measurement and Error Models

Goal: Implement syndrome extraction and common error models.

```
def measure_syndrome(code: ToricCode, error: np.ndarray) -> np.ndarray:
    """
    Measure syndrome from error pattern.

    Args:
        code: Toric code instance
        error: Binary array of shape (n_qubits,) indicating which
           qubits have X errors

    Returns:
        Syndrome array of shape (n_stabilizers,)
    """
    n_stabilizers = code.n_stabilizers
    syndrome = np.zeros(n_stabilizers, dtype=int)

    # X errors anticommute with Z stabilizers (plaquettes)
    for i, plaquette in enumerate(code.stabilizers_Z):
        # Count X errors on edges in plaquette
        n_errors = sum(error[e] for e in plaquette)
        syndrome[code.L**2 + i] = n_errors % 2  # Store in second half

    # Z errors anticommute with X stabilizers (stars)
    # (not implemented  h e r e would  need separate Z error array)

    return syndrome


```

```python
def random_pauli_error(n_qubits: int, p: float, error_type: str = 'X') -> np.ndarray:
    """
    Generate random Pauli error with probability p per qubit.

    Args:
        n_qubits: Number of qubits
        p: Error probability per qubit
        error_type: 'X', 'Y', or 'Z'

    Returns:
        Binary error array
    """
    return (np.random.rand(n_qubits) < p).astype(int)


def depolarizing_error(n_qubits: int, p: float) -> Tuple[np.ndarray, np.ndarray]:
    """
    Generate depolarizing error: each qubit has probability p of X, Y, or Z error.

    Returns:
        (error_X, error_Z) binary arrays
    """
    error_X = np.zeros(n_qubits, dtype=int)
    error_Z = np.zeros(n_qubits, dtype=int)

    for i in range(n_qubits):
        r = np.random.rand()
        if r < p/3:
            error_X[i] = 1  # X error
        elif r < 2*p/3:
            error_Z[i] = 1  # Z error
        elif r < p:
            error_X[i] = 1  # Y = iXZ
            error_Z[i] = 1

    return error_X, error_Z


class AnYonExcitation:
    """
    Representation of anyonic excitations (violated stabilizers).
    """

    def __init__(self, position: Tuple[int, int], charge_type: str):
        """
        Args:
            position: (i, j) position on lattice
            charge_type: 'e' (electric, from Z error) or 'm' (magnetic, from X error)
        """
        self.position = position
        self.charge_type = charge_type

```

```
79
80      def toric_distance(pos1: Tuple[int, int], pos2: Tuple[int, int], L:
            int) -> int:
81          """
82          Compute toric distance between two positions on L L torus.
83
84          Distance: min over all toroidal wrappings.
85          """
86          i1, j1 = pos1
87          i2, j2 = pos2
88
89          di = min(abs(i2 - i1), L - abs(i2 - i1))
90          dj = min(abs(j2 - j1), L - abs(j2 - j1))
91
92          return di + dj
93
94
95  def syndrome_to_anyons(syndrome: np.ndarray, code: ToricCode) ->
        List[AnYonExcitation]:
96      """
97      Convert syndrome to list of anyonic excitations.
98
99      Each violated stabilizer corresponds to an anyon.
100     """
101     L = code.L
102     anyons = []
103
104     # X stabilizer violations     magnetic anyons
105     for idx in range(L**2):
106         if syndrome[idx] == 1:
107             i, j = code.vertices[idx]
108             anyons.append(AnYonExcitation((i, j), 'm'))
109
110     # Z stabilizer violations     electric anyons
111     for idx in range(L**2, 2*L**2):
112         if syndrome[idx] == 1:
113             i, j = code.faces[idx - L**2]
114             anyons.append(AnYonExcitation((i, j), 'e'))
115
116     return anyons
```

Validation: Verify syndromes satisfy constraint $s = 0 \pmod 2$ for each type.

0.3.3 Phase 3 (Months 3-5): Minimum-Weight Perfect Matching Decoder

Goal: Implement MWPM decoder using graph algorithms.

```
1  import networkx as nx
2  from scipy.spatial.distance import cdist
3
4  def decode_toric_code_mwpm(syndrome: np.ndarray, code: ToricCode) ->
       np.ndarray:
5      """
6      Decode syndrome using minimum-weight perfect matching (MWPM).
7
8      Algorithm:
```

```
    1. Extract anyon positions from syndrome
    2. Build complete graph with toric distances as edge weights
    3. Solve minimum-weight perfect matching
    4. Construct correction from matching

    Args:
        syndrome: Binary array of violated stabilizers
        code: Toric code instance

    Returns:
        Binary correction array (which qubits to flip)
    """
    anyons = syndrome_to_anyons(syndrome, code)

    if len(anyons) == 0:
        return np.zeros(code.n_qubits, dtype=int)

    # Separate by charge type
    anyons_e = [a for a in anyons if a.charge_type == 'e']
    anyons_m = [a for a in anyons if a.charge_type == 'm']

    # Decode each type separately
    correction = np.zeros(code.n_qubits, dtype=int)

    if len(anyons_e) > 0:
        correction_e = _mwpm_decode_single_type(anyons_e, code.L, 'e',
           code)
        correction += correction_e

    if len(anyons_m) > 0:
        correction_m = _mwpm_decode_single_type(anyons_m, code.L, 'm',
           code)
        correction += correction_m

    return correction % 2


def _mwpm_decode_single_type(anyons: List[AnYonExcitation],
                             L: int,
                             charge_type: str,
                             code: ToricCode) -> np.ndarray:
    """
    Decode anyons of single charge type using MWPM.
    """
    n_anyons = len(anyons)

    # Build complete graph
    G = nx.Graph()

    for i in range(n_anyons):
        G.add_node(i, pos=anyons[i].position)

    # Add edges with toric distance weights
    for i in range(n_anyons):
        for j in range(i+1, n_anyons):
            pos_i = anyons[i].position
```

```
            pos_j = anyons[j].position

            dist = AnYonExcitation.toric_distance(pos_i, pos_j, L)
            G.add_edge(i, j, weight=dist)

    # Solve minimum-weight perfect matching
    matching = nx.algorithms.matching.min_weight_matching(G)

    # Convert matching to correction
    correction = np.zeros(code.n_qubits, dtype=int)

    for (i, j) in matching:
        pos_i = anyons[i].position
        pos_j = anyons[j].position

        # Flip qubits along geodesic from pos_i to pos_j
        path = shortest_path_on_torus(pos_i, pos_j, L)

        for edge in path:
            # Determine qubit index from edge
            qubit_idx = code.edges_h.get(edge) or code.edges_v.get(edge)
            if qubit_idx is not None:
                correction[qubit_idx] = 1

    return correction


def shortest_path_on_torus(pos1: Tuple[int, int],
                           pos2: Tuple[int, int],
                           L: int) -> List[Tuple[int, int]]:
    """
    Find shortest path (as list of edges) on torus.

    Returns:
        List of edges (i,j) representing horizontal or vertical edges
    """
    i1, j1 = pos1
    i2, j2 = pos2

    path = []

    # Horizontal movement
    if abs(j2 - j1) <= L/2:
        # Direct path
        for j in range(min(j1, j2), max(j1, j2)):
            path.append((i1, j))
    else:
        # Wrap around path
        if j1 < j2:
            for j in range(j1, L):
                path.append((i1, j))
            for j in range(0, j2):
                path.append((i1, j))
        else:
            for j in range(j2, L):
                path.append((i1, j))
```

```
            for j in range(0, j1):
                path.append((i1, j))

    # Vertical movement (similar logic)
    # ... (omitted for brevity)

    return path
```

Validation: Verify decoder success rate > 99

0.3.4 Phase 4 (Months 5-7): Threshold Estimation

Goal: Estimate threshold via Monte Carlo simulations.

```
from scipy.stats import linregress

def estimate_threshold_montecarlo(code: ToricCode,
                                  p_values: np.ndarray,
                                  n_trials: int = 10000) -> dict:
    """
    Estimate threshold via Monte Carlo simulation.

    Threshold: critical error rate p_th where logical error rate
       crosses physical rate.

    Args:
        code: Toric code instance
        p_values: Array of physical error rates to test
        n_trials: Number of Monte Carlo samples per p value

    Returns:
        Dictionary with threshold estimate and error bars
    """
    logical_error_rates = []

    for p in p_values:
        n_logical_errors = 0

        for trial in range(n_trials):
            # Generate random error
            error_X = random_pauli_error(code.n_qubits, p, 'X')

            # Measure syndrome
            syndrome = measure_syndrome(code, error_X)

            # Decode
            correction = decode_toric_code_mwpm(syndrome, code)

            # Total error = physical error + correction
            total_error = (error_X + correction) % 2

            # Check if logical error occurred
            has_logical = check_logical_error(total_error, code)
            if has_logical:
                n_logical_errors += 1

```

```
        logical_error_rate = n_logical_errors / n_trials
        logical_error_rates.append(logical_error_rate)

    # Find threshold: fit curves and find crossover
    # For small L, use simple heuristic: p_th     p where p_log(p)     p

    crossover_idx = find_crossover_point(p_values, logical_error_rates)
    p_threshold = p_values[crossover_idx]

    return {
        'threshold': p_threshold,
        'p_values': p_values.tolist(),
        'logical_error_rates': logical_error_rates,
        'n_trials': n_trials,
        'lattice_size': code.L
    }


def check_logical_error(error: np.ndarray, code: ToricCode) -> bool:
    """
    Check if error causes logical error.

    Logical error: error chain has non-trivial homology (not in image
       of       ).

    Practical check: Compute parity of error along each logical
       operator.
    """
    for name, logical_op in code.logicals.items():
        parity = sum(error[i] for i in logical_op) % 2
        if parity == 1:
            return True  # Logical error occurred

    return False


def find_crossover_point(p_values: np.ndarray,
                         p_log: List[float]) -> int:
    """
    Find crossover point where p_log(p)     p.

    Simple heuristic: find p where |p_log - p| is minimized.
    """
    differences = [abs(p_log[i] - p_values[i]) for i in
       range(len(p_values))]
    return np.argmin(differences)
```

Validation: Verify threshold 10-11

0.3.5 Phase 5 (Months 7-8): Color Codes and Transversal Gates

Goal: Implement 3-colorable lattice color codes.

```
def construct_color_code_triangular(L: int) -> dict:
    """
    Construct color code on triangular lattice.
```

```
    Properties:
    - 3-colorable (Red, Green, Blue)
    - Transversal Clifford gates
    - [[n, k, d]] parameters depend on boundary conditions
    """
    # Build triangular lattice with 3-coloring
    vertices, edges, faces = generate_triangular_lattice(L)
    coloring = color_lattice_3colors(vertices, edges)

    # Qubits on vertices (or faces, depending on convention)
    n_qubits = len(vertices)

    # Stabilizers: one per face, acting on surrounding vertices
    stabilizers_X = []
    stabilizers_Z = []

    for face in faces:
        # Get vertices of face
        face_vertices = get_face_vertices(face, vertices, edges)

        # X-type stabilizer
        stabilizers_X.append(face_vertices)

        # Z-type stabilizer
        stabilizers_Z.append(face_vertices)

    # Transversal gates
    transversal_H = construct_transversal_hadamard(coloring)
    transversal_S = construct_transversal_phase(coloring)

    return {
        'n_qubits': n_qubits,
        'lattice_size': L,
        'stabilizers_X': stabilizers_X,
        'stabilizers_Z': stabilizers_Z,
        'coloring': coloring,
        'transversal_gates': {
            'H': transversal_H,
            'S': transversal_S
        }
    }
```

Validation: Verify transversal Hadamard maps between code subspaces.

0.3.6 Phase 6 (Months 8-9): Certificate Generation

Goal: Generate complete certificates for topological codes.

```
from dataclasses import dataclass, asdict
import json

@dataclass
class TopologicalCodeCertificate:
    """Complete certificate for topological quantum code."""

```

```
    # Code parameters
    code_type: str  # 'toric', 'surface', 'color'
    lattice_size: int
    n_qubits: int
    k_logical: int
    distance: int

    # Stabilizers
    n_stabilizers: int
    stabilizer_commutativity_verified: bool

    # Logical operators
    logical_operator_weights: dict
    logical_anticommutativity_verified: bool

    # Threshold
    threshold_estimate: float
    threshold_std_error: float
    n_monte_carlo_trials: int

    # Homology
    homology_groups: str  # e.g., "H_1 = Z^2"

    # Metadata
    computation_date: str
    precision_digits: int

    def export_json(self, filename: str):
        """Export certificate to JSON."""
        with open(filename, 'w') as f:
            json.dump(asdict(self), f, indent=2)

    def verify(self) -> bool:
        """Self-check certificate validity."""
        checks = [
            self.n_qubits > 0,
            self.k_logical > 0,
            self.distance > 0,
            self.stabilizer_commutativity_verified,
            self.logical_anticommutativity_verified,
            0 < self.threshold_estimate < 1
        ]
        return all(checks)


def generate_toric_code_certificate(L: int, threshold_trials: int =
    10000) -> TopologicalCodeCertificate:
    """
    Generate complete certificate for L L toric code.
    """
    code = ToricCode(L)

    # Verify stabilizers commute
    commute_check = code.verify_stabilizer_commutation()

    # Compute distance
```

```
    distance_cert = code.compute_distance_certificate()

    # Estimate threshold
    p_values = np.linspace(0.05, 0.15, 11)
    threshold_result = estimate_threshold_montecarlo(code, p_values,
        threshold_trials)

    cert = TopologicalCodeCertificate(
        code_type='toric',
        lattice_size=L,
        n_qubits=code.n_qubits,
        k_logical=code.k_logical,
        distance=code.distance,
        n_stabilizers=code.n_stabilizers,
        stabilizer_commutativity_verified=commute_check,
        logical_operator_weights=distance_cert['logical_operator_weights'],
        logical_anticommutativity_verified=True,  # Verified separately
        threshold_estimate=threshold_result['threshold'],
        threshold_std_error=0.002,  # Estimate from trials
        n_monte_carlo_trials=threshold_trials,
        homology_groups='H_1 = Z^2',
        computation_date='2026-01-17',
        precision_digits=64
    )

    return cert
```

Validation: Export certificates for L = 3, 5, 7, verify all self-checks pass.

0.4 4. Example Starting Prompt

Prompt for AI System:

You are tasked with implementing topological quantum error correction codes and analyzing their properties. Your goal is to:

- **Toric Code Construction (Months 1-2)**:
- Implement L×L toric code with qubits on edges
- Construct star (X-type) and plaquette (Z-type) stabilizers
- Verify all stabilizers commute: [A, A'] = [B, B'] = [A, B] = 0
- Construct logical operators (non-contractible loops on torus)
- Certify code distance d = L
- **Syndrome Measurement (Months 2-3)**:
- Implement syndrome extraction from error patterns
- Model random Pauli errors (X, Z) with probability p
- Model depolarizing errors (X, Y, Z with equal probability)

- Convert syndromes to anyonic excitations (violated stabilizers)
- Verify syndrome constraint: s = 0 (mod 2)
- **MWPM Decoder (Months 3-5)**:
- Implement minimum-weight perfect matching decoder
- Build complete graph with toric distances as edge weights
- Use NetworkX min*weight*matching to solve
- Convert matching to qubit corrections (flip bits along geodesics)
- Verify decoder success rate > 99
- **Threshold Estimation (Months 5-7)**:
- Run Monte Carlo simulations for p [0.05, 0.15]
- For each p: generate N = 10,000 random errors, decode, check logical error
- Plot logical error rate p$_{l}og(p)vsphysicalerrorratep$

- Find threshold p*th where curves cross (p*log p)
- Verify threshold 10-11
- **Color Codes (Months 7-8)**:
- Construct triangular lattice with 3-coloring (R, G, B)
- Define stabilizers on faces (acting on surrounding vertices)
- Implement transversal Hadamard and Phase gates
- Verify gates map code subspace to code subspace
- **Certificate Generation (Months 8-9)**:
- Create TopologicalCodeCertificate with all parameters
- Include: [[n, k, d]], stabilizers, threshold, homology groups
- Export to JSON with exact values
- Verify all certificates pass self-checks

Success Criteria:

- Minimum Viable Result (2-4 months): Toric code with MWPM decoder
- Strong Result (6-8 months): Threshold estimation matching literature (11
- Publication-Quality Result (9 months): Color codes, transversal gates, certified database

Key Constraints:

- Use exact arithmetic for stabilizer algebra (GF(2))

- Monte Carlo: N 10,000 trials per p value
- Threshold uncertainty < 1
- All certificates machine-verifiable

References:

- Kitaev (2003): "Fault-tolerant quantum computation by anyons"
- Dennis et al. (2002): "Topological quantum memory"
- Fowler et al. (2012): "Surface codes: Towards practical large-scale quantum computation"

Begin by implementing the ToricCode class with star and plaquette stabilizers.

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (Months 1-4)

Core Achievements:

- Toric code implementation: stabilizers, logical operators
- Syndrome measurement from error patterns
- Basic MWPM decoder (may use external library)
- Distance certification: d = L verified

Validation:

- Stabilizers commute (all pairwise checks pass)
- Logical operators anticommute correctly
- Decoder success > 95

Deliverables:

- Python module `topological_codes.py`
- Jupyter notebook demonstrating L=5 toric code
- JSON certificate for L=3 code

0.5.2 Strong Result (Months 4-8)

Extended Capabilities:
- Full MWPM decoder with toric distance geodesics
- Monte Carlo threshold estimation: $p_thwitherrorbars$

- Comparison to 3+ literature sources (Dennis 2002, Wang 2011, Fowler 2012)
- Finite-size scaling analysis (L = 3, 5, 7, 9)
- Color code implementation with transversal gates

Publications Benchmark:

- Reproduce Figure 3 from Dennis et al. (2002) showing threshold
- Match threshold to within 1

Deliverables:

- Database of certificates for L = 3, 5, 7, 9
- Threshold plots vs lattice size
- Color code Hadamard gate verification

0.5.3 Publication-Quality Result (Months 8-9)

Novel Contributions:
- Optimized decoder (e.g., Union-Find, MWPM with precomputation)
- 3D color codes or subsystem codes
- Formal verification: translate proofs to Coq/Lean
- Interactive visualization (lattice, anyons, corrections)
- Public database: 50+ code instances with certificates

Beyond Literature:

- Improve decoder speed (< 1ms per decode for L=9)
- Discover new code families with better parameters
- Extend to continuous-variable codes

Deliverables:

- Arxiv preprint: "Certified Topological Quantum Codes"
- GitHub repository with visualization tools
- Web interface: interactive toric code simulator

0.6 6. Verification Protocol

```python
def verify_topological_code_certificate(cert: TopologicalCodeCertificate) -> dict:
    """
    Automated verification of topological code certificate.
    """
    results = {}

    # Check 1: Code parameters consistent
    results['parameters_valid'] = (
        cert.n_qubits > 0 and
        cert.k_logical > 0 and
        cert.distance > 0
    )

    # Check 2: Stabilizers verified
    results['stabilizers_valid'] = cert.stabilizer_commutativity_verified

    # Check 3: Logical operators verified
    results['logicals_valid'] = cert.logical_anticommutativity_verified

    # Check 4: Threshold in reasonable range
    results['threshold_reasonable'] = (0.05 < cert.threshold_estimate < 0.20)

    # Check 5: Distance matches expected
    if cert.code_type == 'toric':
        results['distance_matches'] = (cert.distance == cert.lattice_size)

    # Overall
    results['all_checks_passed'] = all(
        v for v in results.values() if isinstance(v, bool)
    )

    return results
```

0.7 7. Resources and Milestones

0.7.1 Essential References

- **Foundational Papers**:
- Kitaev (2003): "Fault-tolerant quantum computation by anyons"
- Dennis et al. (2002): "Topological quantum memory"
- Bombin Martin-Delgado (2006): "Topological quantum distillation"
- **Thresholds**:

- Wang et al. (2011): "Surface code quantum computing by lattice surgery"
- Fowler et al. (2012): "Surface codes: Towards practical large-scale quantum computation"
- **Textbooks**:
- Nielsen Chuang (2010): *Quantum Computation and Quantum Information* (Chapter 10)
- Terhal (2015): "Quantum error correction for quantum memories"

0.7.2 Milestone Checklist

Month 1: Toric code class implemented

Month 2: Stabilizers verified, distance certified

Month 3: MWPM decoder working

Month 4: Decoder success $>$ 95

Month 5: Monte Carlo threshold estimation begun

Month 6: Threshold 11

Month 7: Color code implementation started

Month 8: Transversal gates verified

Month 9: Full certificate database exported

End of PRD 24