The Tri-Quarter Framework: Radial Dual Triangular Lattice Graphs with Exact Bijective Dualities and Equivariant Encodings via the Inversive Hexagonal Dihedral Symmetry Group $\mathbb{T}_{24}$

Nathan O. Schmidt

Cold Hammer Research & Development LLC, Eagle, Idaho, USA

nate.o.schmidt@coldhammer.net

June 10, 2026

Abstract

The Tri-Quarter Framework (TQF) unleashes a radial dual triangular lattice graph with unified complex-Cartesian-polar coordinates, structured orientation phase pair assignments for directional labeling, and topological zones to build exact bijective mappings without approximations. By modifying the Eisenstein integer lattice and establishing combinatorial duality for radial separation, Escher reflective duality for zone swapping, and bijective self-duality for reversible transformations, the discretized framework leverages the lattice graph's order-6 rotational symmetry to natively support angular sectors, modular decompositions, equivariant encodings, and trihexagonal six-coloring for conflict-free parallel algorithms. At this discretized framework's core is the *Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group* $\mathbb{T}_{24}$—the order-24 direct product $D_6 \times \mathbb{Z}_2$ with ι_r as a central involution—which exploits rotational, reflective, and inversive symmetries to unlock these bijective transformations with exact precision. As an abstract group, $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$ is the classical centrosymmetric hexagonal point group D_{6h}; *our contribution is its concrete realization* as a circle inversion action on Λ_r, together with a radial dual construction whose zone bijections, constant-size balanced separator, and equivariant encodings hold exactly by design rather than by approximation. We provide formal proofs of these dualities, along with numerous step-by-step examples, and demonstrate practical efficiency through benchmarked simulations. For inversion-based path mirroring via bijections, we achieve measured speedups of 1.6–1.8× with bitwise-exact agreement against full recomputation. For symmetry-reduced clustering, computed in exact rational arithmetic, the measured speedups are a steady 3.5–4.0× (peaking near 4.0×), below the idealized $\mathbb{Z}_6$-symmetry ceiling of 6×—the gap reflecting the uniform per-vertex cost of exact rational arithmetic and orbit bookkeeping—while the orbit-reduced coefficient reproduces the full graph computation *exactly*, as the same rational number, at every R. Most significantly, the equivariant trihexagonal six-coloring partitions the lattice graph into six conflict-free independent sets that map directly onto data parallel hardware: on a consumer NVIDIA GeForce RTX 4060 laptop GPU, a color-ordered relaxation sweep achieves a median ∼8× (ranging 6.6–8.6× across five sessions) over a single threaded CPU baseline at ∼290k vertices, where the GPU runtime remains nearly flat as the lattice graph grows while the CPU cost scales linearly—so the advantage widens with scale. This work advances scalable computations on symmetric structures, with applications in computational geometry, graph traversals, tiling, clustering, and conflict-free data parallel computation. This work aims to solidify a practical mathematical and computational foundation for both classical and non-classical computing paradigms—targeting future integrations in complex emergent systems that harness intricate "superposition-like" symmetries to advance symmetry-aware algorithms and data structures across diverse computing architectures.

Notation

The following conventions are used throughout this paper.

- $X = \mathbb{C} \setminus \{(0,0)_C\}$ — the origin-punctured complex plane.
- $\vec{x}, \vec{v}, \vec{p}, \vec{\omega}$ — vectors and lattice graph vertices, written with overhead arrows throughout.
- $x_{\mathbb{R}}$, $x_{\mathbb{I}}$ — scalar real and imaginary components of $\vec{x}$.
- $\vec{x}_{\mathbb{R}} = (x_{\mathbb{R}}, 0)_C$, $\vec{x}_{\mathbb{I}} = (0, x_{\mathbb{I}})_C$ — axis-aligned component vectors of $\vec{x}$.
- $||\vec{x}||$ — Euclidean norm of $\vec{x}$.
- $\langle \vec{x} \rangle$ — phase (argument) of $\vec{x} \in X$, valued in $[0, 2\pi)$.
- $\phi(\vec{x}) = (\langle \vec{x}_{\mathbb{R}} \rangle, \langle \vec{x}_{\mathbb{I}} \rangle)_\phi$ — phase pair assignment of $\vec{x}$, encoding directional orientation.
- $(\cdot\,, \cdot)_C$ — Cartesian coordinate pair; $(\cdot\,, \cdot)_P$ — polar coordinate pair; $(\cdot\,, \cdot)_\phi$ — phase pair.
- $X_{-,r}$, T_r, $X_{+,r}$ — inner zone, boundary circle, and outer zone of X for radius $r > 0$.
- $\iota_r(\vec{x}) = r^2 \vec{x} / ||\vec{x}||^2$ — circle inversion map; an involution fixing T_r.
- $\mathcal{R}_\theta$ — rotation operator by angle θ about the origin.
- $\mathcal{R}_\pi$ — the *half-turn*; rotation by π radians, equivalently point reflection through the origin ($\vec{x} \mapsto -\vec{x}$).
- S_t — angular sector for $t \in \mathbb{Z}_6$, spanning phases $[t\pi/3, (t{+}1)\pi/3)$.
- $\vec{\omega}_0 = 1$, $\vec{\omega}_1 = e^{i\pi/3}$ — basis vectors of the base triangular lattice L.
- L — base triangular lattice on X, generated by $\vec{\omega}_0$ and $\vec{\omega}_1$.
- $\Lambda_r = (V_r, E_r)$ — radial dual triangular lattice graph with admissible inversion radius $r > 0$.
- Λ_r^R — truncated version of Λ_r with truncation radius $R \gg r$.
- $V_{+,r}$, $V_{T,r}$, $V_{-,r}$ — outer, boundary, and inner zone vertex sets of Λ_r.
- $\Lambda_{+,r}$, $\Lambda_{T,r}$, $\Lambda_{-,r}$ — outer, boundary, and inner zone subgraphs of Λ_r.
- $F_r = F_{+,r} \sqcup F_{T,r} \sqcup F_{-,r}$ — face set (triangular 2-cells) of Λ_r, partitioned into outer, boundary, and inner zone faces (mirrors the vertex zone trichotomy).
- D_6 — dihedral group of order 12; point symmetry group of L, with six rotations and six reflections.
- $\mathbb{Z}_6$ — cyclic group of order 6; generated by rotation $\mathcal{R}_{\pi/3}$.
- $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$ — Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group of order 24, with $\mathbb{Z}_2 = \mathrm{gen}(\iota_r)$.
- $d_{hops}(\vec{v}_i, \vec{v}_j)$ — discrete dual metric: shortest-path hop distance within a zone subgraph.
- $d_{\mathbb{H}}(\vec{v}_i, \vec{v}_j)$ — continuous dual metric: hyperbolic geodesic distance.
- $u(\vec{v}_i) = |\log(\|\vec{v}_i\|/r)|$ — (non-negative) boundary-relative depth of vertex $\vec{v}_i$.
- $\tilde{u}(\vec{v}_i) = \log(\|\vec{v}_i\|/r)$ — signed boundary-relative depth (positive on $\Lambda_{+,r}$, negative on $\Lambda_{-,r}$).
- $s_6 : V(\Lambda_r) \to \mathbb{Z}_6$ — angular sector-based six-encoding; assigns each vertex its sector index.
- $e_6 : \Lambda_r \to \{0, \ldots, 5\}$ — trihexagonal six-coloring; proper equivariant six-coloring of Λ_r.
- $c : \Lambda_r \to \{0, 1, 2\}$ — proper three-coloring of Λ_r inherited from the base triangular lattice L.
- acosh, asinh, atanh — inverse hyperbolic functions.
- $C_k(\Lambda_r; \mathbb{Z}_2)$ — space of k-chains: $\mathbb{Z}_2$-valued labelings (bit-vectors) of the k-cells of Λ_r (vertices for k=0, edges for k=1, faces for k=2).
- ∂_k, ∂_k^* — boundary operator $C_k \to C_{k-1}$ (a cell to its rim) and its transpose coboundary $C_k \to C_{k+1}$.
- $[S]$ — indicator chain of a cell-set S; the bit-vector equal to 1 on the cells of S and 0 elsewhere.
- $\rho_k(g)$ — action of $g \in \mathbb{T}_{24}$ on C_k; the permutation of k-cells induced by g, written as a $\mathbb{Z}_2$-linear map.
- $H_k(\Lambda_r; \mathbb{Z}_2) = \ker \partial_k / \mathrm{im}\, \partial_{k+1}$ — $\mathbb{Z}_2$-homology, with cycle space $\ker \partial_1$, boundary space $\mathrm{im}\, \partial_2$, and cut space $\mathrm{im}\, \partial_0^*$.

1 Introduction

The pursuit of a first-principles approach to advanced computing paradigms—targeting both classical and non-classical models—necessitates robust mathematical and computational foundations that prioritize exactness and symmetry exploitation, particularly in models that harness intricate, "superposition-like" behaviors through discrete structures. Traditional methods in graph theory and computational geometry often rely on approximations or lack native support for radial dualities, which limits efficiency in symmetry-aware applications such as the optimization of networks [1], distributed systems [2], and parallel algorithms [3]—where balanced, invertible operations are key.

Existing frameworks provide valuable tools but often lack built-in support to efficiently process:

- Radial data and radial queries.
- Directional classifications via structured orientation phase pairs (or equivalent angular labeling).
- Exact bijective swaps via inversion for symmetric transformations.
- Exact integer-arithmetic comparisons that avoid floating-point drift in precision-critical radial computations.
- Symmetry-equivariant labelings and encodings (e.g., proper colorings, sector indices, and chain-level boundary operators) that remain consistent under the framework's group actions.

For instance, quadtrees are designed for spatial partitioning via recursive nested boxes but are not optimized for radial patterns, which often incur costly conversions [4]. Voronoi diagrams excel at proximity-based partitioning to form territorial maps around points but do not natively support bijective swaps or angular classifications [5]. Discrete conformal mappings preserve angles and have seen advances in conformally symmetric lattices [6], but they typically overlook origin-centered inversion for duality and directional labeling, and thus miss key opportunities for scalable, reversible transformations in lattice-based computations [7]. On the precision front, robust computational geometry libraries (e.g., Shewchuk's adaptive predicates [8] and CGAL's exact computation paradigm [9]) do achieve exactness, but they treat it as a runtime correctness patch bolted on top of a floating-point design—wrapping each predicate in adaptive precision filters or lazy exact evaluators—rather than as a structural property of the underlying lattice, which leaves radial admissibility, zone membership, and boundary tests to pay per-predicate overhead on every query. Likewise, group-equivariant graph neural networks and the broader geometric deep learning program [10, 11] have shown that equivariant labelings preserve information under group actions and yield real efficiency gains, but they typically *learn* approximately-equivariant features over a generic graph substrate that itself carries no inherent group structure—whereas if the substrate is built around a faithful group action from the outset, an equivariant labeling can be constructed exactly rather than approximated, and exploited in $O(1)$ per vertex without learning anything. These gaps underscore the need for a unified approach that fundamentally embeds radial symmetries and exact bijections by design.

In this paper, we resolve these gaps by extending the Tri-Quarter Framework (TQF) [12] from the continuous domain to the discrete domain with radial dual triangular lattice graphs. The discrete substrate of our construction is inspired by the *Eisenstein integers* $\mathbb{Z}[\omega]$ with $\omega = e^{2\pi i/3}$, introduced by Gotthold Eisenstein in 1844 in his proof of the law of cubic reciprocity [13]. Eisenstein integers form a triangular lattice in the complex plane whose unit group $\{\pm 1, \pm\omega, \pm\omega^2\}$ realizes exactly the order-6 rotational symmetry we exploit. Together with their unique factorization, integer-valued norm $N(a+b\omega) = a^2 - ab + b^2$, and natural alignment with $\pi/3$-angular structure, they make a uniquely well-suited algebraic foundation for the exact, symmetry-preserving discrete computations we develop here. The framework structures the complex plane and its coordinate system to natively support radial symmetries and inversions by design. In doing so, we establish combinatorial duality for separation, Escher reflective duality for swapping, and bijective self-duality for reversibility—all while preserving exact mappings without approximations. This novel discretization enables scalable, symmetry-preserving computations that directly address the identified gaps, supported by formal proofs, numerical results, and practical efficiency demonstrations. Key contributions include:

- A unified complex-Cartesian-polar coordinate system with structured orientation phase pair assignments and angular sectors for rapid directional labeling, zones for exact bijections, and integer-arithmetic norm comparisons that eliminate floating-point error in admissibility, zone membership, and boundary tests.

- Proofs of dualities that preserve symmetries and encodings, together with a constant-size balanced separator theorem (Theorem 3.28)—an R-independent, perfectly balanced vertex separator of size $6k$ that precisely splits Λ_r^R into its two "twin" zones—and a truncation-stability guarantee that makes every finitely-supported computation on the infinite lattice Λ_r *exact* on every sufficiently large finite truncation Λ_r^R.

- The *Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group* $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$, an order-24 structure with ι_r as a central involution, which provides a unified setting for the framework's rotational, reflective, and inversive symmetries. We discover that the abstract group $\mathbb{T}_{24}$ is actually the classical centrosymmetric hexagonal point group D_{6h} $(6/mmm)$ [14]—the contribution is not the group itself but rather its concrete realization, in which the central involution is the circle inversion ι_r that operates on Λ_r (see Remark 4.17).

- Dual metrics and equivariant encodings, such as trihexagonal six-coloring, for efficient algorithms.

- A face trichotomy and a $\mathbb{T}_{24}$-equivariant $\mathbb{Z}_2$-chain complex on vertices, edges, and faces, which allows cycle space and cut space algorithms (e.g., for network reliability, planar duality, or parity-based scheduling) to inherit the same symmetry-reduction benefits that were already established for vertex-based computations, and which exposes a homological obstruction (the H_1 puncture class) that detects whether a configuration wraps the origin.

- Exactness as a structural property of the discrete layer, which is achieved by design (rather than as a runtime patch). Integer-only norm comparisons remove floating-point error from the framework's membership and boundary tests. An integer angular-sector index—a lattice cross product sign test on (a, b)—makes the sector partition exactly uniform and the trihexagonal six-coloring exactly $\mathbb{T}_{24}$-equivariant. And exact rational computation of the clustering coefficient makes the symmetry reduction bitwise-lossless.

- Simulation experiments that demonstrate:

    - 1.6–1.8x speedups in inversion-based path mirroring, which we verify to be bitwise-exact against the corresponding full recomputation (e.g., to accelerate lattice graph neural networks by inverting embeddings to supercharge AI-driven pattern recognition and data structure optimization).
    - Symmetry-reduced clustering via $\mathbb{Z}_6$ orbit-transversal replication with a steady $\sim$3.5–4.0x speedup (that peaks near $\sim$4.0x) below the theoretical $\sim$6x $\mathbb{Z}_6$-symmetry ceiling—computed in exact rational arithmetic so that the orbit-replicated coefficient reproduces the full graph computation bitwise (e.g., for efficient motif analysis in symmetric networks).
    - Conflict-free data parallel computation via the equivariant trihexagonal six-coloring, where six independent color classes map directly onto GPU hardware: a color-ordered relaxation sweep reaches a median $\sim$8x (ranging 6.6–8.6x across five sessions) over a single threaded CPU baseline on a consumer NVIDIA GeForce RTX 4060 laptop GPU, where the advantage widens as the lattice graph grows (e.g., for equivariant message passing in graph neural networks and large-scale lattice diffusion).

We state the scope of this paper's contributions plainly. The mathematical substrate upon which the TQF resides is classical: circle inversion and its Möbius geometry [15, 16], the Eisenstein integers and their order-6 unit group [13], the dihedral point symmetry of the triangular lattice and its associated colorings [17, 18], the centrosymmetric hexagonal point group D_{6h} [14], and the standard $\mathbb{Z}_2$ graph-homology chain complex [18] are all long established and are used here as building blocks rather than claimed as new. The contribution of this first principles approach is the specific assembly of these blocks into the radial dual triangular lattice graph Λ_r, the formulation and proof of its three dualities on that structure, the realization of $\mathbb{T}_{24}$ as a group acting on Λ_r with circle inversion as its central involution, the equivariant encodings tailored to that action, and the empirical results that demonstrate the symmetry-reduction. Throughout this work, we aim to attribute each applied component to its source and to delimit the novelty claim to the construction and its application.

Rooted in our discretized TQF is $\mathbb{T}_{24}$, a unified order-24 algebraic structure that extends the lattice graph's dihedral symmetries with circle inversion to forge equivariant operations and reversible mappings across angular sectors and zones. We structure this paper as follows—in Section 2, we review the continuous TQF to set the stage for discretization; in Section 3, we discretize the TQF to radial dual triangular lattice graphs; in Section 4, we prove dualities, introduce $\mathbb{T}_{24}$, and explore their implications; in Section 5, we introduce encodings for computational tasks; in Section 6, our inversion-based path mirroring, symmetry-reduced clustering, and trihexagonal six-coloring simulation experiment benchmarks demonstrate consistent speedups for the TQF approach over the standard approach; and in Section 7, we conclude with a brief recap and projected future directions of exploration.

2 Tri-Quarter Framework Fundamentals

In this section, we review the fundamentals of our continuous TQF as introduced in [12]—by focusing on the complex plane and its topological properties, we create a foundation that is both mathematically rigorous and computationally practical to ultimately set the stage for the framework's upcoming discretization into the radial dual triangular lattice graphs of Section 3 and the applications beyond. Here, we summarize and extend the key elements of the framework by highlighting its utility for problems that require unified coordinates, structured orientation phase pair directional classification, and symmetric transformations.

2.1 Unified Coordinate System on X

To build the TQF, we start with the complex plane $\mathbb{C}$ equipped with the standard Euclidean topology, which provides a natural setting for coordinates and distances. For a complex number $\vec{x} = x_{\mathbb{R}} + x_{\mathbb{I}}i \in \mathbb{C}$, we represent it as a vector $\vec{x} = \vec{x}_{\mathbb{R}} + \vec{x}_{\mathbb{I}} \in \mathbb{C}$, where $\vec{x}_{\mathbb{R}} = (x_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$ and $\vec{x}_{\mathbb{I}} = (0, x_{\mathbb{I}})_C \in \{0\} \times \mathbb{R}$ in Cartesian form. This decomposition emphasizes the interchangeability of 2D points, complex numbers, and vectors, which enables the directional classifications and symmetries that are central to the framework. A subtle—yet *crucial*—key property of this decomposition is to treat components as axis-aligned orthogonal vectors instead of just scalars—this provides the foundation for phase pair assignments, as detailed in [12].

To avoid singularities (and reinforce the well-defined structured orientation phase assignments in the subsequent Subsection 2.2), we define the space $X = \mathbb{C} \setminus \{(0,0)_C\}$ to exclude the origin because phase pairs are undefined at the origin. Each point $\vec{x} \in X$ is characterized by its non-zero Euclidean norm $||\vec{x}|| = \sqrt{x_{\mathbb{R}}^2 + x_{\mathbb{I}}^2} \in \mathbb{R}_+$ and phase $\langle \vec{x} \rangle \in [0, 2\pi)$. So for $\vec{x}$ this gives synchronized polar coordinates $(||\vec{x}||, \langle \vec{x} \rangle)_P$ and Cartesian coordinates $(x_{\mathbb{R}}, x_{\mathbb{I}})_C = (||\vec{x}|| \cos\langle \vec{x} \rangle, ||\vec{x}|| \sin\langle \vec{x} \rangle)_C$ to yield a unified coordinate system:

$$\vec{x} = \vec{x}_{\mathbb{R}} + \vec{x}_{\mathbb{I}} = (x_{\mathbb{R}}, x_{\mathbb{I}})_C = (||\vec{x}|| \cos\langle \vec{x} \rangle, ||\vec{x}|| \sin\langle \vec{x} \rangle)_C = (||\vec{x}||, \langle \vec{x} \rangle)_P. \tag{1}$$

For example, if $\vec{x} = (1, 0)_C$, then we have norm $||\vec{x}|| = 1$ and phase $\langle \vec{x} \rangle = 0$, so it aligns with the positive real axis in both representations. This unification supports consistent directional assignments and the bijective mappings that are central to the framework.

This exclusion of the origin has minimal (or negligible) negative impact on origin-centered algorithms in the continuous setting because phase-based operations naturally avoid singularities by design. In fact, this exclusion enables a structured set of symmetries and computational properties—supporting exact bijective mappings via circle inversion that swap inner and outer zones while preserving directional labels, and enabling modular decompositions with order-6 rotational invariance that align with lattice graph structures, as articulated in the dualities and encodings ahead. Algorithms initiated at the origin in the continuous setting can be defined by limits along radial sequences approaching the puncture. (Note: As we will see in the discrete extension of Section 3, the base triangular lattice L inherently excludes the origin as well to align seamlessly with X.)

2.2 Phase Pair Assignments on X

To exploit, formalize, and leverage the structured orientation directional classification and symmetry of X, we assign phase pairs to each point $\vec{x} \in X$ based on the signs of its real and imaginary components. These *quadrant phase pair assignments* categorize points according to their position in the four quadrants of X as defined in Table 1. The phase pair $\phi(\vec{x}) = (\langle \vec{x}_{\mathbb{R}} \rangle, \langle \vec{x}_{\mathbb{I}} \rangle)_\phi$ encodes the directional orientation of $\vec{x}$ relative to the real and imaginary axes.

TABLE 1. Quadrant Phase Pair Assignments

Quadrant	**Condition**	**Phase Pair**
I	$\langle \vec{x} \rangle \in (0, \pi/2)$	$(0, \pi/2)_\phi$
II	$\langle \vec{x} \rangle \in (\pi/2, \pi)$	$(\pi, \pi/2)_\phi$
III	$\langle \vec{x} \rangle \in (\pi, 3\pi/2)$	$(\pi, 3\pi/2)_\phi$
IV	$\langle \vec{x} \rangle \in (3\pi/2, 2\pi)$	$(0, 3\pi/2)_\phi$

For points that exist precisely on a coordinate axis (when exactly one component is zero), we define *axis boundary phase pair assignments* to ensure consistency as defined in Table 2. These assignments handle the boundary cases where $\vec{x}$ aligns with the positive or negative real or imaginary axis, which implies unique phase pair assignment to these axis bound points. Hence, together with the quadrant assignments, these guarantee unique phase pairs across all four quadrants and four axis boundaries.

TABLE 2. Axis Boundary Phase Pair Assignments

Axis	**Condition**	**Phase Pair**
East	$\langle \vec{x} \rangle = 0$	$(0, 0)_\phi$
North	$\langle \vec{x} \rangle = \pi/2$	$(\pi/2, \pi/2)_\phi$
West	$\langle \vec{x} \rangle = \pi$	$(\pi, 0)_\phi$
South	$\langle \vec{x} \rangle = 3\pi/2$	$(\pi/2, 3\pi/2)_\phi$

For example, the point $\vec{x} = (1, 1)_C$—with phase $\langle \vec{x} \rangle = \pi/4$—resides in Quadrant I, so it gets assigned the phase pair $\phi(\vec{x}) = (0, \pi/2)_\phi$ as per Table 1, which safeguards unique directional classification without ambiguity. Moreover, for a non-axis radial ray emanating from (but not including) the origin and traversing across Quadrant I at phase $\pi/6$, consider points like $k \cdot (3/2, \sqrt{3}/2)_C$ for $k > 0$—the assigned phase pair $(0, \pi/2)_\phi$ remains constant along the radial ray—a property that holds for all radial rays and extends to the discrete case in the upcoming Section 3.

The results of Tables 1–2 are then consolidated and formalized into the *phase assignment rules* of Table 3 to uphold consistency across all points in X. This secures a combinatorial classification of directions in X that partitions points into directional categories based on their quadrant or axis alignment—essential for computational analysis because it features an efficient structured orientation with consistent labeling of directions, which will be critical when discretizing to a lattice graph where exact angular assignments align with geometric symmetries.

TABLE 3. Phase Assignment Rules

Component	**Condition**	**Phase**
$\vec{x}_\mathbb{R}$	$x_\mathbb{R} > 0$	0
	$x_\mathbb{R} = 0$	$\pi/2$
	$x_\mathbb{R} < 0$	π
$\vec{x}_\mathbb{I}$	$x_\mathbb{I} > 0$	$\pi/2$
	$x_\mathbb{I} = 0$	0
	$x_\mathbb{I} < 0$	$3\pi/2$

These rules in Table 3—based entirely on component signs—ensure the assignment of unique phase pairs for all cases and avoid ambiguity at axis boundaries where standard argument functions might yield overlapping or undefined outputs, as detailed in the original TQF [12]. More specifically, the assignment $\langle \vec{x}_\mathbb{R} \rangle = \frac{\pi}{2}$ when $x_\mathbb{R} = 0$, and the assignment $\langle \vec{x}_\mathbb{I} \rangle = 0$ when $x_\mathbb{I} = 0$, facilitate uniqueness for axis boundary cases. In other words, for example, our choice to assign $\langle \vec{x}_\mathbb{R} \rangle = \frac{\pi}{2}$ when $x_\mathbb{R} = 0$ equips us with the means to bypass conflicts with real-axis phases (0 or π) and rapidly distinguish the vertical axis from the

horizontal axis (without additional case analysis). Phase pair assignment is computationally efficient because it requires only $O(1)$ time per point via sign checks on components, which supports scalability in large-scale applications like spatial data structures—e.g., in high-throughput real-world systems for signal processing and data classification, where the computational cost of executing each conditional check matters. These assignments imply constant-time directional binning, which is crucial for the upcoming mod 6 angular sector partitioning of the discrete lattice graphs in Section 3.

2.3 Topological Zones on X

To catalyze radial separation and support bijective mappings, we partition X into three distinct topological zones based on the Euclidean norm $||\vec{x}||$ for any radius $r > 0$ and $\vec{x} \in X$ as per the TQF. These zones are:

(1) The *inner zone* $X_{-,r}$, which contains all points closer to the origin than the radius r.

(2) The *boundary zone* T_r (a circle of radius r), which contains all points exactly at the radius r.

(3) The *outer zone* $X_{+,r}$, which contains all points farther from the origin than the radius r.

The partition is defined by a norm trichotomy to ensure that, for any $\vec{x} \in X$, exactly one of the following zone conditions holds:

$$\begin{aligned} ||\vec{x}|| < r &\iff \vec{x} \in X_{-,r} \\ ||\vec{x}|| = r &\iff \vec{x} \in T_r \\ ||\vec{x}|| > r &\iff \vec{x} \in X_{+,r}, \end{aligned} \tag{2}$$

where the zones are mutually disjoint and cover X entirely:

$$\begin{aligned} X_{+,r} &= \{\vec{x} \in X : ||\vec{x}|| > r\} \\ T_r &= \{\vec{x} \in X : ||\vec{x}|| = r\} \\ X_{-,r} &= \{\vec{x} \in X : ||\vec{x}|| < r\}, \end{aligned} \tag{3}$$

such that $X_{-,r} \cap T_r = T_r \cap X_{+,r} = X_{-,r} \cap X_{+,r} = \emptyset$ and $X_{-,r} \cup T_r \cup X_{+,r} = X$. The circular boundary zone T_r acts as a separator between the inner zone $X_{-,r}$ (an open disk punctured at the origin) and the outer zone $X_{+,r}$ (the exterior of the circle). This trichotomy partitions the four quadrants of X into three radial zones—a "Tri-Quarter"—to solidify a topological foundation for duality mappings [19]. Throughout, $T_r \subset X$ denotes the continuous boundary circle $\{\vec{x} \in X : \|\vec{x}\| = r\}$; in the discrete setting of Section 3 we will write $V_{T,r} = L \cap T_r$ for the lattice's intersection with this circle, reserving T_r itself exclusively for the continuous circle.

Let $\mathcal{R}_\theta$ be the rotation operator by angle θ. For $\vec{x} \in X$ we have $||\mathcal{R}_\theta(\vec{x})|| = ||\vec{x}||$ (norm invariance under rotation), so zone assignment (inner/boundary/outer) remains unchanged. Thus, we have phase shifts by θ. So for $\theta = \pi/2$ we have $\mathcal{R}_{\pi/2}$ for quadrant rotations that preserve sign categories and consistently cycle quadrant-based phase pairs (e.g., $(0, \pi/2)_\phi \to (\pi, \pi/2)_\phi \to (\pi, 3\pi/2)_\phi \to (0, 3\pi/2)_\phi \to (0, \pi/2)_\phi \to \ldots$, repeating cyclically) because the rules only depend on the component signs, which consistently rotate.

This norm-based partitioning leverages X's rotational invariance around the origin $(0, 0)_C$, where properties remain unchanged under rotations. The phase pairs from Subsection 2.2 complement this by providing discrete angular labels to ensure that bijective transformations (e.g., the circle inversion in Subsection 2.4) preserve directional information. This structure suits radial applications (e.g., polar Voronoi [20]) to reduce conversion errors. This radial partitioning contrasts with planar graph dualities—where vertices map to faces in embeddings (as in Whitney's combinatorial duality [21])—by emphasizing origin-centered norm separation rather than face-vertex correspondences.

2.4 Escher Reflective Duality on X

The TQF characterizes a reflective duality to establish exact bijective mappings between $X_{-,r}$ and $X_{+,r}$, which is effectuated by the separator T_r. For any $r > 0$, we utilize the well-known *circle inversion map*

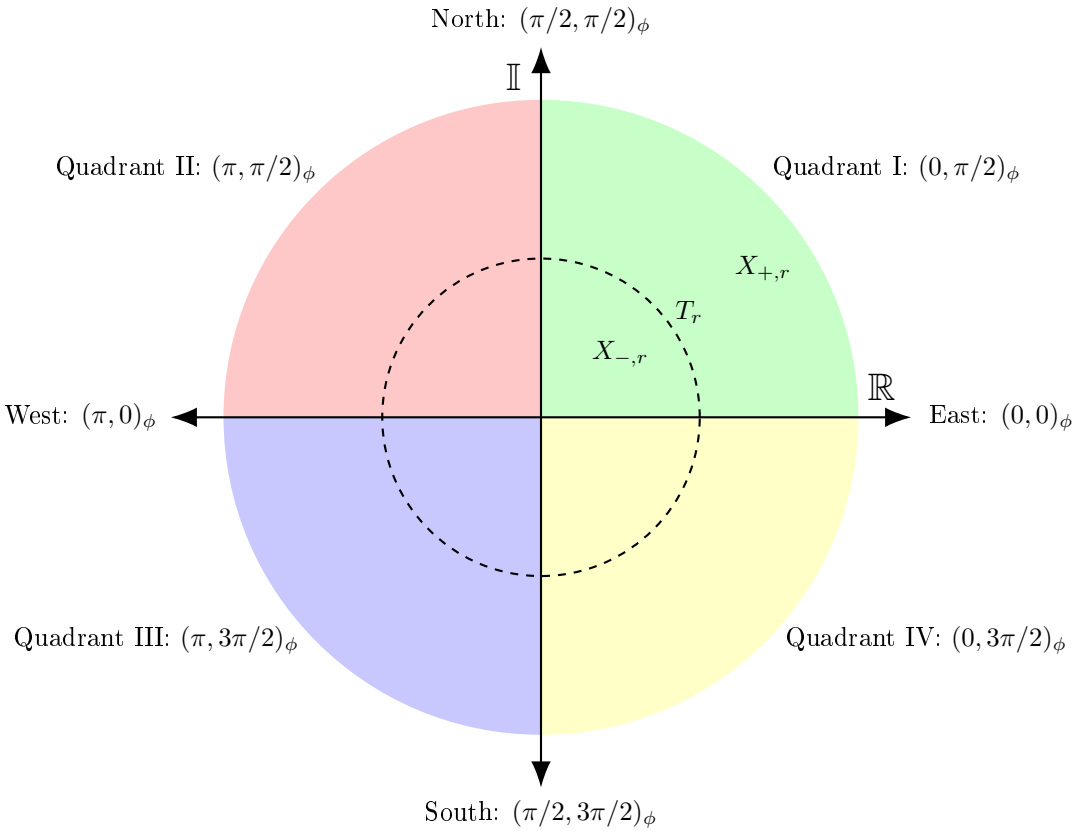

FIGURE 1. Continuous zones, quadrants, and axes in the (origin-punctured) complex plane X, with phase pairs labeled as per Tables 1–2. The dashed circle represents the boundary zone T_r that separates the inner zone $X_{-,r}$ and outer zone $X_{+,r}$.

$\iota_r : X \to X$ [15, 16] as

$$\iota_r(\vec{x}) = \frac{r^2 \vec{x}}{||\vec{x}||^2}, \tag{4}$$

where $\vec{x} \in X$ and $||\vec{x}||$ is the Euclidean norm that encodes its radial distance from the origin. The map ι_r transforms a point $\vec{x}$ by scaling it inversely along its radial direction. Since the scaling factor $r^2/\|\vec{x}\|^2$ is a positive real, the direction (and hence the phase $\langle\vec{x}\rangle$) is preserved exactly while only the radial magnitude is swapped [22, 23].

T_r exhibits *Escher Tri-Quarter Reflective Duality*[1] between $X_{-,r}$ and $X_{+,r}$ if there exists a map ι_r that satisfies the following conditions [12]:

(1) $\iota_r(\vec{x}) = \vec{x}$ for all $\vec{x} \in T_r$, which implies that T_r is fixed under inversion.

(2) $\iota_r(X_{-,r}) = X_{+,r}$ and $\iota_r(X_{+,r}) = X_{-,r}$, which implies that $X_{-,r}$ and $X_{+,r}$ are swapped.

(3) $\phi(\iota_r(\vec{x})) = \phi(\vec{x})$ for all $\vec{x} \in X$, which implies that the phase pairs are preserved to maintain directional consistency.

(4) ι_r is an involution (i.e., $\iota_r^{-1} = \iota_r$), which implies that the map is its own inverse for reversible transformations.

[1]Named after M.C. Escher's art that features reflective symmetries and infinite tilings that visually inspire the mathematical inversion and zone swapping duality here [24].

The circle inversion map ι_r satisfies these conditions to exemplify reflective duality across T_r between $X_{-,r}$ and $X_{+,r}$. Specifically:

(1) For $\vec{x} \in T_r$, $||\vec{x}|| = r$, so $\iota_r(\vec{x}) = \frac{r^2\vec{x}}{r^2} = \vec{x}$, which fixes T_r as required in condition 1.

(2) For $\vec{x} \in X_{-,r}$ where $||\vec{x}|| < r$, compute $||\iota_r(\vec{x})|| = \left|\left| \frac{r^2\vec{x}}{||\vec{x}||^2} \right|\right| = \frac{r^2}{||\vec{x}||} > r$, which maps to $X_{+,r}$; symmetrically similar for the reverse, which satisfies condition 2.

(3) The phase preservation follows from the real scalar $\frac{r^2}{||\vec{x}||^2} > 0$, which scales the magnitude but preserves the direction: $\langle \iota_r(\vec{x}) \rangle = \langle \vec{x} \rangle$, and thus the component signs remain unchanged to ensure $\phi(\iota_r(\vec{x})) = \phi(\vec{x})$ as in condition 3.

(4) The involution is derived algebraically: $\iota_r(\iota_r(\vec{x})) = \iota_r\left(\frac{r^2\vec{x}}{||\vec{x}||^2}\right) = \frac{r^2 \cdot \frac{r^2\vec{x}}{||\vec{x}||^2}}{\left|\left|\frac{r^2\vec{x}}{||\vec{x}||^2}\right|\right|^2} = \frac{r^4\vec{x}/||\vec{x}||^2}{r^4/||\vec{x}||^2} = \vec{x}$, which confirms $\iota_r^{-1} = \iota_r$ for condition 4.

For example, inverting the point $\vec{x} = (0.5, 0)_C$ (with $||\vec{x}|| = 0.5 < r = 1$) yields the corresponding "twin" point $\iota_1(\vec{x}) = (2, 0)_C$—with ($||\iota_1(\vec{x})|| = 2 > r = 1$), which both have the same phase pair $(0, 0)_\phi$. Additionally, for a radial ray at phase $\pi/3$, points like $k \cdot (0.5, \sqrt{3}/2)_C$ for $k > 0$ all have the same phase pair $(0, \pi/2)_\phi$; the reversible inversion maps back-and-forth between a k-valued norm and $1/k$-valued norm. These examples illustrate how inversion maintains directional consistency to set the stage for discrete extensions in Section 3.

Indeed, this Escher-inspired reflective duality is a key component of TQF because it provides an exact bijection between $X_{-,r}$ and $X_{+,r}$ to leverage the radial symmetry of T_r. The preservation of phase pairs implies that the structured orientation's directional information remains consistent under inversion, which is critical for real-world applications that require stable orientations, such as graph traversals or pattern recognition. Compared to other geometric transformations (e.g., translations or rotations), circle inversion uniquely swaps radial distances while preserving angular structure, which makes it ideal for symmetric computational models [16, 23, 25].

These continuous elements, equipped with symmetry and bijection, extend naturally to discrete radial dual triangular lattice graphs in Section 3 to enable valuable computational applications.

3 The Radial Dual Triangular Lattice Graph

The continuous TQF [12], as we recapitulated in Section 2, provides a unified coordinate system, structured orientation phase pair assignments, and topological zones that natively support duality and reflection through exact bijective mappings. To ultimately extend and apply TQF to real-world computational contexts—such as networks [1], distributed systems [2], parallel algorithms [3], tiling [17], robotic path planning [26], multi-agent coordination [27], lattice-based cryptography [28], image processing [29], signal processing [30], and others—we must first discretize the framework to encode complex states and transitions for computational implementations.

In this pursuit, we discretize the TQF onto a *base triangular lattice* L and then construct the *radial dual triangular lattice graph* Λ_r. This section begins with related work, followed by the definition, motivation, and structure of L, which includes an Eisenstein integer basis that generates the encoded triangular points of L in $X = \mathbb{C} \setminus \{(0, 0)_C\}$ and a hexagonally symmetric angular sector partition—to focus solely on geometric properties without graph connectivity. We then advance to the definition, motivation, and structure of Λ_r with vertices (and edges) that adhere to a TQF bijective zone partition to enable exact mappings.

3.1 Related Work

Our TQF draws upon foundational techniques in graph theory and computational geometry, such as quadtrees for hierarchical spatial partitioning and Voronoi diagrams for proximity-based tessellations, while innovating with radial duality to better exploit symmetries in discrete structures. In this subsection, we briefly survey these and other key related discretization methods—along with their extensions like discrete conformal mappings and self-dual plane graphs—and articulate how TQF overcomes their shortcomings in

handling radial, symmetry-aware computations, such as costly polar conversions or the absence of built-in bijective zone swaps.

- **Spatial Structure Partitioning:** Traditional tools like quadtrees [4] recursively divide space into nested boxes that are aligned with Cartesian axes to enable efficient point queries in $O(\log n)$ time. However, they often misalign with radial patterns (e.g., data emanating from a central origin), and thus require costly conversions for polar queries and potentially degrading to $O(\sqrt{|V|})$ in unbalanced radial cases [31]. In contrast, TQF's radial zones and phase pairs provide native support for origin-centered efficiency, with $O(1)$ directional binning (via sign-based phase assignments or modular angular sector indexing) and balanced parallel traversals at approximately $O(|V|/6 + C)$ per angular sector—where C is synchronization overhead cost—to yield practical ∼2-4x speedups in symmetry-aware algorithms like breadth-first search (BFS), as benchmarked on similar lattices [32]. Similarly, Voronoi diagrams [5] excel at proximity-based territorial partitioning around multiple points, but they lack built-in radial duality from a single origin. Extensions like polar Voronoi [20] use angular sweeps for diagrams, whereas TQF utilizes circle inversion for exact zone swaps to reduce redundancy in radial shortest-path queries [33–35]. This complements the multi-site Voronoi approach in applications like robotics path planning [36], where single-hub symmetry accelerates navigation.

- **Discrete Conformal and Lattice Mappings:** Existing discrete conformal mappings on triangular lattices—such as those using circle patterns [6, 37]—preserve angles for simulations by "stretching" the grid like a rubber sheet. Extensions include conformally equivalent lattices [38] and discrete complex analysis [39–41], often via circle packings [42]. Within the circle packing tradition, the working substrate is circle inversion (and Möbius geometry more generally) acting on triangulated and hexagonal lattices. The tradition originates with Thurston's circle packings on triangulated surfaces [43], develops through the variational theory of circle patterns [44], and includes the inversive distance generalization of Bowers and Stephenson [42, 45]. The individual ingredients of inversion operating on a triangular lattice are thus well established in that body of work. TQF differs from this tradition not in introducing inversion to the lattice environment, but in *how* it uses inversion. Rather than deforming a metric conformally or solving for radii, TQF fixes a single origin-centered admissible circle and uses one circle inversion to pair the outer and inner zones into a single lattice graph by transport. This achieves radial duality with bijective phase constancy over carefully crafted isomorphic subgraphs that are ideal for combinatorial paths—an origin-centered zone swap that purely conformal or radius-solving approaches do not target. Our choice of Eisenstein-based triangular lattice graph optimizes this: its degree-6 vertices enable denser radial rays and order-6 symmetry to outperform square grids (mismatched angles) or hexagonal grids (sparser connections) [17], as detailed in Table 4.

- **Graph Dualities and Self-Dualities:** Self-dual plane graphs [46] are isomorphic to their planar duals via vertex-face swaps in embeddings, which differ from our TQF's embedding-independent radial duality due to its norm trichotomy. Whitney's combinatorial duality [21] maps cycles to cuts for planarity, but it lacks our mod 6 periodicity for angular partitioning because it is designed for general planar graphs without exploiting rotational symmetries. In contrast, TQF leverages the triangular lattice's inherent order-6 symmetry (from 60-degree angles) to enable equitable angular sector decompositions via mod 6 arithmetic, where the dualities (combinatorial for separation and reflective for swaps) are directly tied to—and preserved by—these symmetries for balanced parallel traversals. TQF's $O(|V_{T,r}|)$ boundary separator enables efficient max-flow via Ford-Fulkerson [47] on symmetric cuts to reduce synchronization to $O(6)$ for attacking parallelizable problems and supporting optimization [18] or multi-agent coordination [48]. Quantitatively, symmetries permit $O(1)$ per-vertex assignments (e.g., phase pairs via component signs or angular sector binning via floor-modular arithmetic) and parallel traversals at $O(|V|/6 + C)$ to facilitate applications in tiling [17] and beyond. As demonstrated in our symmetry-reduced clustering benchmarks (Section 6.2), this yields up to ∼6x speedups in motif analysis by computing on orbit representatives and replicating via rotations.

- **Exact-Arithmetic Geometric Computation:** The standard answer to numerical robustness in computational geometry is the *exact-predicate* approach—wrapping floating-point operations in adaptive precision filters or lazy exact evaluators—epitomized by Shewchuk's adaptive predicates [8] and CGAL's exact computation paradigm [9]. Both deliver provably-correct sign-of-determinant and orientation tests, but they do so by treating exactness as a runtime correctness patch layered on top of an inherently floating-point design, which costs per-predicate filter evaluation overhead and a non-trivial fallback path on every query. TQF takes a structurally different route: by anchoring the framework to the Eisenstein integers $\mathbb{Z}[\omega]$, the precision-critical decisions—admissibility (Definition 3.14), zone membership (Definitions 3.16–3.18), and boundary tests—all reduce to comparisons of *integer-valued* norms $N(a + b\omega) = a^2 - ab + b^2$ against r^2 (with $r^2 \in \mathbb{Z}_{>0}$ by Definition 3.14). In other words, instead of paying for exactness at every query, we get it for free as a property of the lattice itself: the comparisons are $O(1)$ per vertex in native integer arithmetic, with no floating-point fallback to fall back to.
- **Equivariant Labelings and Symmetry-Aware Encodings:** Group-equivariant representations on graphs—spanning group-equivariant convolutional neural networks [10], the broader geometric deep learning program [11], and equivariant graph signal processing [30]—have made a strong empirical case that labelings which commute with a group action yield substantial sample- and compute-efficiency gains. The catch is that these approaches typically *learn* approximately-equivariant features over a generic graph substrate that itself carries no inherent group structure, so the equivariance is an emergent property of the model rather than a property of the data. TQF inverts this design: the substrate Λ_r comes equipped with a faithful $\mathbb{T}_{24}$-action by construction (Definition 4.16), and the equivariant labelings developed in Section 5—the angular sector-based six-encoding s_6 (Definition 5.3), the trihexagonal six-coloring e_6 (Definition 5.10), and the $\mathbb{T}_{24}$-equivariant $\mathbb{Z}_2$-chain complex on vertices, edges, and faces (Definitions 5.22–5.23)—are constructed to commute exactly with $\mathbb{T}_{24}$, with closed-form proofs of equivariance (Lemma 3.40, Theorems 5.4 and 5.15, Proposition 5.25). The practical consequence is that orbit-transversal computation—where you compute on a single representative per orbit and replicate to the rest of the orbit via rotations—becomes a structural consequence of how the labeling is defined rather than a learned approximation that has to be retrained per task, which is what gives the ∼6x ceiling in our symmetry-reduced clustering benchmarks its theoretical grounding.

This discretization extends the continuous TQF [12]—without approximations—to bridge gaps in radial symmetry for scalable computations. To address these gaps, the TQF introduces a novel radial duality that embeds origin-centered symmetries and exact bijections by design, as formalized below.

Definition 3.1 (Radial Duality). A *radial duality* on a discrete structure (e.g., a lattice graph on the punctured complex plane $X = \mathbb{C} \setminus \{(0,0)_C\}$) is a pair of operations—a norm-based trichotomy partition into the inner zone (elements with Euclidean norms $< r$), the boundary zone (elements with norms $= r$), and the outer zone (elements with norms $> r$) for some radius $r > 0$ that ensures symmetric boundaries, and a phase-preserving circle inversion map ι_r (recall Subsection 2.4 and see upcoming Definition 4.12)—that induce an exact bijection between the inner zone and outer zone while satisfying:

(1) a fixed boundary zone,

(2) the inner direction and outer direction with respect to the boundary zone, and

(3) the phase pair assignments of Tables 1–2.

Remark 3.2. *This duality exploits origin-centered radial symmetries (e.g., order-6 rotational invariance of $\mathbb{Z}_6$ under D_6) to enable reversible, embedding-independent transformations without approximations.*

This radial duality provides the geometric foundation for discretizing the continuous TQF onto symmetric lattice structures, beginning with the base triangular lattice in the next subsection.

3.2 The Base Triangular Lattice L

To discretize the continuous domain of the TQF onto a lattice graph, we first define the base triangular lattice L and introduce lattice-specific notions of rays and angular sectors that align with the radial symmetry and phase constancy that are central to the framework. These formalizations expedite a seamless transition from continuous rays in X to discrete rays on L, which then facilitate exact bijections and symmetry exploitation in computational applications—paving the way for the full radial dual graph construction of Λ_r in the next subsection.

The choice of basis vectors $\vec{\omega}_0$ and $\vec{\omega}_1$ deliberately echoes the *Eisenstein integers* $\mathbb{Z}[\omega] = \{a + b\omega : a, b \in \mathbb{Z}\}$ with $\omega = e^{2\pi i/3}$, the ring introduced by Gotthold Eisenstein in 1844 in his proof of the law of cubic reciprocity [13]. Eisenstein integers occupy a privileged place in number theory: their integer-valued norm $N(a+b\omega) = a^2-ab+b^2$, their unit group $\{\pm 1, \pm\omega, \pm\omega^2\}$ of exactly six sixth-roots of unity acting by rotation, and their identification of "primary" representations via quadratic forms together provide the precise algebraic machinery—exact arithmetic, order-6 symmetry, and well-understood representability—that underwrites the discrete dualities and admissibility conditions developed in this section.

Definition 3.3 (Base Triangular Lattice). The *base triangular lattice* L on the origin-punctured complex plane $X = \mathbb{C} \setminus \{(0,0)_C\}$ is the set of points generated by basis vectors $\vec{\omega}_0 = 1$ and $\vec{\omega}_1 = e^{i\pi/3}$, which consist of $\vec{x} = a\vec{\omega}_0 + b\vec{\omega}_1$ for $(a, b) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$.

Remark 3.4. *Writing $\omega = e^{2\pi i/3}$ for the standard Eisenstein primitive cube root of unity, our choice $\vec{\omega}_1 = e^{i\pi/3} = 1 + \omega$ corresponds to the basis $(1, 1 + \omega)$ of the same lattice $\mathbb{Z}[\omega]$ generated by the canonical Eisenstein basis $(1, \omega)$. Equivalently, $\|a\vec{\omega}_0+b\vec{\omega}_1\|^2 = a^2+ab+b^2$ in our basis and $\|a'+b'\omega\|^2 = a'^2-a'b'+b'^2$ in the canonical basis represent the same set of admissible integers as (a, b) and (a', b') range over $\mathbb{Z}^2$. We retain the sixth-root convention throughout because it yields primitive direction vectors with $\{0, \pm 1\}$ coefficients (Definition 3.39).*

Definition 3.5 (Lattice Ray). A *lattice ray* in the base triangular lattice L is the set of lattice points that lie on a half-line in the plane (i.e., the intersection of a half-line with L), defined as $\{\vec{p}_0 + k\vec{p}_1 \mid k \in \mathbb{N}_0\}$, where $\mathbb{N}_0 = \{0, 1, 2, \dots\}$, $\vec{p}_0 \in L$ is the starting point, and $\vec{p}_1 \in L$ is a primitive direction vector (i.e., $\vec{p}_1 = a\vec{\omega}_0 + b\vec{\omega}_1$ for $a, b \in \mathbb{Z}$ with $\gcd(a, b) = 1$). Equivalently, for a fixed direction from $\vec{p}_0$, it consists of all points in L that are collinear with $\vec{p}_0$ in that direction, ordered by increasing distance from $\vec{p}_0$.

Definition 3.6 (Lattice Radial Ray). A *lattice radial ray* in the base triangular lattice L is a lattice ray defined as the set $\{k\vec{p} \mid k \in \mathbb{N}\}$, where $\mathbb{N} = \{1, 2, 3, \dots\}$ (excluding the origin), and $\vec{p} \in L$ is a primitive lattice vector (i.e., $\vec{p} = a\vec{\omega}_0 + b\vec{\omega}_1$ for $a, b \in \mathbb{Z}$ with $\gcd(a, b) = 1$). Equivalently, for a fixed phase θ such that there exists $\vec{p} \in L$ with $\langle\vec{p}\rangle = \theta$ (which yields an infinite number of such rays that densely fill the possible directions), the ray consists of all points in L with that phase, ordered by increasing norm from the origin.

Definition 3.7 (Primary Ray). A *primary ray* in the base triangular lattice L is one of the six lattice radial rays at phases $t\pi/3$ for $t \in \mathbb{Z}_6$—which emanate from (but do not include) the origin and are aligned with the symmetry axes of D_6—and which bound the angular sectors S_t.

Remark 3.8. *L exhibits the dihedral group D_6 as its point symmetry group, which consists of six rotations $\mathcal{R}_{t\pi/3}$ by multiples of $\pi/3$ around the origin (for $t \in \mathbb{Z}_6$) and six reflections across axes aligned with the primary ray directions (e.g., the real axis and rays at $\pi/3, 2\pi/3$, etc.). This order-12 D_6 group underpins L's equilateral geometry, empowering modular decompositions and efficient computations via symmetry exploitation. While the continuous framework in Section 2 leverages $\mathcal{R}_{\pi/2}$ quadrant rotations to preserve sign categories and cycle phase pairs, here we additionally exploit $\mathcal{R}_{\pi/3}$ rotations for lattice symmetry to align with the denser angular structure of L's triangular grid.*

Our choice of L is driven by its natural alignment with the $\pi/3$-based angular structure, which serves the order-6 rotational symmetry to facilitate exact, efficient encodings without the approximation errors that are common in other grids, such as square or hexagonal lattices [17, 18, 49–51]. Unlike square lattices, which lack order-6 symmetry and impose orthogonal constraints, or hexagonal lattices (referring here to the honeycomb lattice with degree 3, as the dual of the triangular lattice [52]), which have sparser point arrangements, L's equilateral geometry and symmetry align with the TQF's directional and radial structures to ultimately enable precise computations and scalable algorithms.

To quantitatively compare lattice choices, we summarize key properties in Table 4. These comparisons exemplify why L provides a key fundamental balance: its point arrangement supports richer radial structures while maintaining exact symmetries, unlike hexagonal (with sparser lattice radial rays) or square (with mismatched angles) lattices [17]. For example, in (a non-truncated) L, each point has six neighbors at $\pi/3$ intervals, thereby supporting denser lattice radial rays (e.g., six primary lattice radial rays per full rotation) when compared to a hexagonal lattice's three neighbors at $2\pi/3$ (halving the lattice radial ray options and sparsifying lattice radial rays) or a square lattice's four neighbors at $\pi/2$ (due to misalignment with $\pi/3$ angles).

TABLE 4. Comparison of Lattice Types for the TQF

Lattice Type	**Symmetry Order**	**Neighbor Count**	**$\pi/3$ Phase Alignment**	**Inversion Bijectivity Preserving Framework Symmetries?**
Triangular	12 (full D_6)	6	Exact	Yes
Square	8 (full D_4)	4	$\approx \pi/2$	No
Hexagonal	12 (full D_6)	3	Exact	Yes, but with fewer rays and connections

Note. Lattice comparisons highlight triangular optimality for bijectivity under inversion, unlike square (mismatched angles) or hexagonal (sparser rays) grids [17].

L forms a regular triangulation of X and has a dual hexagonal (honeycomb) structure [52]. Theoretically, L is infinite and it ensures exact preservation of the continuous TQF's properties: the angular point distributions are discrete multiples of $\pi/3$, which align perfectly with the phase assignment rules of Tables 1–2 for precise directional categorization (e.g., see upcoming Corollary 3.11). Norms are computed exactly because squared distances are integers (e.g., $||\vec{x}||^2 = a^2 + ab + b^2$ for $\vec{x} = a\vec{\omega}_0 + b\vec{\omega}_1$), which enables unambiguous comparisons without floating-point error. A key feature of L is its order-6 rotational symmetry [17], where the six primary rays (Definition 3.7) align precisely with the six unique $\pi/3$ angular increments at phases $t\pi/3$ for $t \in \mathbb{Z}_6$ and are each assigned one of the six unique phase pairs from Table 5 (e.g., the ray at phase 0 is assigned $(0,0)_\phi$). This symmetry enhances computational efficiency—allowing constant-time operations under rotations and reflections—and geometrically characterizes the framework's directional classifications. For instance, it enables the formation of regular hexagonal boundaries, which transforms abstract radial zones into computable separators consistent with L's equilateral geometry—suitable for applications like partitioning or routing [53].

The phase pair assignments of Subsection 2.2 apply directly to L, where the phase pair assigned to a given lattice radial ray remains constant due to sign preservation, as in the continuous case. This constancy, paired with the rotational symmetry, sets up our formal partitioning into angular sectors.

Definition 3.9 (Angular Sectors). For $t \in \mathbb{Z}_6$ an *angular sector* S_t in the origin-punctured complex plane X is the set of points whose phase $\langle\vec{x}\rangle$ satisfies $t\pi/3 \leq \langle\vec{x}\rangle < (t+1)\pi/3 \pmod{2\pi}$, where each angular sector spans $\pi/3$ radians and is bounded by primary rays on $L \in X$ at phases $t\pi/3$ that emanate from (but do not include) the origin. Equivalently, for a fixed $t \in \mathbb{Z}_6$, S_t consists of all points in X with phases in that interval, which includes the minimum-phase boundary ray via floor-based indexing $t = \lfloor 6\langle\vec{x}\rangle/(2\pi)\rfloor \bmod 6$ to form a non-overlapping partition of X (and thus $L \subset X$).

These angular sectors formalize L's rotational symmetry as follows.

Lemma 3.10 (Angular Sector Partitioning). Each angular sector S_t (for $t \in \mathbb{Z}_6$) aligns with the base triangular lattice L's order-6 symmetry with the phase pair constancy along the lattice radial rays (including the boundaries) and modular indexing $t = \lfloor 6\langle\vec{x}\rangle/(2\pi)\rfloor \bmod 6$. Equivalently, the angular sectors partition X (and thus $L \subset X$) into six $\pi/3$-radian wedges that are bounded by the primary rays at phases $t\pi/3$, which secures consistent directional labeling and rotational invariance under D_6.

Proof. To verify the properties of the angular sectors S_t for $t \in \mathbb{Z}_6$, we proceed in steps to confirm the partition, lattice alignment, phase constancy, and rotational invariance.

- **Step 1: Verify the Non-Overlapping Exhaustive Partition of X and $L \subset X$.** The angular sectors S_t consist of points with phase $\theta \bmod 2\pi \in [t\pi/3, (t+1)\pi/3)$ and are bounded by primary rays at phases $t\pi/3$. The floor-based indexing $t = \lfloor 6\langle\vec{x}\rangle/(2\pi)\rfloor \bmod 6$ includes the lower boundary in S_t (e.g., at exactly $t\pi/3$, it yields t) while excluding the upper boundary (assigned to $S_{t+1 \bmod 6}$). This forms a non-overlapping, exhaustive partition of the origin-punctured plane X into six $\pi/3$-radian wedges, and thus of the base triangular lattice $L \subset X$. ☑
- **Step 2: Confirm Alignment with the Order-6 Rotational Symmetry of D_6.** By Definition 3.3, L has basis vectors $\vec{\omega}_0 = 1$ and $\vec{\omega}_1 = e^{i\pi/3}$ that are separated by $\pi/3$, which matches the angular sector spacing. Since $\mathcal{R}_{\pi/3}(\vec{\omega}_0) = \vec{\omega}_1$ and $\mathcal{R}_{\pi/3}(\vec{\omega}_1) = \vec{\omega}_1 - \vec{\omega}_0$ are both elements of L, we have $\mathcal{R}_{\pi/3}(L) = L$. This aligns L with the order-6 rotational symmetry of D_6 because rotations $\mathcal{R}_{k\pi/3}$ (for $k \in \mathbb{Z}_6$) cycle the angular sectors $S_t \mapsto S_{t+k \bmod 6}$. ☑
- **Step 3: Establish Phase Pair Constancy Along Lattice Radial Rays (Including Angular Sector Boundary Primary Rays).** For $\vec{x} = a\vec{\omega}_0 + b\vec{\omega}_1 \in L$, the phase $\langle\vec{x}\rangle = \arg(a + be^{i\pi/3})$ is fixed along a lattice radial ray, which consists of positive integer multiples of a primitive vector. The phase pair $\phi(\vec{x}) = (\langle\vec{x}_{\mathbb{R}}\rangle, \langle\vec{x}_{\mathbb{I}}\rangle)_\phi$ from Tables 1–2 depends solely on the signs of the real and imaginary components, which remain constant due to the fixed direction. This holds for boundary primary rays as well, with unique assignments per those tables. ☑
- **Step 4: Verify Modular Indexing and Rotational Invariance with Phase Pair Preservation.** The index $t = \lfloor 6\langle\vec{x}\rangle/(2\pi)\rfloor \bmod 6$ assigns $\vec{x}$ to S_t, including boundaries. Rotations $\mathcal{R}_{\pi/3}$ (when elements of D_6) map S_t to $S_{t+1 \bmod 6}$, permuting phase pairs equivariantly under the induced cyclic action on quadrant/axis labels (Subsection 2.2; see also the explicit $\mathcal{R}_{\pi/2}$ cycle in the discussion following Equation 3), rather than fixing them pointwise. The full D_6 (including reflections across primary rays) likewise acts equivariantly on the lattice points and phase pair assignments via fixed permutations of the directional labels. For example, $\vec{x} = \vec{\omega}_0 + \vec{\omega}_1$ has $\langle\vec{x}\rangle = \pi/6$ to yield $t = 0$ (in S_0) and phase pair $(0, \pi/2)_\phi$, which remain constant along its ray. ☑
- **Step 5: Conclude the Overall Partition Properties.** Equivalently, this partitions X (and L) into six disjoint, exhaustive $\pi/3$-radian wedges that are bounded by primary rays (as in the remark following Definition 3.6), where each inherits consistent phase pair labeling and D_6-invariance. ☑

Thus, the angular sectors S_t partition L with order-6 symmetry, phase pairs remain constant along lattice radial rays, and modular indexing is consistent. □

Corollary 3.11 (Unique Phase Pairs for Primary Rays). The six primary rays—which bound the angular sectors S_t for $t \in \mathbb{Z}_6$—each receive a unique phase pair as listed in Table 5. The phase pair assignment rules of Tables 1–2 ensure their uniqueness, while Lemma 3.10 guarantees constancy along each primary ray. Thus, these six distinct phase pairs align with the order-6 rotational symmetry of L.

Remark 3.12. *The six angular sector boundaries refer to primary rays in L (and the subsequently defined graph rays in Λ_r). These primary rays consist of all points in L that are collinear with the origin at phases $t\pi/3$ (for $t \in \mathbb{Z}_6$), ordered by increasing norm—a property that reinforces their role in bounding the angular sectors S_t.*

TABLE 5. Unique Phase Pairs for Angular Sector Boundary Primary Rays

Angle (radians)	Angular Sector	Phase Pair	Description
0	S_0	$(0,0)_\phi$	East Axis
$\pi/3$	S_1	$(0,\pi/2)_\phi$	Quadrant I Boundary
$2\pi/3$	S_2	$(\pi,\pi/2)_\phi$	Quadrant II Boundary
π	S_3	$(\pi,0)_\phi$	West Axis
$4\pi/3$	S_4	$(\pi,3\pi/2)_\phi$	Quadrant III Boundary
$5\pi/3$	S_5	$(0,3\pi/2)_\phi$	Quadrant IV Boundary

This mod 6 partitioning is intuitive, similar to clock arithmetic where the hours wrap around every 12, but here instead of 12 circular positions, we have six circular positions for rotational symmetry. This characterizes an efficient angular sector traversal (or cycling)—like turning a clock hand by $\mathcal{R}_{\pi/3}$ increments.

Example 3.13 (Phase Pair Assignments for Sample Points on L). To further exemplify the structure of L, let's consider some small examples with $r = 1$:

- The point $\vec{x}_0 = \vec{\omega}_0 = 1 \in L$ has coordinate $(\vec{x}_{0,\mathbb{R}}, \vec{x}_{0,\mathbb{I}})_C = (1,0)_C$, norm $||\vec{x}_0|| = 1$, phase $\langle \vec{x}_0 \rangle = 0$, assigned phase pair $\phi(\vec{x}_0) = (0,0)_\phi$ (from Table 2), and exists in the boundary zone T_1, angular sector S_0, and the positive real axis.
- The point $\vec{x}_1 = \vec{\omega}_0 + \vec{\omega}_1 = 1 + e^{i\pi/3} \in L$, has exact coordinate $(x_{1,\mathbb{R}}, x_{1,\mathbb{I}})_C = (3/2, \sqrt{3}/2)_C \approx (1.5, 0.866)_C$, norm $||\vec{x}_1|| = \sqrt{3} > 1$, phase $\langle \vec{x}_1 \rangle = \pi/6$, assigned phase pair $\phi(\vec{x}_1) = (0,\pi/2)_\phi$ (from Table 1), and exists in the outer zone $X_{+,1}$, angular sector S_0, and Quadrant I.
- The point $\vec{x}_2 = -\vec{\omega}_0 = -1 \in L$ has coordinate $(x_{2,\mathbb{R}}, x_{2,\mathbb{I}})_C = (-1,0)_C$, norm $||\vec{x}_2|| = 1$, phase $\langle \vec{x}_2 \rangle = \pi$, assigned phase pair $\phi(\vec{x}_2) = (\pi,0)_\phi$, and exists in the boundary zone T_1, angular sector S_3, and the negative real axis.
- The point $\vec{x}_3 = \vec{\omega}_1 = e^{i\pi/3} \in L$ has coordinate $(x_{3,\mathbb{R}}, x_{3,\mathbb{I}})_C = (0.5, \sqrt{3}/2)_C$, norm $||\vec{x}_3|| = 1$, phase $\langle \vec{x}_3 \rangle = \pi/3$, assigned phase pair $(0,\pi/2)_\phi$, and exists in the boundary zone T_1, angular sector S_1, and Quadrant I.
- Given the previous point $\vec{x}_3 = \vec{\omega}_1 = e^{i\pi/3} \in L \cap T_1$, we further extend to the point $\vec{x}_4 = 2\vec{x}_3 = 2\vec{\omega}_1 \in L$ in the outer zone $X_{+,1}$ to observe that the phase $\langle \vec{x}_4 \rangle = \pi/3$, assigned phase pair $(0,\pi/2)_\phi$, angular sector S_1, and Quadrant I are all preserved.

These examples show how the framework efficiently categorizes points across zones and angular sectors, with exact alignments due to L's symmetry, which includes off-axis directions.

With L and its angular sectors established, along with its symmetry and ray properties, we now advance to construct the radial dual triangular lattice graph Λ_r equipped with dualities that are fundamental to the framework.

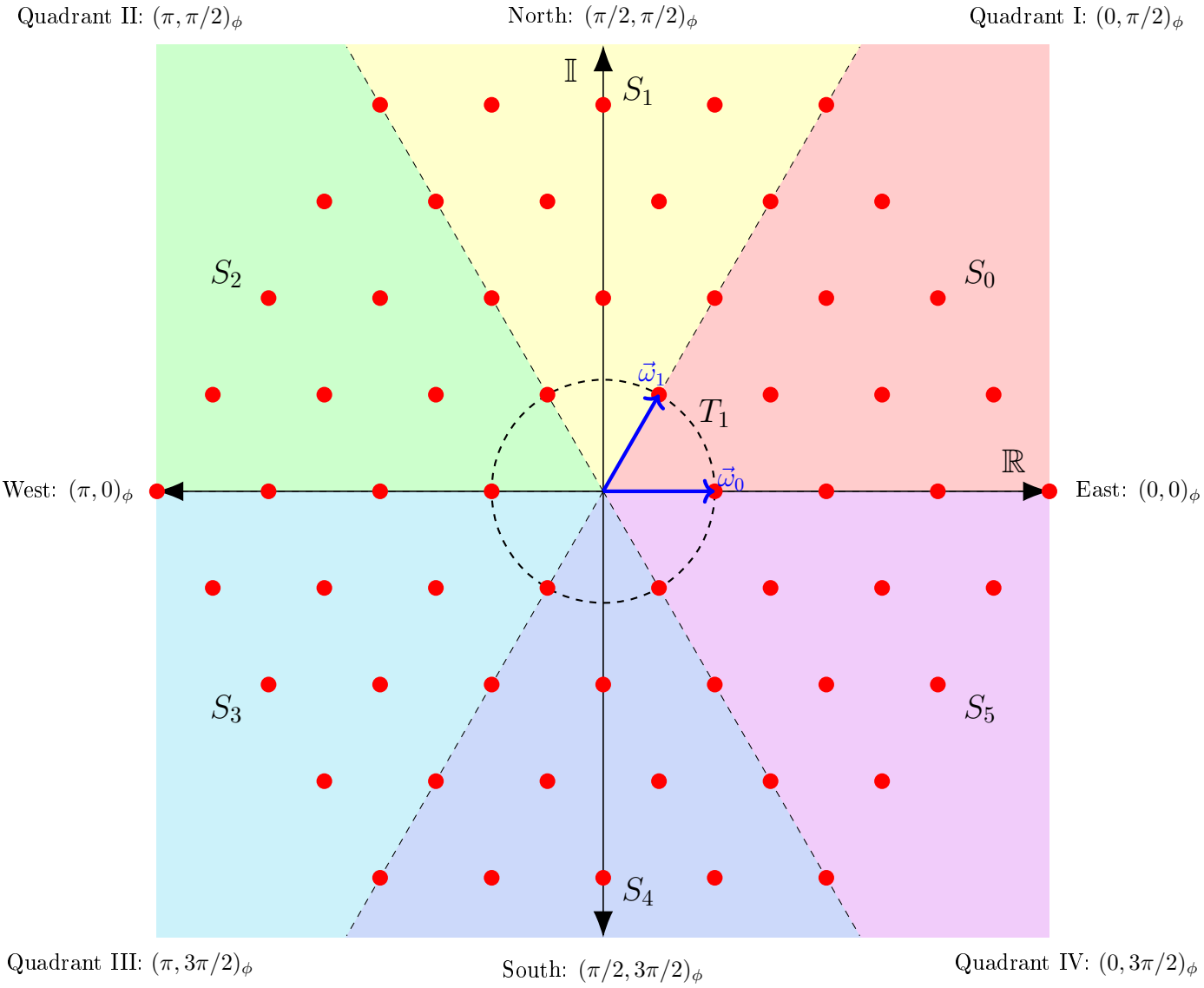

FIGURE 2. The structure of the base triangular lattice L with basis vectors $\vec{\omega}_0$ and $\vec{\omega}_1$, colored angular sectors S_t, and dashed boundary primary rays that illustrate order-6 rotational symmetry and mod 6 partitioning superimposed on the quadrants and axes with phase pair assignments. L is truncated symmetrically within the truncation radius $R = 4$ for balanced visualization. Note: The boundary zone T_1 aligns with the angular sector boundaries.

3.3 Constructing Λ_r

Building on the base triangular lattice L and its symmetries, we now construct the radial dual triangular lattice graph Λ_r to enable exact bijective zone partitioning and duality mappings. To achieve this, we first define admissible inversion radii for symmetric boundaries, then specify the vertex sets for the inner, boundary, and outer zones, and finally induce edges that reinforce topological structure across zones.

Definition 3.14 (Admissible Inversion Radius). An inversion radius $r > 0$ is *admissible* if $r^2 = N$ is a positive integer representable as $N = a^2 + ab + b^2$ for $(a, b) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$. For such r, the boundary set is $V_{T,r} = L \cap T_r = \{\vec{v} \in L : \|\vec{v}\| = r\}$—which contains exactly $6k$ points (or vertices, as formalized in the upcoming Definition 3.17) for some $k \in \mathbb{N}$—which forms a symmetric boundary set under the order-6 rotational symmetry of the triangular lattice L (due to the action of the Eisenstein unit group), as per the theory of quadratic forms [54].

Remark 3.15. *See Table 6 for small example values of r that satisfy the admissible conditions of Definition 3.14. For some N (e.g., when $N = 12$) the boundary vertices may not lie exactly on primary rays (though they still form a symmetric set with a uniform distribution across the angular sectors due to rotational symmetry, which preserves bijectivity for duality). Note that $|V_{T,r}| = 6k$ does* not *imply that $V_{T,r}$ is a single $\mathbb{Z}_6$-orbit under the order-6 rotation group: for $N = r^2$ admitting multiple inequivalent quadratic-form representations (for example, $N = 7$ yields twelve points forming two $\mathbb{Z}_6$-orbits of six, listed explicitly in Table 9), the boundary set decomposes into several $\mathbb{Z}_6$-orbits whose union still forms a symmetric set under D_6. The rotation orbit size is exactly six because the Eisenstein unit group $\{\pm 1, \pm\omega, \pm\omega^2\} \cong \mathbb{Z}_6$ acts freely on nonzero lattice points: a nontrivial unit fixing $\vec{v} \neq 0$ would imply $\vec{v} = 0$ by linearity, so every rotation orbit consists of exactly six distinct points. The full D_6 (rotations together with reflections) acts more coarsely: an on-axis representation gives a single D_6-orbit of six (each such vertex lies on a reflection axis, as in the minimal case $N = 1$), whereas an off-axis representation such as $N = 7$ gives a single D_6-orbit of twelve, since reflection fuses its two rotation orbits. In all cases, the dualities, bijections, and equivariance results in this paper depend only on the D_6-invariance of the union $V_{T,r}$ as a whole, not on its orbit decomposition.*

TABLE 6. Examples of Admissible Inversion Radii

r	$N = r^2$	k	$\lvert V_{T,r}\rvert = 6k$	**Single Cycle?**	**Orbits under $\mathbb{Z}_6$**
1	1	1	6	Yes	1 orbit of 6
$\sqrt{3}$	3	1	6	Yes	1 orbit of 6
2	4	1	6	Yes	1 orbit of 6
$\sqrt{7}$	7	2	12	No	2 orbits of 6
3	9	1	6	Yes	1 orbit of 6
$\sqrt{12} = 2\sqrt{3}$	12	1	6	Yes	1 orbit of 6
$\sqrt{13}$	13	2	12	No	2 orbits of 6
4	16	1	6	Yes	1 orbit of 6
$\sqrt{19}$	19	2	12	No	2 orbits of 6

Note. Orbits are counted under the order-6 rotation group $\mathbb{Z}_6$ (the Eisenstein unit group), which acts freely on nonzero lattice points so every rotation orbit has size six. Under the full D_6 (rotations and reflections) the on-axis single-orbit cases ($k = 1$) remain single D_6-orbits of six, whereas the off-axis cases $N = 7, 13, 19$ each form a single D_6-orbit of twelve (their two rotation orbits fuse under reflection).

Definition 3.16 (Outer Zone Vertex Set). For radius $r > 0$, the *outer zone vertex set* is $V_{+,r} := \{\vec{v}_i \in L : ||\vec{v}_i|| > r\}$, which consists of lattice points that are farther from the origin than r.

Definition 3.17 (Boundary Zone Vertex Set). For radius $r > 0$, the *boundary zone vertex set* is $V_{T,r} := L \cap T_r = \{\vec{v}_i \in L : ||\vec{v}_i|| = r\}$ (the lattice points lying exactly on the continuous boundary circle T_r), which consists of lattice points that are exactly at distance r from the origin.

Definition 3.18 (Inner Zone Vertex Set). For radius $r > 0$, the *inner zone vertex set* is $V_{-,r} := \iota_r(V_{+,r})$, which consists of the image of the outer zone vertex set $V_{+,r}$ under the circle inversion map ι_r (and thus are closer to the origin than r). Note that $V_{-,r}$ is generally *not* a subset of L (its elements have rational coordinates in $\mathbb{Q}(\sqrt{3})$ rather than integer coordinates in the lattice basis). The inner zone inherits its graph structure by transport along ι_r rather than by inheritance from L, as formalized in Remark 3.26.

Definition 3.19 (Total Vertex Set). For radius $r > 0$, the *total vertex set* is $V_r := V_{-,r} \sqcup V_{T,r} \sqcup V_{+,r}$, which consists of all vertices across the inner, boundary, and outer zones.

The vertex sets $V_{-,r}$, $V_{T,r}$, and $V_{+,r}$ form a partition of the total vertex set V_r, which is mediated by the norm trichotomy for any $\vec{v}_i \in V_r$:

$$\begin{aligned} ||\vec{v}_i|| < r &\iff \vec{v}_i \in V_{-,r} \\ ||\vec{v}_i|| = r &\iff \vec{v}_i \in V_{T,r} \\ ||\vec{v}_i|| > r &\iff \vec{v}_i \in V_{+,r}. \end{aligned} \tag{5}$$

Thus, we've established a discretized zone trichotomy of mutually disjoint vertex sets, which aligns with the continuous zone trichotomy (Equation 2). Indeed, we will soon see that V_r is key for the upcoming construction of Λ_r.

To illustrate the simplicity of zone assignment, let's consider the following pseudocode, which leverages integer squared norms for exact $O(1)$ comparisons per vertex without computing square roots or risking floating-point errors:

ALGORITHM PSEUDOCODE 1. Zone Assignment

```
function AssignZone(vertex_v, r_sq):
    norm_sq = ||vertex_v||^2  # compute squared norm
    if norm_sq < r_sq:
        return "inner"
    else if norm_sq == r_sq:
        return "boundary"
    else:
        return "outer"
```

Remark 3.20. *For admissible r (where $r^2 = N \in \mathbb{N}$ representable as $N = a^2 + ab + b^2$ for $(a, b) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$, as per Definition 3.14), the circle inversion ι_r maps each outer vertex $\vec{v}_i \in V_{+,r}$ to a point $\iota_r(\vec{v}_i) = r^2\vec{v}_i/\|\vec{v}_i\|^2$ with real part in $\mathbb{Q}$ and imaginary part in $\sqrt{3} \cdot \mathbb{Q}$ (i.e., coordinates in the real quadratic field $\mathbb{Q}(\sqrt{3})$), since the basis vectors $\vec{\omega}_0 = 1$ and $\vec{\omega}_1 = e^{i\pi/3}$ give every $\vec{v}_i \in L$ real part in $\mathbb{Q}$ and imaginary part in $\sqrt{3} \cdot \mathbb{Q}$, and $\|\vec{v}_i\|^2 \in \mathbb{N}$. Since ι_r is bijective on X, its restriction to $V_{+,r}$ is injective, which guarantees discreteness and no collisions in $V_{-,r}$. Note that the inverted images do* not *lie on L in general, but rather $V_{-,r} = \iota_r(V_{+,r})$ is a separate countable set of points whose graph structure—induced by transport from $V_{+,r}$ along ι_r—remains isomorphic to that of $V_{+,r}$ (as formalized in the subgraph context of Proposition 4.15) to secure exact bijective duality.*

We choose an admissible r such that the symmetric boundary set $V_{T,r}$ consists of $|V_{T,r}| = 6k$ vertices that are uniformly and symmetrically distributed across the angular sectors (Definition 3.9 and Lemma 3.10). For example, the minimal case (when $r = 1$ and $k = 1$) yields a unit hexagon with vertices positioned on the primary rays at phases 0, $\pi/3$, $2\pi/3$, π, $4\pi/3$, and $5\pi/3$. This configuration assures that each primary ray intersects exactly one boundary vertex in $V_{T,r}$, as visualized in Figure 4.

While we prioritize the minimal admissible inversion radius $r = 1$ (with $k = 1$) for simplicity, the framework extends to larger admissible inversion radii, where the boundary zone vertex set $V_{T,r}$ comprises multiples of six vertices ($|V_{T,r}| = 6k$ for $k \in \mathbb{N}$). The action of the Eisenstein unit group ensures that these vertices are uniformly and symmetrically distributed across the angular sectors S_t ($t \in \mathbb{Z}_6$), with exact alignment to the primary rays bounding the angular sectors for minimal $k = 1$. We note that exceptions occur for certain $N = r^2$ with larger k. For instance, $r = \sqrt{7}$ (with $N = 7$, $k = 2$) yields a denser set with $|V_{T,\sqrt{7}}| = 12$, where two lattice radial rays intersect the boundary per angular sector (see examples in Table 6).

Interestingly, even as the number of boundary-intersecting lattice radial rays per angular sector increase, the underlying periodicity of the mod 6 angular sectors and phase constancy persist. These key properties uphold the combinatorial and Escher reflective dualities in Section 4, which include bijective mappings via ι_r. Formal verification of these extensions, which encompass isomorphic and bijective properties under ι_r, appear in the upcoming Propositions 4.2 and 4.15.

Throughout the remainder of this paper, we assume that $r > 0$ is admissible. We now construct Λ_r:

Definition 3.21 (Radial Dual Triangular Lattice Graph). The *radial dual triangular lattice graph* $\Lambda_r = (V_r, E_r)$ is the graph with vertex set $V(\Lambda_r) = V_r = V_{-,r} \cup V_{T,r} \cup V_{+,r}$, where the edge set $E(\Lambda_r) = E_r$ is defined as follows: the subgraph induced on $V_{+,r} \cup V_{T,r}$ has edges between vertices at Euclidean distance 1 (the standard nearest-neighbor connections in the triangular lattice), the subgraph induced on $V_{-,r}$ has edges $\{\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)\}$ for each such edge $\{\vec{v}_i, \vec{v}_j\}$ in the subgraph induced on $V_{+,r}$, and for each edge $\{\vec{v}_i, \vec{v}_j\}$, where $\vec{v}_j \in V_{T,r}$ and $\vec{v}_i \in V_{+,r}$, the twin edge $\{\vec{v}_j, \iota_r(\vec{v}_i)\}$ is added. This implies that no direct edges exist between $V_{-,r}$ and $V_{+,r}$, so reflective twins must symmetrically connect across $V_{T,r}$.

Remark 3.22. *The topological structure of $V_{+,r}$ is preserved in $V_{-,r}$, even though the inner edge length of $\{\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)\}$ for any $\iota_r(\vec{v}_i), \iota_r(\vec{v}_j) \in V_{-,r}$ is variable (and rational).*

Example 3.23 (Edge Structure of Λ_r via Circle Inversion). To illustrate the edge structure of $\Lambda_r = (V_r, E_r)$, consider the vertices $\vec{v}_1 = -4\vec{\omega}_0 = (-4, 0)_C$, $\vec{v}_2 = -3\vec{\omega}_0 = (-3, 0)_C$, and $\vec{v}_3 = -2\vec{\omega}_0 = (-2, 0)_C$ along the west axis (the primary ray at phase π with phase pair $(\pi, 0)_\phi$, per Table 5) in $V_{+,1}$ (for admissible $r = 1$, though this generalizes to other admissible $r > 0$ as per Definition 3.14). Their images under the circle inversion map are $\iota_1(\vec{v}_1) = -\frac{1}{4}\vec{\omega}_0 = (-\frac{1}{4}, 0)_C$, $\iota_1(\vec{v}_2) = -\frac{1}{3}\vec{\omega}_0 = (-\frac{1}{3}, 0)_C$, and $\iota_1(\vec{v}_3) = -\frac{1}{2}\vec{\omega}_0 = (-\frac{1}{2}, 0)_C$, which form edges in $V_{-,1}$, as shown in Figure 3. The outer edge $\{\vec{v}_1, \vec{v}_2\} \in E_{+,1}$ (where $E_{+,1} = E_1 \cap (V_{+,1} \times V_{+,1})$, per Definition 3.24) has length $\|\vec{v}_2 - \vec{v}_1\| = 1$ (standard lattice spacing). Under inversion, this maps to the inner edge $\{\iota_1(\vec{v}_1), \iota_1(\vec{v}_2)\} \in E_{-,1}$ (where $E_{-,1} = E_1 \cap (V_{-,1} \times V_{-,1})$) of length $\|\iota_1(\vec{v}_2) - \iota_1(\vec{v}_1)\| = \frac{1}{12}$, as the images are closer to the origin (e.g., $\|\iota_1(\vec{v}_1)\| = \frac{1}{4} < \|\vec{v}_1\| = 4$). Similarly, the length-1 outer edge $\{\vec{v}_2, \vec{v}_3\} \in E_{+,1}$ maps to the length-$\frac{1}{6}$ inner edge $\{\iota_1(\vec{v}_3), \iota_1(\vec{v}_2)\} \in E_{-,1}$. Despite variable geometric lengths, adjacency is preserved combinatorially by transport along ι_r, as formalized in Definition 3.25 and Remark 3.26 (where the conformality of circle inversion [37] provides the underlying geometric motivation). For combinatorial algorithms (e.g., unweighted shortest paths under the discrete dual metric of Definition 4.27), a 3-hop path in $V_{+,1}$ maps to a 3-hop path in $V_{-,1}$. Note: For weighted graphs, the continuous dual metric of Definition 4.31 can normalize the lengths if needed.

Definition 3.24 (Zone Subgraphs). The *outer zone subgraph* $\Lambda_{+,r} = (V_{+,r}, E_{+,r})$ is the induced subgraph of Λ_r on $V_{+,r}$, where $E_{+,r} := E_r \cap (V_{+,r} \times V_{+,r})$. The *boundary zone subgraph* $\Lambda_{T,r} = (V_{T,r}, E_{T,r})$ is the induced subgraph of Λ_r on $V_{T,r}$, where $E_{T,r} := E_r \cap (V_{T,r} \times V_{T,r})$. The *inner zone subgraph* $\Lambda_{-,r} = (V_{-,r}, E_{-,r})$ is the induced subgraph of Λ_r on $V_{-,r}$, where $E_{-,r} := E_r \cap (V_{-,r} \times V_{-,r})$.

$\Lambda_{T,r}$ acts as a separator between $\Lambda_{-,r}$ and $\Lambda_{+,r}$. Squared norms of vertices in $\Lambda_{+,r}$ are integers, while those in $\Lambda_{-,r}$ are rational (for rational r^2), which permits exact comparisons without floating-point error. In both $\Lambda_{-,r}$ and $\Lambda_{+,r}$, the squared edge lengths in $E_{+,r}$ and $E_{-,r}$ (subsets of E_r) are rational—uniformly 1 in $\Lambda_{+,r}$ and variable in $\Lambda_{-,r}$—due to inversion scaling factors, as $\Lambda_{-,r}$ is defined as the image of $\Lambda_{+,r}$ under the circle inversion map ι_r. This renders $\Lambda_{-,r}$ countably infinite and discrete, with vertices accumulating near the

origin while remaining distinct. Note that vertices in $\Lambda_{-,r}$ exist at inverted positions: although all vertices in Λ_r share the same phase alignments as points in the base lattice L, those in $\Lambda_{-,r}$ have different norms and thus do not land on L. This yields the "radial dual" structure of Λ_r. As established in Section 4, $\Lambda_{-,r}$ has an isomorphic combinatorial structure to $\Lambda_{+,r}$. Consequently, Λ_r maintains the triangular connectivity patterns and degree-6 vertices of L, but with transformed geometry in $\Lambda_{-,r}$, including variable edge lengths along radial directions. This construction avoids enumeration issues near the origin—such as infinite accumulation in the infinite lattice or precision challenges in finite truncations with radius R—while enforcing exact duality over a naive partition by norm $< r$. It discards any pre-existing points not in the image and establishes balanced zones with $|V_{-,r}| = |V_{+,r}|$ and $|E_{-,r}| = |E_{+,r}|$. Specifically, $\Lambda_{-,r}$ is induced by inverting edges from $\Lambda_{+,r}$: an edge $(\vec{v}_i, \vec{v}_j) \in E_{+,r}$ maps to $(\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)) \in E_{-,r}$. This preserves adjacency by construction, as formalized in Definition 3.25 and Remark 3.26; the underlying conformality of circle inversion [37, 55] is the geometric motivation.

This construction discretizes the continuous circle inversion map ι_r from Subsection 2.4 to ensure that $\Lambda_{-,r}$ mirrors the $\Lambda_{+,r}$ topologically.

The construction above defines the edges of $\Lambda_{-,r}$ by *transporting* the edges of $\Lambda_{+,r}$ along the circle inversion map ι_r. This is a substantive structural distinction worth naming explicitly: edges in $\Lambda_{+,r}$ arise geometrically (unit Euclidean distance in the underlying triangular lattice L), whereas edges in $\Lambda_{-,r}$ arise combinatorially (transported through ι_r from $\Lambda_{+,r}$). The inverted vertex positions lie in $\mathbb{Q}(\sqrt{3})$ and do not form a sublattice of L, so the adjacency relation in $\Lambda_{-,r}$ is not inherited from a native nearest-neighbor structure but is unambiguously defined by transport. We formalize this notion as follows.

Definition 3.25 (Transported Edge Graph Isomorphism). Let $f : V(G_1) \to V(G_2)$ be a bijection between the vertex sets of graphs $G_1 = (V(G_1), E(G_1))$ and $G_2 = (V(G_2), E(G_2))$. We say $E(G_2)$ is the *transported edge structure induced by f from G_1* if $E(G_2) = \{\{f(\vec{u}), f(\vec{v})\} : \{\vec{u}, \vec{v}\} \in E(G_1)\}$. When this holds, f is a *transported edge graph isomorphism* from G_1 to G_2—equivalently, an ordinary graph isomorphism whose codomain edge structure is defined by transporting the domain's edges through f.

Remark 3.26. *By Definition 3.21, the edge set $E_{-,r}$ is exactly the transported edge structure induced by ι_r from $\Lambda_{+,r}$, and $\iota_r : \Lambda_{+,r} \to \Lambda_{-,r}$ is a transported edge graph isomorphism. Since $\iota_r|_{V_{+,r}}$ is a bijection onto $V_{-,r}$ (Proposition 4.38) and edges are unordered pairs of vertices, the induced map on 2-element vertex subsets is also a bijection. Therefore, it follows that $\iota_r : (\Lambda_{+,r}, E_{+,r}) \to (\Lambda_{-,r}, E_{-,r})$ is a graph isomorphism in the standard sense (no duplicated or spurious edges). This framing distinguishes combinatorial adjacency (preserved across zones by transport) from geometric adjacency (a property of $\Lambda_{+,r}$ as a subgraph of L that does not survive inversion as a geometric notion). The combinatorial preservation is what underwrites the dualities of Section 4 and the equivariant encodings of Section 5.*

While Λ_r provides a complete theoretical foundation with countably infinite zones, practical implementations require finite approximations that retain the exact bijections and symmetries. For this, we define a truncated version below, which induces balanced finite subgraphs suitable for simulations and scalable algorithms.

Definition 3.27 (Truncated Radial Dual Triangular Lattice Graph). For admissible $r > 0$ and truncation radius R with $R \gg r$, the *truncated radial dual triangular lattice graph* $\Lambda_r^R = (V_r^R, E_r^R)$ is the finite induced subgraph of Λ_r on the truncated vertex set $V_r^R := V_{-,r}^R \cup V_{T,r} \cup V_{+,r}^R$, where $V_{+,r}^R := \{\vec{v}_i \in V_{+,r} : \|\vec{v}_i\| \leq R\}$, $V_{-,r}^R := \iota_r(V_{+,r}^R)$, and $V_{T,r}^R := V_{T,r}$ (fully included since $r \ll R$), and $E_r^R := E_r \cap (V_r^R \times V_r^R)$. The truncated zone subgraphs are defined analogously as the induced subgraphs on these sets: the truncated outer zone subgraph $\Lambda_{+,r}^R = (V_{+,r}^R, E_{+,r}^R)$ where $E_{+,r}^R := E_r^R \cap (V_{+,r}^R \times V_{+,r}^R)$, the truncated boundary zone subgraph $\Lambda_{T,r}^R = (V_{T,r}^R, E_{T,r}^R)$ where $E_{T,r}^R := E_r^R \cap (V_{T,r}^R \times V_{T,r}^R)$, and the truncated inner zone subgraph $\Lambda_{-,r}^R = (V_{-,r}^R, E_{-,r}^R)$ where $E_{-,r}^R := E_r^R \cap (V_{-,r}^R \times V_{-,r}^R)$. This construction preserves the exact bijections and phase pair constancy within the truncation bounds, where rays are truncated to finite segments of their infinite counterparts.

This truncation ensures that $|V^R_{-,r}| = |V^R_{+,r}|$ exactly (via the bijection), which enables efficient, symmetry-preserving computations on finite structures of Λ^R_r that approximates the infinite Λ_r with vanishing errors as $R \to \infty$ (detailed in Subsection 3.4).

The radial dual construction has a structural consequence that is central to TQF's divide-and-conquer and parallel zone algorithms: the boundary set $V_{T,r}$ is not merely a convenient separator but a *constant-size, perfectly balanced* one—its cardinality does not grow with the truncation radius. We state and prove this next.

Theorem 3.28 (The Constant-Sized Balanced Bisection Separator Theorem). Let $r > 0$ be admissible and let $R \gg r$ be any truncation radius. Then the boundary vertex set $V_{T,r}$ is a vertex separator of the truncated radial dual triangular lattice graph Λ^R_r with the following properties:

(1) $|V_{T,r}| = 6k$ for some integer $k \geq 1$ (Definition 3.14), *independent of* R;

(2) deleting $V_{T,r}$ from Λ^R_r leaves exactly two connected components, the truncated outer zone subgraph $\Lambda^R_{+,r}$ and the truncated inner zone subgraph $\Lambda^R_{-,r}$; and

(3) the two components are balanced: $|V^R_{+,r}| = |V^R_{-,r}|$.

Consequently, Λ^R_r admits a perfectly balanced vertex separator of constant size $O_r(1)$ on a vertex set of size $|V^R_r| = \Theta(R^2)$.

Proof. Fix an admissible r and a truncation radius $R \gg r$, and recall the partition $V^R_r = V^R_{-,r} \sqcup V_{T,r} \sqcup V^R_{+,r}$ of Definition 3.27, where the outer zone vertices satisfy $\|\vec{v}\| > r$, inner zone vertices satisfy $\|\vec{v}\| < r$, and boundary vertices satisfy $\|\vec{v}\| = r$.

- **Step 1: Classify every edge of Λ^R_r.** By construction (Definition 3.21 and Remark 3.26), every edge of Λ^R_r is of exactly one of three types: an *outer–outer* edge (a geometric unit distance edge of L that joins two outer zone vertices of norm $> r$), an *inner–inner* edge (an edge transported through ι_r that joins two inner zone vertices of norm $< r$, because ι_r sends each outer zone vertex of norm $\|\vec{v}\| > r$ to an inner zone vertex of norm $r^2/\|\vec{v}\| < r$), or an edge *incident to* $V_{T,r}$ (a boundary–boundary or boundary–outer geometric unit distance edge of L, or a boundary–inner twin edge). In particular, no edge joins an inner zone vertex to an outer zone vertex: by Definition 3.21 the edge set consists only of the geometric unit distance edges induced on $V_{+,r} \cup V_{T,r}$, the ι_r-images of the geometric unit distance edges induced on $V_{+,r}$, and the boundary–inner twin edges across $V_{T,r}$, so no edge directly joins $V_{-,r}$ to $V_{+,r}$. Equivalently, the inner zone vertices are inversion images rather than the interior lattice points of L, so a geometric unit distance edge of L that would straddle the boundary circle has no inner zone endpoint to attach to. ☑
- **Step 2: Deleting $V_{T,r}$ disconnects the two zones.** Removing $V_{T,r}$ from Λ^R_r deletes precisely the edges of the third type, leaving only outer–outer and inner–inner edges. Since (Step 1) no surviving edge joins $V^R_{+,r}$ to $V^R_{-,r}$, no path in $\Lambda^R_r - V_{T,r}$ connects an outer zone vertex to an inner zone vertex. Hence $\Lambda^R_{+,r}$ and $\Lambda^R_{-,r}$ lie in distinct components, so $V_{T,r}$ is a vertex separator between them. ☑
- **Step 3: Exactly two components, each connected.** The truncated outer zone subgraph $\Lambda^R_{+,r}$ is connected (Lemma 4.28), and the truncated inner zone subgraph $\Lambda^R_{-,r} = \iota_r(\Lambda^R_{+,r})$ is its image under the transported edge graph isomorphism ι_r (Remark 3.26) and is therefore connected as well. By Step 2 these are mutually disconnected after the removal of $V_{T,r}$, so $\Lambda^R_r - V_{T,r}$ has exactly two connected components, namely $\Lambda^R_{+,r}$ and $\Lambda^R_{-,r}$. ☑
- **Step 4: Balance and constant size.** The truncation convention $V^R_{-,r} = \iota_r(V^R_{+,r})$ together with the injectivity of ι_r (Proposition 4.38) gives $|V^R_{-,r}| = |V^R_{+,r}|$, so the two components are balanced. Finally, $|V_{T,r}| = 6k$ with $k \geq 1$ by Definition 3.14, a quantity that depends only on r (through the quadratic-form representations of $N = r^2$) and not on R. Since $|V^R_r| = \Theta(R^2)$, the separator $V_{T,r}$ has size $O_r(1)$, i.e. constant in R. ☑

□

Remark 3.29 (Breaking the $\sqrt{|V|}$ Barrier for Two Zones). *The constant separator size is a genuine consequence of the radial dual* gluing*, not an artifact of the lattice alone. A triangular-lattice disc of radius R on its own—like the square grid—has no balanced vertex separator smaller than $\Theta(R) = \Theta(\sqrt{|V|})$, matching the asymptotically tight separators guaranteed for bounded-degree planar graphs by the Lipton–Tarjan theory [53]. The radial dual construction circumvents this for the specific inner/outer bipartition: by gluing the outer zone to its circle inversion image along the single norm-r shell $V_{T,r}$, it makes that bipartition separable by a set whose size is fixed at $6k$ regardless of R. In other words, the cheapest balanced cut between the two zones drops from the generic $\Theta(\sqrt{|V|})$ to a constant. This is precisely the structural fact that lets the zone swap (Proposition 4.15) and self-duality (Proposition 4.38) operations recurse across the boundary without paying a growing separation cost, and it underpins the symmetric cut and flow applications discussed below.*

Corollary 3.30 (Constant-Cost Inner–Outer Min Cut). For admissible r and any $R \gg r$, the minimum vertex cut that separates $\Lambda^R_{+,r}$ from $\Lambda^R_{-,r}$ in Λ^R_r has size at most $6k = |V_{T,r}|$, which is independent of R. Hence any cut-based or flow-based algorithm that operates across the inner/outer bipartition (e.g. the symmetric min-cut and max-flow routines referenced in Corollary 4.10) only requires a separation overhead cost of $O_r(1)$ that is R-independent.

So why does this matter in practice? Well, let's note what the separator size actually controls. In a standard planar divide-and-conquer approach, the work required to separate the set at each level of the recursion is governed by its size, so a $\Theta(\sqrt{|V|})$ separator requires a boundary cost that grows with the subproblem—the source of the familiar superlinear overheads in separator-based algorithms [53]. Here the inner/outer separator is fixed at $6k$, so the cross-boundary work requires $O_r(1)$ at every level of the recursion and at every truncation radius R. The same bound governs the flow side: since the minimum vertex cut between the zones is at most $6k$, a max flow computation across the bipartition admits at most $6k$ augmenting paths under the unit vertex capacities, regardless of how large the lattice grows [47]. So in terms of implementation, the boundary behaves as a fixed-width interface: one computes on a single zone, hands off through the $6k$ boundary vertices, and recovers the other twin zone through the bijection (Proposition 4.38), where the interface cost never scales with the data. This is what makes the zone parallel and mirror-and-reuse strategies of Section 3.4 a net win rather than a break-even tie.

With the graph Λ_r and its zones defined, we now introduce discretized rays and paths to characterize radial structures.

Definition 3.31 (Graph Ray). A *graph ray* in the radial dual triangular lattice graph Λ_r is an infinite sequence of distinct vertices $\gamma = \{\vec{v}_i\}_{i=1}^{\infty} \subseteq V(\Lambda_r)$ that emanates from (but does not include) the origin along a lattice radial ray in the base triangular lattice L, such that consecutive vertices $\vec{v}_i$ and $\vec{v}_{i+1}$ are adjacent in $E(\Lambda_r)$ for all $i \in \mathbb{N}$ and the phase $\langle \vec{v}_i \rangle = \langle \vec{v}_{i+1} \rangle = \theta$ (fixed) for all $i \in \mathbb{N}$, ordered by strictly increasing Euclidean norms $\|\vec{v}_i\| < \|\vec{v}_{i+1}\|$ (the simultaneous adjacency and phase-constancy conditions are satisfied only along the six primary directions $\theta = t\pi/3$ for $t \in \mathbb{Z}_6$, as elaborated in Remarks 3.35 and 3.36 below, such that non-primary directions are handled separately via shortest paths with bounded phase deviation). Equivalently, $\gamma = \{k\vec{p} \mid k \in \mathbb{N},\ \vec{p} \in L$ primitive, $\langle \vec{p} \rangle = \theta$ fixed$\}$, where the sequence respects the zone-adjacent structure of Λ_r (i.e., inner-to-boundary and boundary-to-outer connections via twin edges when crossing $V_{T,r}$).

Definition 3.32 (Finite Graph Ray). A *finite graph ray* in the truncated radial dual triangular lattice graph Λ^R_r is a finite initial segment of a graph ray, i.e., a finite sequence of distinct vertices $\gamma = \{\vec{v}_i\}_{i=1}^{n} \subseteq V(\Lambda^R_r)$ (for $n \in \mathbb{N}$) that emanates from (but does not include) the origin along a lattice radial ray in L, such that consecutive vertices $\vec{v}_i$ and $\vec{v}_{i+1}$ are adjacent in $E(\Lambda^R_r)$ for all $i = 1, \ldots, n-1$, the phase $\langle \vec{v}_i \rangle = \theta$ (fixed) for all $i = 1, \ldots, n$, and ordered by strictly increasing Euclidean norms $\|\vec{v}_i\| < \|\vec{v}_{i+1}\|$ up to the truncation radius R.

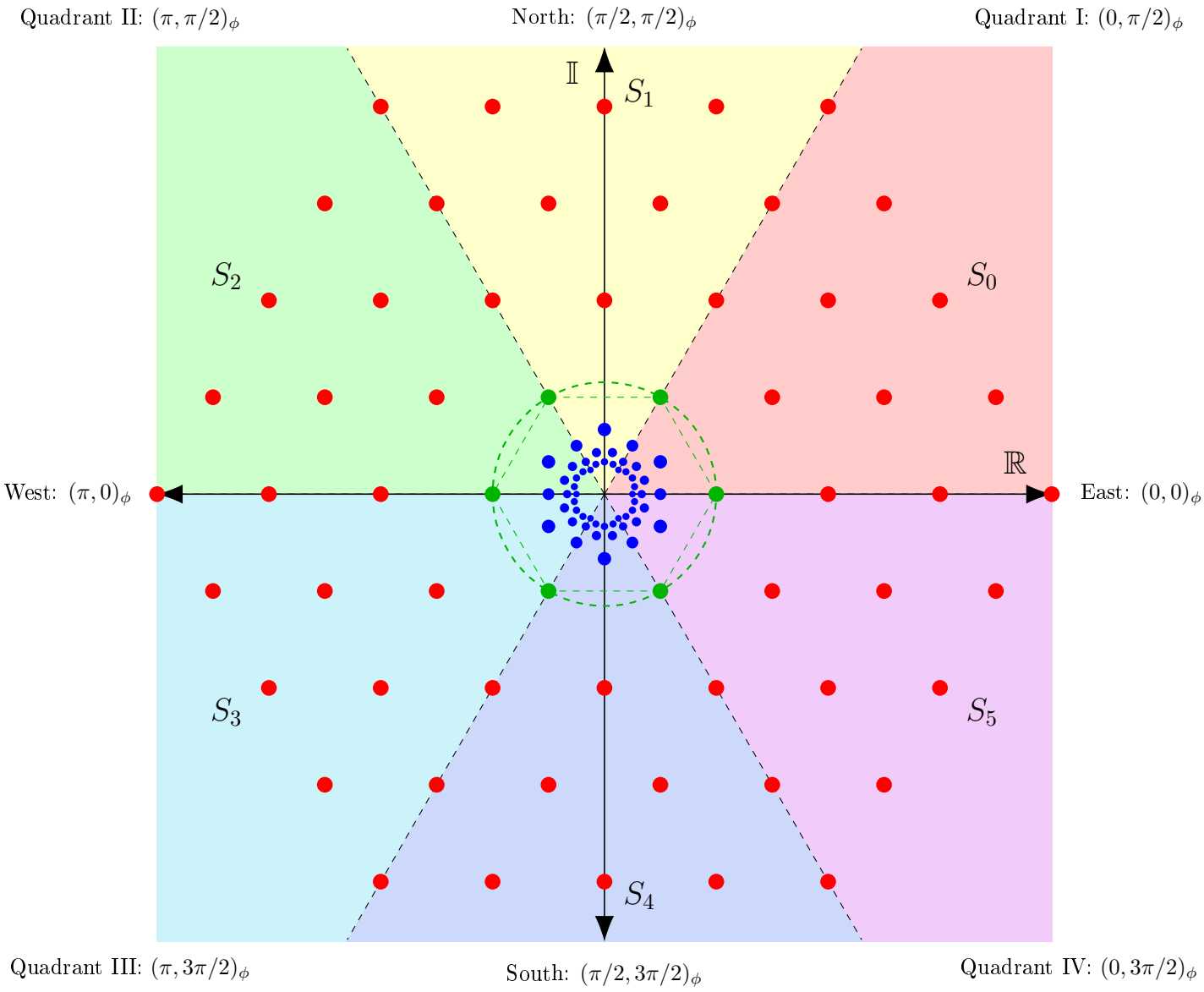

FIGURE 3. Zone and angular sector partitioning on a TQF radial dual triangular lattice graph Λ_1^4 (with admissible inversion radius $r = 1$ and truncation radius $R = 4$). The boundary zone subgraph $\Lambda_{T,1}^4$ consists of six green vertices with uniform size that form a hexagon to encode an order-6 cyclic group, which align to precisely intersect the six angular sector boundary primary rays. The outer zone subgraph $\Lambda_{+,1}^4$ has red vertices with uniform size. The inner zone subgraph $\Lambda_{-,1}^4$ has blue vertices that decrease in (perceived) size as they approach the punctured origin (for illustration only). We see an example of Escher-inspired reflected twin paths: the red connected path in $\Lambda_{+,1}^4$ maps to the blue connected path in $\Lambda_{-,1}^4$ under circle inversion. For a dynamic version with randomly connected paths, see Appendix A (with a screenshot in Figure 4).

Definition 3.33 (Primary Graph Ray). A *primary graph ray* in Λ_r is a graph ray along one of the six primary lattice radial rays at phases $t\pi/3$ for $t \in \mathbb{Z}_6$ (per Definition 3.7), which bound the angular sectors S_t. Equivalently, a *primary finite graph ray* in Λ_r^R is a finite graph ray along the corresponding truncation of one of these primary lattice radial rays.

Remark 3.34. *The phase constancy requirement in Definition 3.31 ensures that graph rays are radial (origin-aligned), which corresponds to lattice radial rays of Definition 3.6 with the additional properties of adjacency and primitivity. The general lattice rays of Definition 3.5 do not have direct graph analogs, which supports TQF's emphasis on origin-centered rotational symmetries such as order-6 under D_6.*

Remark 3.35. *Graph rays are infinite in the six primary phases of the six angular sector boundary primary rays of Λ_r. For other phases, we may consider approximating paths with a bounded phase variation.*

Remark 3.36. *Exact infinite graph rays exist only along the six primary graph rays bounding the angular sectors of Λ_r because their primitive direction vectors have unit Euclidean length in $\Lambda_{+,r}$ (which assures adjacency between consecutive multiples) and topological adjacency (1-hop under the discrete dual metric) is preserved in $\Lambda_{-,r}$ via the induced graph isomorphism (in the upcoming Corollary 4.21). For non-primary directions, the shortest paths in the graph metric approximate radial rays with bounded phase deviation.*

To illustrate graph rays, let's consider a primary graph ray along the constant phase $\pi/3$ in the angular sector S_1 (revisiting the example from Subsection 3.2). For admissible $r = 1$, the outer zone segment of this graph ray consists of $\vec{v}_k = k\vec{\omega}_1$ for integer $k \geq 2$ in $\Lambda_{+,1}$ with Euclidean norms $\|\vec{v}_k\| = k > 1$ (with $k = 1$ giving the fixed boundary vertex $\vec{\omega}_1 \in \Lambda_{T,1}$). The corresponding inverted graph ray in $\Lambda_{-,1}$ is $\iota_1(\vec{v}_k) = \vec{\omega}_1/k$ with Euclidean norms $\|\iota_1(\vec{v}_k)\| = 1/k < 1$, where the phase $\pi/3$ and assigned phase pair $(0, \pi/2)_\phi$ (as per Table 5) both remain constant under the map ι_1. To verify this inversion, let's consider the following pseudocode, which computes inverted positions of the image by using rational arithmetic for exactness:

ALGORITHM PSEUDOCODE 2. Inverting a Graph Ray

```
function InvertGraphRay(graph_ray_vertices, r_sq):
    inverted_graph_ray = []
    for v in graph_ray_vertices:
        norm_sq = ||\vect{v}||^2  # compute squared norm
        inverted_v = (r_sq / norm_sq) * v
        inverted_graph_ray.append(inverted_v)
    return inverted_graph_ray
```

Definition 3.37 (Graph Path). A *graph path* in the radial dual triangular lattice graph $\Lambda_r = (V_r, E_r)$ is a sequence of distinct vertices $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{k-1} \in V_r$ (finite, of length $k \in \mathbb{N}$) or an infinite sequence $\vec{v}_0, \vec{v}_1, \cdots \in V_r$ such that $\{\vec{v}_i, \vec{v}_{i+1}\} \in E_r$ for each $i \geq 0$. Equivalently, it is an ordered chain of adjacent vertices in Λ_r with no repeats (a special case being the graph rays of Definition 3.31).

Remark 3.38. *For the sake of conciseness, we adopt the standard notational convention of directly identifying vertices as elements of their lattice graph or lattice subgraph: that is, for $\vec{v}_i \in V(\Lambda_r)$, we may equivalently write $\vec{v}_i \in \Lambda_r$ (and similarly for the zone subgraphs $\Lambda_{\pm,r}$ and $\Lambda_{T,r}$). This aligns with the construction in Subsection 3.3, where the equal cardinalities $|V(\Lambda_{-,r})| = |V(\Lambda_{+,r})|$ (and thus $|V(\Lambda_r)| = |\Lambda_r|$) follow from the bijective mapping ι_r.*

For an interactive exploration of these structures and dualities, check out our simulation Python script in Appendix A, namely `simulation_01_visualize_random_connections.py`, which generates a dynamic

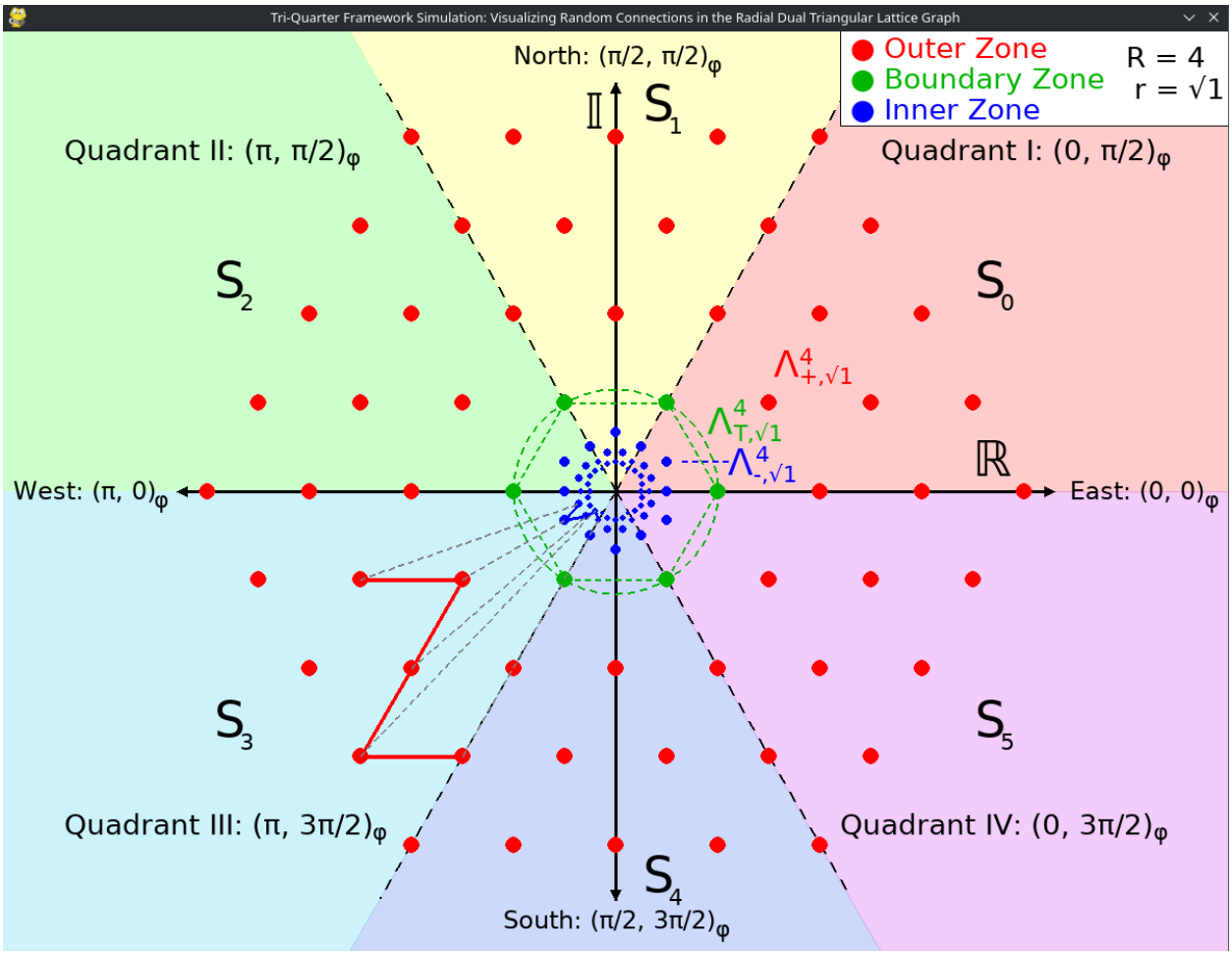

FIGURE 4. A screenshot of our `simulation_01_visualize_random_connections.py` Python script in Appendix A that visualizes randomly selected adjacent edge connections to build twin paths (with lengths ranging from 3 to 5) in the TQF radial dual triangular lattice graph Λ_1^4 (with admissible inversion radius $r = 1$ and truncation radius $R = 4$). We observe the animated Escher-inspired reflections [24] between the randomly connected red outer zone vertices in $\Lambda_{+,1}^4$ and the corresponding blue inner zone vertices in $\Lambda_{-,1}^4$ due to the circle inversion map ι_1. The process repeatedly selects a new pair of twin paths every 5 seconds. The source code is freely available at [56].

visualization of randomly selected paths in $\Lambda^4_{+,1}$ being mirrored in $\Lambda^4_{-,1}$ via circle inversion of ι_1—see screenshot in Figure 4). This simulation demonstrates the animated Escher-inspired reflections [24] in real-time. This source code is freely available online at [56].

To complement the vertex-level zone trichotomy and graph-ray structure of Λ_r, we also equip the edges of $\Lambda_{+,r}$ with a six-valued direction label inherited from the basis vectors of L. This consistent edge framing supports later applications that need to track local direction along graph paths (e.g., canonical planar embedding traversal, primitive-direction-respecting flow algorithms, and any orientation-aware indexing scheme), and we show that the framing behaves predictably under the symmetries of $\mathbb{T}_{24}$.

Definition 3.39 (Consistent Edge Framing on $\Lambda_{+,r}$). Define the six unit primitive direction vectors $\vec{e}_0, \vec{e}_1, \ldots, \vec{e}_5 \in L$ by $\vec{e}_k = \mathcal{R}_{k\pi/3}(\vec{\omega}_0)$, so that $\vec{e}_0 = \vec{\omega}_0$, $\vec{e}_1 = \vec{\omega}_1$, $\vec{e}_2 = \vec{\omega}_1 - \vec{\omega}_0$, and $\vec{e}_{k+3 \bmod 6} = -\vec{e}_k$ for all $k \in \mathbb{Z}_6$ (so the six primitive vectors form three antipodal pairs). For each undirected edge $\{\vec{v}_i, \vec{v}_j\} \in E_{+,r}$ (which has unit Euclidean length in L by construction of $\Lambda_{+,r}$), the difference $\vec{v}_j - \vec{v}_i \in \{\vec{e}_0, \ldots, \vec{e}_5\}$, and we define the *edge framing* $\eta(\{\vec{v}_i, \vec{v}_j\}) \in \{\{0,3\},\{1,4\},\{2,5\}\}$ to be the unordered pair of opposite labels $\{k, k+3 \bmod 6\}$ such that $\vec{v}_j - \vec{v}_i = \vec{e}_k$ (equivalently, $\vec{v}_i - \vec{v}_j = \vec{e}_{k+3 \bmod 6}$). The framing extends to $E_{-,r}$ by transport along ι_r: for the edge $\{\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)\} \in E_{-,r}$, set $\eta(\{\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)\}) := \eta(\{\vec{v}_i, \vec{v}_j\})$.

Lemma 3.40 (Equivariance of the Edge Framing). The edge framing η of Definition 3.39 is equivariant under the action of $\mathbb{T}_{24}$, in the sense that:

(1) under a rotation $\mathcal{R}_{k\pi/3}$, the framing shifts as $\eta \mapsto \eta + k \bmod 6$ (where we interpret addition componentwise on the underlying pair of labels, which preserves the opposite-pair structure since $k + k + 3 \equiv (k+3) + k \bmod 6$);

(2) under each reflection $\sigma \in D_6$, the framing undergoes a fixed permutation of the three direction-pair classes $\{0,3\},\{1,4\},\{2,5\}$; and

(3) under the circle inversion ι_r, the framing on outer edges is preserved exactly (by construction), and the framing on inner edges agrees with that of their outer twins.

Proof.

- **Step 1: Rotations.** A rotation $\mathcal{R}_{k\pi/3}$ acts linearly on L, sending $\vec{e}_j$ to $\vec{e}_{j+k \bmod 6}$. Hence for an outer edge $\{\vec{v}_i, \vec{v}_j\}$ with $\vec{v}_j - \vec{v}_i = \vec{e}_l$, the rotated edge $\{\mathcal{R}_{k\pi/3}(\vec{v}_i), \mathcal{R}_{k\pi/3}(\vec{v}_j)\}$ has difference $\vec{e}_{l+k \bmod 6}$, yielding the cyclic label shift. ☑
- **Step 2: Reflections.** Each reflection $\sigma \in D_6$ is a linear involution of the plane that commutes with negation, so $\sigma(-\vec{e}_k) = -\sigma(\vec{e}_k)$ and hence σ preserves the antipodal pairing $\vec{e}_k \leftrightarrow \vec{e}_{k+3 \bmod 6}$. Consequently, σ induces a well-defined involution on the three direction-pair classes $\{0,3\},\{1,4\},\{2,5\}$. (The induced action of D_6 on these three classes is the standard S_3-action via the quotient $D_6 \twoheadrightarrow S_3$, consistent with Remark 3.42.) ☑
- **Step 3: Circle Inversion ι_r.** By Definition 3.39, the framing on $E_{-,r}$ is defined by transport to equal the framing on the outer twin, so η is preserved exactly on every ι_r-twin pair. ☑

□

Remark 3.41. *Lemma 3.40 reduces the symmetry analysis of any direction-aware computation on Λ_r to a single representative direction class $\{k, k+3\}$, after which the result extends to all of Λ_r by the $\mathbb{T}_{24}$ action on the framing. This makes the framing a useful auxiliary structure for symmetry-reduced traversal scheduling, canonical edge ordering, and any edge-directional indexing scheme that must remain consistent across the bijection ι_r.*

Remark 3.42. *The induced $\mathbb{T}_{24}$-action on the three direction-pair classes $\{0,3\},\{1,4\},\{2,5\}$ factors through the quotient $\mathbb{T}_{24} \twoheadrightarrow D_6 \twoheadrightarrow S_3$, in which the central inversion ι_r acts trivially. The framing groups*

the six unit primitive directions of L into three antipodal pairs (each pair being a direction class up to sign), and the order-12 rotation/reflection group D_6 acts on these three classes as the symmetric group S_3—the standard action of a dihedral group on its three antipodal axis pairs (see, e.g., [17, 18]).

Remark 3.43. *The partition of $E_{+,r}$ (and by transport, $E_{-,r}$) into the three direction-pair classes is a proper 3-edge-coloring of each zone subgraph that is ι_r-invariant and D_6-equivariant. Combined with the proper 3-vertex-coloring c from Subsection 3.2, this gives Λ_r a fully $\mathbb{T}_{24}$-compatible structure of both a proper 3-vertex-coloring and a proper 3-edge-coloring.*

Remark 3.44. *The partition of the medial vertex set into the three direction-pair classes $\{0,3\},\{1,4\},\{2,5\}$ inherited from η is a proper three-coloring of the* medial graph *of Λ_r (the graph whose vertices are the edges of Λ_r and whose edges connect medial vertices that share a face). For the triangular lattice, this medial graph is the* kagome lattice*—a well-studied semi-regular tiling of triangles and hexagons [17]—and the three-coloring is ι_r-invariant and D_6-equivariant via the quotient $D_6 \twoheadrightarrow S_3$, paralleling the proper three-coloring of Λ_r itself by c.*

The infinite graph Λ_r is the object the framework's dualities (Propositions 4.2, 4.15, 4.38) are formulated on, but every implementable computation runs on a finite truncation Λ_r^R. The natural concern is that truncation might negatively impact the results (e.g., a shortest path that uses one extra hop in Λ_r^R than in Λ_r, a bijection that ceases to be onto, or an equivariant labeling that gets clipped off mid-orbit, etc.). The following lemma rules each of these out: every finitely-supported computation on Λ_r is captured *exactly*—not just approximately—by some Λ_r^R for R chosen large enough relative to the input. In other words, the truncated lattice is not merely a numerical proxy for the infinite one, but rather, for any specific finite query it is, at sufficiently large R, mathematically identical to the infinite case. This is the formal bridge that lets the rest of the paper move freely between the infinite Λ_r in proofs and the finite Λ_r^R in simulations.

Lemma 3.45 (Truncation Stability). Fix an admissible inversion radius $r > 0$. For every pair of vertices $\vec{v}_i, \vec{v}_j \in V(\Lambda_r)$ that reside in a common zone ($\Lambda_{+,r}$ or $\Lambda_{-,r}$), there exists $R_0 = R_0(\vec{v}_i, \vec{v}_j) < \infty$ such that for all truncation radii $R \geq R_0$:

(1) both $\vec{v}_i$ and $\vec{v}_j$ are vertices of Λ_r^R;

(2) $d_{hops}^{\Lambda_r^R}(\vec{v}_i, \vec{v}_j) = d_{hops}^{\Lambda_r}(\vec{v}_i, \vec{v}_j)$ (the discrete dual metric agrees on truncated and untruncated graphs); and

(3) ι_r restricts to a bijection between the truncated zone subgraphs $\Lambda_{+,r}^R$ and $\Lambda_{-,r}^R$, with all properties of Proposition 4.38 preserved within Λ_r^R.

Proof.

- **Step 1: Both Vertices Lie in Λ_r^R for R Large Enough.** Since $\vec{v}_i, \vec{v}_j$ have finite Euclidean norms $\|\vec{v}_i\|, \|\vec{v}_j\| < \infty$, both lie within Λ_r^R for any $R \geq \max(\|\vec{v}_i\|, \|\vec{v}_j\|)$. ☑
- **Step 2: The Discrete Dual Metric Agrees on the Truncation.** Connectivity of $\Lambda_{+,r}$ (Lemma 4.28) guarantees a shortest path $\pi^* = (\vec{v}_i = \vec{u}_0, \vec{u}_1, \ldots, \vec{u}_K = \vec{v}_j)$ of length $K = d_{hops}^{\Lambda_r}(\vec{v}_i, \vec{v}_j)$. Since π^* has finitely many vertices, the quantity $R_\pi = \max_l \|\vec{u}_l\|$ is finite. For $R \geq R_\pi$ the entire path π^* is contained in $\Lambda_{+,r}^R$, so $d_{hops}^{\Lambda_r^R}(\vec{v}_i, \vec{v}_j) \leq K$. The reverse inequality $d_{hops}^{\Lambda_r^R}(\vec{v}_i, \vec{v}_j) \geq K$ is automatic since Λ_r^R is a subgraph of Λ_r. The inner zone case follows by applying the same argument to $\iota_r(\vec{v}_i), \iota_r(\vec{v}_j) \in \Lambda_{+,r}$ and transporting through Proposition 4.38. Take $R_0 = \max(\|\vec{v}_i\|, \|\vec{v}_j\|, R_\pi)$. ☑
- **Step 3: Restricted Inversion is a Bijection on Truncated Zones.** Since ι_r is determined pointwise by $\iota_r(\vec{v}) = r^2\vec{v}/\|\vec{v}\|^2$ (depending only on the vertex itself, not on R), its restriction to

$\Lambda^R_{+,r}$ has image $\iota_r(\Lambda^R_{+,r}) = \Lambda^R_{-,r}$ by the symmetric truncation convention of Definition 3.27 (see Subsection 3.3). This restriction inherits bijectivity from ι_r itself. ☑

□

Remark 3.46. *Lemma 3.45 has three concrete consequences for implementations:*

(1) Shortest-path algorithms on Λ^R_r return the same hop-counts as on the infinite Λ_r for any pair of vertices within radius R_0, so BFS/Dijkstra/DFS results are not artifacts of truncation.

(2) The bijection ι_r restricts cleanly to the truncated zones, so any zone swap algorithm (path mirroring, equivariant aggregation, ι_r-orbit replication) "just works" on Λ^R_r without boundary corrections.

(3) The same finite-support argument transports to the $\mathbb{Z}_2$-chain complex of Subsection 5.5—every finitely-supported chain on Λ_r is captured exactly by a sufficiently large Λ^R_r, and the boundary operators commute with the inclusion $\Lambda^R_r \hookrightarrow \Lambda_r$.

Together these guarantee that practical implementations on finite truncations yield results that are exact in the $R \to \infty$ limit rather than merely asymptotic approximations. This complements the geometric truncation-error analysis later in Subsection 3.4, which quantifies how the unresolved area near the origin shrinks as $O(1/R^2)$.

With Λ_r now fully constructed—complete with its zone subgraphs, graph rays, and paths—we've forged a discrete structure via first principles by extending the continuous TQF while preserving exact symmetries and bijections. This sets the stage for practical implementations, where truncation and vertex distributions uphold scalable computations intended for efficiency gains in symmetry-aware applications.

3.4 Applications and Practical Considerations

This zone partitioning facilitates applications like graph partitioning, where the hexagonal boundary subgraph $\Lambda_{T,r}$ serves as a minimal separator for divide-and-conquer strategies. For example, in network flow problems, $\Lambda_{T,r}$ can act as a cut to optimize flow between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ by leveraging the symmetry of Λ_r for efficient computation [47, 53].

The radial dual construction of Λ_r achieves balanced, countably infinite partitions via inversion symmetry to enable efficient computations with constant-time mappings and reduced redundancy through symmetries [18, 49]. This means $\Lambda_{-,r}$ mirrors $\Lambda_{+,r}$ exactly under inversion and avoids explicit enumeration near the origin while preserving completeness. Depending on the problem and the implementation of its solution, symmetries can potentially allow computing on one angular sector and then replicating the results across other angular sectors, similar to divide-and-conquer in parallel algorithms or load balancing [57].

For practical computations on a given Λ^R_r with admissible $r > 0$, we choose a truncation radius R such that $R \gg r$ to truncate Λ_r to Λ^R_r with finite size that is sufficiently large to encode and capture the required computational dynamics. In theory, the graph rays of Λ_r are infinite and extend indefinitely, while in practice, the finite truncation of Λ^R_r approximates the full structure of Λ_r—much like viewing a symmetric pattern through a "looking scope" with a finite lens. The finite graph rays of Λ^R_r are initial segments of their infinite counterparts (per Definition 3.32), which start from the innermost vertex in $\Lambda^R_{+,r}$ (i.e., the vertex with the smallest Euclidean norm strictly greater than r) and extend to the outermost vertex with Euclidean norm at most R, thereby preserving exact phase and bijections within bounds (as the image $\Lambda^R_{-,r}$ is symmetrically truncated with respect to $\Lambda^R_{T,r}$). For example, if we truncate Λ_r at $R = 10$ with $r = 1$, then we obtain Λ^{10}_1 with $|\Lambda^{10}_{-,1}| = |\Lambda^{10}_{+,1}| = 360$ inner/outer vertices plus $|\Lambda^{10}_{T,1}| = 6$ boundary vertices for $|\Lambda^{10}_1| = |\Lambda^{10}_{-,1}| + |\Lambda^{10}_{T,1}| + |\Lambda^{10}_{+,1}| = 726$ total vertices due to inversion symmetry, which enables balanced simulations. The integer squared norms ensure exact zone assignments and avoid floating-point precision errors. In floating-point code, arithmetic comparisons are robust within machine precision $\epsilon \approx 2^{-52}$ [58], but the risk for boundary misclassifications exists in $O(1/R)$ for the large-R limit. However, since we use integer squared norms, we avoid floating-point altogether and thus totally eliminate this risk—as recommended for precision-critical applications—to ensure that float implementations only incur issues if norms (not squared) are used incorrectly. Truncation gaps arise from excluding points beyond the outer radius R and their inverted counterparts below r^2/R near the origin in the inner zone. The area of this unresolved region near

the origin in the inner zone is $O(1/R^2)$, and Λ_r's symmetries facilitate the extrapolation of boundary effects, as quantified in Table 7 (computed via our `compute_truncation_errors.py` Python script in Appendix B.3) and available online [56]. For example, if $R = 10$, then the unresolved area is approximately 0.01% of the total viewed area.

Our `compute_truncation_errors.py` script computes truncation error percentages to quantify the information loss (and consequent reduction in encoding capability) near the punctured origin when truncating Λ_r at radius R to obtain Λ_r^R. Larger R expands the truncation radius and thus the viewed area of the truncated radial dual triangular lattice graph Λ_r^R, but leaves a small "blind spot" in $\Lambda_{-,r}^R$ near the origin. The script fixes $r = 1$ and, for various $R = 4, 10, 20, 50$, computes the unresolved area π/R^2, the total viewed area $\approx \pi R^2$, and the error percentage. As R increases, the error shrinks quadratically—e.g., 0.39% at $R = 4$, near 0% at $R = 50$. This aids in determining "how large R needs to be" to achieve sufficient encoding fidelity and approximation accuracy for our models without excess computation. In Table 7, we observe how rapidly the error percentage vanishes. In other words, as R grows large and the viewed area expands, the blind spot near the origin shrinks like a tiny disk of radius r^2/R, whose area scales as $\pi(r^2/R)^2 = O(1/R^2)$—a vanishingly small fraction of the total viewed area $O(R^2)$. This verifies that the truncated finite Λ_r^R approximates the infinite Λ_r with a sufficiently high degree of accuracy while allowing symmetries to "fill in" the edge details without requiring full recomputation. Our `simulation_01_visualize_random_connections.py` script in Appendix A truncates Λ_r to construct Λ_1^4 and animates random adjacent connections to illustrate balanced simulations and inversion symmetry in practice. For larger R, memory usage scales as $O(R^2)$ per zone, with time complexities for traversals like BFS at $O(|V| + |E|)$ for sequential computing, which (depending on the implementation) may be reduced to $O(|V|/6 + C)$ via symmetry-aware parallel algorithms or distributed load-balanced computing due to Λ_r^R's built-in angular sector parallelism and symmetries (with synchronization overhead cost C).

TABLE 7. Λ_r^R Finite Truncation Error Percentages for Various R (with $r = 1$)

R	**Unresolved Area ($\pi r^4/R^2$)**	**Total Viewed Area ($\approx \pi R^2$)**	**Error (%)**
4	0.1963	50.27	0.3906%
10	0.0314	314.16	0.0100%
20	0.0079	1256.64	0.0006%
50	0.0013	7853.98	0.0000%

Note. "Looking scope" error percentages obtained via our `compute_truncation_errors.py` script in Appendix B.3 for fixed admissible inversion radius $r = 1$ and various truncation radii R. $|\Lambda_{T,r}^R|$ serves as min-cut size in symmetric flows.

Table 8 summarizes vertex distributions across radial zones and angular sectors for the truncated Λ_r^R with admissible $r = 1$ (giving $|\Lambda_{T,1}^R| = 6$ vertices) and $r = \sqrt{7}$ (giving $|\Lambda_{T,\sqrt{7}}^R| = 12$ vertices) for various truncation radii R. (Note: For denser boundaries like $r = \sqrt{7}$, the boundary intersections may be off the primary graph rays but we still count the number of graph rays per angular sector. Additionally, the $r = \sqrt{7}, R = 4$ row has a narrow effective annular width $R - r \approx 1.35$, which makes the absolute per-sector counts small relative to wider configurations; the per-sector counts remain exactly uniform (here 5 outer vertices per sector), just smaller in this near-boundary regime.) The six sectors partition the non-origin vertices into equal counts. The norm-based truncation is D_6-invariant, and the sector rule—decided from the integer coordinate pair (a, b) via the sign of the lattice cross product—is equivariant under the order-6 rotation, so each $\mathbb{Z}_6$ orbit places one vertex in each sector. Thus, every sector contains exactly $|\Lambda_r^R|/6$ vertices, and the primary graph rays carry equal counts across the six directions. We confirmed this with `get_vertex_counts.py`: at $r = 1$ the per-sector totals are $19, 121, 485, 3019$ for $R = 4, 10, 20, 50$, and at $r = \sqrt{7}$ they are $12, 114, 478, 3012$—identical across all six sectors at every R, including the narrow-annulus $r = \sqrt{7}$, $R = 4$ case. These counts illustrate the balanced partitioning that underpins the dualities in Section 4 and they were calculated using our `radial_dual_triangular_lattice_graph.py` Python script in Appendix B.2 and available online [56].

To validate truncation accuracy, one can iteratively increase R and evaluate the correspondingly truncated Λ_r^R until the sequence of resulting error percentages converge to within a given error tolerance (e.g., the path

TABLE 8. Vertex Counts for TQF Radial Dual Triangular Lattice Graphs for Admissible Inversion Radii $r = 1$ and $r = \sqrt{7}$ with Various Truncation Radii R

r	R	$\lvert\Lambda^R_{+,r}\rvert$	$\lvert\Lambda^R_{T,r}\rvert$	$\lvert\Lambda^R_{-,r}\rvert$	$\lvert\Lambda^R_r\rvert$	**Average # Vertices Per S_t**				**Average # Vertices Per S_t Boundary**			
						$\Lambda^R_{+,r}$	$\Lambda^R_{T,r}$	$\Lambda^R_{-,r}$	**Total**	$\Lambda^R_{+,r}$	$\Lambda^R_{T,r}$	$\Lambda^R_{-,r}$	**Total**
1	4	54	6	54	114	9	1	9	19	3	1	3	7
1	10	360	6	360	726	60	1	60	121	9	1	9	19
1	20	1452	6	1452	2910	242	1	242	485	19	1	19	39
1	50	9054	6	9054	18114	1509	1	1509	3019	49	1	49	99
$\sqrt{7}$	4	30	12	30	72	5	2	5	12	2	0	2	4
$\sqrt{7}$	10	336	12	336	684	56	2	56	114	8	0	8	16
$\sqrt{7}$	20	1428	12	1428	2868	238	2	238	478	18	0	18	36
$\sqrt{7}$	50	9030	12	9030	18072	1505	2	1505	3012	48	0	48	96

Note. Vertex counts obtained via our `radial_dual_triangular_lattice_graph.py` script in Appendix B.2. $\lvert\Lambda^R_{T,r}\rvert$ serves as min-cut size in symmetric flows.

lengths stabilize within a given tolerance). In doing so, one calculates errors by comparing differences between larger-R baselines—this aligns with finite-size scaling practices in lattice models [59]. For example, in terms of network flow optimization, one increases R until the cut capacities stabilize to sufficiently approximate the infinite lattice behavior. Hence, one can obtain reliable approximations to the infinite case, where exact bijections hold without gaps.

4 Dualities and Bijective Self-Duality

Building on the practical considerations and balanced vertex distributions of Λ^R_r as outlined in the previous section—which underscore Λ^R_r's scalability for real-world computations—we now delve into the dualities that form the theoretical core of the discrete TQF. These dualities exploit Λ_r's symmetries, including those of the TQF's Inversive Hexagonal Dihedral Symmetry Group $\mathbb{T}_{24}$, to enable exact bijective mappings, unlock reversible transformations, and enhance efficiency in symmetric computational paradigms. By leveraging these symmetries and bijective mappings, the dualities drive advanced graph operations and analyses.

4.1 Combinatorial Duality on Λ_r

Here we prove the extension of the TQF Topological Duality Theorem of [12] in the discrete setting of the TQF radial dual triangular lattice graph Λ_r.

It is known that the field of combinatorics—the study of discrete objects such as graphs and lattice graphs—focuses on their enumeration, arrangement, and interrelations. In this context, combinatorial duality denotes a correspondence between dual structures, wherein the elements and relations in one are faithfully mirrored in the other. For example, in planar graph duality the vertices of the primal graph map to the faces of its dual, thereby preserving adjacency through a topological "flip" [25]. By contrast, our combinatorial duality omits vertex-face mappings to instead target radial separation and modular periodicity modulo 6. Specifically, within Λ_r, the boundary $\Lambda_{T,r}$ serves as a combinatorial dual to both $\Lambda_{-,r}$ and $\Lambda_{+,r}$ simultaneously: it functions as a symmetric vertex separator while carrying directional labels (phase pairs and angular sectors) that remain invariant along origin-emanating graph rays. This structure facilitates symmetric operations, such as circle inversion, that bijectively swap $\Lambda_{-,r}$ and $\Lambda_{+,r}$ while preserving essential properties like path lengths under the discrete dual metric. Unlike classical planar dualities in the literature—which rely on embeddings to equate vertices with faces and edges with crossings [25]—our radial formulation prioritizes norm-based partitioning via symmetric boundary sets and mod 6 angular invariance, which upholds efficient, exact bijective zone swaps that persist discrete path labels without embedding dependencies.

This duality is "combinatorial" because it relies on discrete counts and arrangements of vertices, orientations, and paths, rather than continuous measures (e.g., such as continuous metrics on X). Notably, the symmetric form of $\Lambda_{T,r}$ ties into mod 6 arithmetic by utilizing Λ_r's order-6 rotational symmetry (the cyclic group $\mathbb{Z}_6$, subgroup of the dihedral group D_6): vertices of Λ_r can be indexed mod 6—while their phases and

paths are periodic every $\mathcal{R}_{\pi/3}$—to facilitate modular computations. The subgroups $\mathbb{Z}_2, \mathbb{Z}_3 \subset \mathbb{Z}_6$ catalyze finer decompositions, as we explore in the upcoming corollaries.

Definition 4.1 (Combinatorial Duality of Λ_r). Let Λ_r be the TQF radial dual triangular lattice graph on $X = \mathbb{C} \setminus \{(0,0)_C\}$. For any admissible $r > 0$ such that the symmetric boundary subgraph $\Lambda_{T,r}$ is induced on $V_{T,r}$ (e.g., with order-6 rotational symmetry manifesting as $\mathbb{Z}_6$ periodicity in the minimal case), define $\Lambda_{-,r}$, $\Lambda_{T,r}$, $\Lambda_{+,r}$ as above, with $\Lambda_{-,r} \cup \Lambda_{T,r} \cup \Lambda_{+,r} = \Lambda_r$ and $\Lambda_{-,r} \cap \Lambda_{T,r} = \Lambda_{T,r} \cap \Lambda_{+,r} = \Lambda_{-,r} \cap \Lambda_{+,r} = \emptyset$. $\Lambda_{T,r}$ is *combinatorially dual* to $\Lambda_{-,r}$ and $\Lambda_{+,r}$ if:

(1) $\Lambda_{T,r}$ is a symmetric vertex separator between $\Lambda_{-,r}$ and $\Lambda_{+,r}$,

(2) $\Lambda_{T,r}$ has a structured orientation defined by a map $\phi : \Lambda_{T,r} \to \{0, \frac{\pi}{2}, \pi\} \times \{0, \frac{\pi}{2}, \frac{3\pi}{2}\}$, which assigns phase pairs based on quadrant and axis rules per Tables 1–2, and

(3) the structured orientation ϕ is constant along each graph ray that intersects $\Lambda_{T,r}$, which enables the norm $||\vec{v}_i||$ to consistently separate inner directional paths (from $\Lambda_{-,r}$ to $\Lambda_{T,r}$) from outer directional paths (from $\Lambda_{+,r}$ to $\Lambda_{T,r}$) across Λ_r with respect to $\Lambda_{T,r}$.

Unlike Whitney's combinatorial duality [21], which defines duals where boundary sets correspond to cuts for planarity testing, our radial duality separates via norm trichotomy and mod 6 boundary sets to establish origin-centered bijections (as formalized in Proposition 4.2). In our TQF, the choice of admissible r to form a symmetric set—such as a regular hexagon aligned with the order-6 rotational symmetry in the minimal case—is key. Geometrically, this resulting $\Lambda_{T,r}$ embodies the inherent periodicity, which transforms it into a structured separator that aligns with the triangular tiling's equilateral geometry. Computationally, it enables efficient, constant-time checks for zone membership and path traversals, because symmetries reduce the need for exhaustive searches. Algebraically, it invites group actions—rotations and reflections—that preserve the structure, which bridges to representation theory where $\Lambda_{T,r}$ is comprised of fixed vertices under automorphisms. Combinatorially, the cycle facilitates decompositions into matchings and colorings, as we further explore in Section 5—revealing enumerative insights like balanced partitions of inner and outer vertices. The mod 6 connection further clarifies this: the hexagonal vertex arrangement of $\Lambda_{T,r}$ can be indexed mod 6, with angular sectors $\pi/3 \cdot k$ for k mod 6, to ensure periodic invariance for algorithmic implementations like modular hashing or cyclic traversals.

Proposition 4.2 (Combinatorial Duality of Λ_r). Let L be the base triangular lattice on $X = \mathbb{C} \setminus \{(0,0)_C\}$ with the generalized complex-Cartesian-polar coordinate system of Equation (1), where $\vec{x} = \vec{x}_{\mathbb{R}} + \vec{x}_{\mathbb{I}}$ for $\vec{x} \in L$, with $\vec{x}_{\mathbb{R}} = (x_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$, $\vec{x}_{\mathbb{I}} = (0, x_{\mathbb{I}})_C \in \{0\} \times \mathbb{R}$, and phase pairs $\phi(\vec{x}) = (\langle \vec{x}_{\mathbb{R}} \rangle, \langle \vec{x}_{\mathbb{I}} \rangle)_\phi$ assigned as in Tables 1–2. Let Λ_r be the radial dual graph constructed from L as above. For any admissible $r > 0$ such that $\Lambda_{T,r}$ is induced on a symmetric set $V_{T,r}$ (e.g., with $\mathbb{Z}_6$ periodicity), define $\Lambda_{-,r}$, $\Lambda_{T,r}$, $\Lambda_{+,r}$ as above (with $\Lambda_{-,r} \cap \Lambda_{T,r} = \Lambda_{T,r} \cap \Lambda_{+,r} = \Lambda_{-,r} \cap \Lambda_{+,r} = \emptyset$ and $\Lambda_r = \Lambda_{-,r} \cup \Lambda_{T,r} \cup \Lambda_{+,r}$). Then $\Lambda_{T,r}$ is combinatorially dual to $\Lambda_{-,r}$ and $\Lambda_{+,r}$, where the structured orientation ensures consistent separation between the inner directional paths (approaching $\Lambda_{T,r}$ from $\Lambda_{-,r}$) and the outer directional paths (approaching $\Lambda_{T,r}$ from $\Lambda_{+,r}$) across Λ_r with respect to $\Lambda_{T,r}$.

Proof. Let Λ_r be the TQF radial dual triangular lattice graph on $X = \mathbb{C} \setminus \{(0,0)_C\}$, and for any admissible $r > 0$ such that $\Lambda_{T,r}$ forms a symmetric set, we define $\Lambda_{-,r}$, $\Lambda_{T,r}$, $\Lambda_{+,r}$ as above. We proceed in steps to verify each condition of Definition 4.1:

- **Step 1: Verify Partition and Boundaries on Λ_r.** For selected admissible r, $\Lambda_{T,r}$ forms a finite symmetric set that separates Λ_r into two distinct (countably infinite) disjoint subgraphs: $\Lambda_{+,r}$ and $\Lambda_{-,r}$. In the discrete topology of Λ_r, every vertex in $\Lambda_{T,r}$ is adjacent to vertices in both $\Lambda_{-,r}$ and $\Lambda_{+,r}$ simultaneously, as norms increase/decrease along radial edges. This satisfies condition (1) of combinatorial duality. For larger admissible r, $|\Lambda_{T,r}|$ scales as $6k$ for integer $k \geq 1$, which maintains $\mathbb{Z}_6$ periodicity and separation, while the rotational symmetry ensures uniform graph ray distribution across the angular sectors. ☑

- **Step 2: Verify Structured Orientation on $\Lambda_{T,r}$.** We define $\phi : \Lambda_{T,r} \to \{0, \frac{\pi}{2}, \pi\} \times \{0, \frac{\pi}{2}, \frac{3\pi}{2}\}$ by $\phi(\vec{v}_i) = (\langle \vec{v}_{i,\mathbb{R}} \rangle, \langle \vec{v}_{i,\mathbb{I}} \rangle)_\phi$, using the same rules of Tables 1–2 as in the continuous case of Subsection 2.2. Since the vertices are discrete and angularly spaced by multiples of $\pi/3$ (Lemma 3.10), ϕ uniquely assigns phase pairs to the vertices, which provides a periodic mod 6 orientation on $\Lambda_{T,r}$ for angular consistency. This satisfies condition (2) of combinatorial duality. ☑
- **Step 3: Verify Inner and Outer Directional Separation across $\Lambda_{T,r}$.** For $\vec{v}_i \in \Lambda_{T,r}$, consider sequences of adjacent vertices that approximate rays emanating from (but not including) the origin (punctured as per Subsection 2.1) through $\vec{v}_i$, starting from nearest neighbors and following nearest-neighbor edges in radial directions. Inner directional paths from $\Lambda_{-,r}$ have norms approaching r from below along graph paths, while outer directional paths from $\Lambda_{+,r}$ approach from above. Graph rays provide shortest-path approximations to continuous rays in the graph metric (hop distance—formalized as the discrete dual metric in Subsection 4.4) for primitive direction vectors, where phase constancy holds along these paths as angular sectors are invariant mod 6 (Lemma 3.10). The norm trichotomy (Equation 5) distinguishes between the inner and outer directions exactly because norms are discrete and comparable without ambiguity. Along each of the six primary graph rays, the phase pair remains constant by Lemma 3.10, which yields exact separation. This satisfies condition (3) of combinatorial duality. ☑

Thus, $\Lambda_{T,r}$ is combinatorially dual to $\Lambda_{-,r}$ and $\Lambda_{+,r}$. □

Now since a graph ray is a discrete encoding of a continuous ray emanating from the origin, it consists of a path of adjacent vertices that maintains constant phase and intersects $\Lambda_{T,r}$. Invariance means these paths "look the same" before and after duality operations (e.g., swapping inner/outer zones). The number of such graph rays and their directional labels (phase pairs and angular sectors) stay consistent. This is like preserving edge labels in a graph isomorphism. This is useful for algorithms such as BFS [33], depth-first search (DFS) [34], or shortest-path computations [35] in graph traversals or network routing. Symmetry ensures balanced workloads. For example, graph rays can be binned into balanced mod 6 groups and then processed in parallel without recalculating directions, which leverages exact bijective mappings and phase pair assignments for efficient invertible operations. Moreover, for instance, in network routing on triangular lattices, graph rays enable symmetric traversals that optimize load balancing while preserving directional consistency [60]. Similarly, in robotics path planning, graph rays facilitate balanced exploration in lattice-based environments, which promotes multi-agent coordination without redundant computations [26, 36, 61]. These symmetry-driven efficiencies extend to advanced encodings, such as the trihexagonal six-coloring discussed in Section 5, which further drive the optimization of parallel algorithms and data structures.

Corollary 4.3 (Adaptive Boundary Scaling with Radial Duality and Isomorphic Bijection). For admissible $r > 0$ with $|\Lambda_{T,r}| = 6k$ and integer $k > 1$, the combinatorial duality extends in a similar way. The graph ray count per angular sector increases proportionally, which preserves the mod 6 partitioning and phase constancy. The bijective isomorphism via ι_r follows from the conformal mapping.

Corollary 4.4 (Uniform Angular Sector Partitioning of Hexagonal Boundary Radial Rays). The lattice radial rays (Definition 3.6) that pass through vertices in $\Lambda_{T,r}$ partition it into exactly six equitable classes under the rotations in D_6 (one per angular sector S_t, per Lemma 3.10). Rotational symmetry implies the ray density per angular sector remains uniform—specifically, $|\Lambda_{T,r}|/6$, like one per angular sector for the minimal admissible $r > 0$ (where $|\Lambda_{T,r}| = 6$) or two per angular sector for $r = \sqrt{7}$ (where $|\Lambda_{T,\sqrt{7}}| = 12$), as in Table 6.

Remark 4.5. *Along each graph ray γ that emanates from (but does not include) the origin—which aligns with a lattice radial ray in the base triangular lattice L—the inner segment $\gamma \cap \Lambda_{-,r}$ and outer segment $\gamma \cap \Lambda_{+,r}$ are dual to each other via the Escher reflective duality (Proposition 4.15) and contain the same number of vertices due to the bijective self-duality induced by ι_r. Although the vertex density (i.e., number of vertices per unit length) can vary slightly across different graph rays—since Λ_r's vertices pack more tightly*

at some phases than others (a fundamental characteristic of L)—this setup upholds modular decompositions in graph algorithms, such as distributing traversals across angular sectors for better load balancing.

The equidistribution arises because the boundary vertices of $\Lambda_{T,r}$ form complete orbits under the action of the order-6 Eisenstein unit group, which ensures that exactly k vertices per $\pi/3$-radian angular sector S_t for admissible r with $|\Lambda_{T,r}| = 6k$. For instance, when $N = 7$, exactly two distinct graph rays intersect $\Lambda_{T,r}$ per angular sector S_t, which is confirmed by the representations in Table 9 generated using our `compute_boundary_vertices.py` script from Appendix B.4 (available online [56]).

TABLE 9. Explicit Boundary Vertices of $\Lambda_{T,\sqrt{7}}$ for Admissible Inversion Radius $r = \sqrt{7}$ ($N = 7$): Illustrating Uniform Equidistribution with Exactly Two Vertices per Angular Sector for $\mathbb{Z}_6$ Rotational Symmetry of D_6

Sector	**Vertex 1** (a, b)	**Phase 1 (rad)**	**Vertex 2** (a, b)	**Phase 2 (rad)**
S_0	(2,1)	0.333	(1,2)	0.714
S_1	(-1,3)	1.381	(-2,3)	1.761
S_2	(-3,2)	2.428	(-3,1)	2.808
S_3	(-2,-1)	3.475	(-1,-2)	3.855
S_4	(1,-3)	4.522	(2,-3)	4.903
S_5	(3,-2)	5.569	(3,-1)	5.950

Note. Boundary vertices obtained via our `compute_boundary_vertices.py` script in Appendix B.4.

For instance, in parallel BFS on symmetric lattices, graph rays can be partitioned into mod 6 groups for load-balanced processing across threads, processors, or nodes. This reduces time from $O(|V| + |E|)$ to approximately $O((|V| + |E|)/6 + C)$, where C includes cross-angular-sector merge/synchronization costs—typically $O(\sqrt{|V|})$ from the $O(R)$ cross-angular-sector edges (with $R \sim \sqrt{|V|}$) and synchronization overhead (e.g., $O(R)$ barriers in level-sync BFS). For this we assume balanced angular sectors (each $\sim|V|/6$ vertices) and $p = 6$ processors to leverage the equitable partitioning from our corollaries on rotational invariance.

Quantitatively, for a lattice graph with $|V| = 10^6$ and uniform ray distribution, sequential BFS is $O(10^6)$. Parallel BFS yields approximately 1.7×10^5 operations per processor, with theoretical speedups up to 6x in idealized balanced cases—though practical implementations on lattices may achieve ∼2-4x due to cross-sector edges and truncation effects [57, 62, 63].

The following pseudocode illustrates this at a high level: perform $O(|V|)$ preprocessing to bin the rays by angular sector mod 6, assign the bins to processors for local BFS at $O((|V| + |E|)/6 + C)$ per processor (assuming balanced distribution via rotational symmetry), and merge the results via rotational symmetry (which exploits phase-pair constancy to infer the cross-sector distances) without recalculating the cross-sector edges. In practice, BFS across Λ_r^R requires us to handle the cross-sector edges (with a synchronization overhead cost C) through a synchronization mechanism such as message passing (e.g., via MPI or OpenMP) or barriers for the level frontiers.

ALGORITHM PSEUDOCODE 3. Symmetry-Reduced Parallel BFS with level-by-level scheduling for traversing Λ_r. The angular sectors are binned for load balancing, with synchronization via barriers to handle cross-sector edges. This algorithm is for illustrative purposes only, because full parallel implementations will likely require a proper synchronization mechanism.

```
function SymmetryReducedParallelBFS(graph Λ_r, start_vertices):
    # Step 1: Preprocess - Bin vertices by sector mod 6
    sectors = array of 6 empty lists
    for each vertex v in Λ_r:
        phase = arg(v)  # Compute phase
        k = floor(6 * phase / (2π)) mod 6
        sectors[k].append(v)
```

```
    # Step 2: Parallel local BFS on each sector with synchronization
    distances = {}  # Global distances
    visited = set() # Global visited set
    current_queue = start_vertices  # Initial frontier
    while current_queue not empty:
        next_queue = []  # Next level frontier
        parallel for k in 0 to 5:  # Process sectors in parallel (e.g., threads)
            local_graph = induced subgraph on sectors[k]  # Subgraph for sector k
            # Local frontier portion
            local_queue = intersection of current_queue and sectors[k]
            # Local traversal (intra-sector)
            local_distances = BFS(local_graph, local_queue)
            # Collect cross-sector neighbors for global sync
            cross_neighbors = []  # List for pending updates
            for v in local_queue:
                for neighbor in Λ_r.neighbors(v):  # Full graph neighbors
                    if neighbor not in visited and neighbor not in sectors[k]:
                        cross_neighbors.append(neighbor)
            # Merge locally (with locks if needed)
            update global distances with local_distances

        # Barrier: Exchange/propagate cross-sector neighbors across processors
        # merge updates to next_queue and distances
        synchronize()
        current_queue = next_queue  # Advance to next level

    # Step 3: Merge via rotational invariance (post-sync adjustments if needed)
    return distances
```

This kind of symmetry extends to other traversal algorithms (e.g., like DFS or Dijkstra) in areas such as network analysis or robotics, where it helps reduce redundancy and improve efficiency. Path invariance like this provides a solid approach to maintain structural consistency through transformations, which can be applied to streamline tasks in graph theory. In symmetric networks, such predictability leads to computations that are not only faster but easier to verify.

Scaling $\Lambda_{T,r}$ to denser configurations with $|\Lambda_{T,r}| = 6k$ and $k > 1$ enhances granularity while preserving mod 6 partitioning and phase constancy (as in Corollary 3.11). Per-vertex operations remain $O(1)$ due to symmetry, while boundary-specific operations scale as $O(|\Lambda_{T,r}|)$ and remain "parallel friendly" for algorithmic parallelization across angular sectors.

The $\mathbb{Z}_6$ cyclic group acts like clock arithmetic (mod 6), where we rotate Λ_r by $\mathcal{R}_{\pi/3}$ increments. Invariance means that properties (like zones or directional labels) don't change under these spins, so you can compute on one "slice" (a $\pi/3$ angular sector, per Lemma 3.10) and copy results to the rest. Depending on the problem and implementation, this can potentially reduce the computational effort by a factor of up to 6x in symmetric algorithms [57, 63, 64]. This approach to symmetry exploitation mirrors applications in image processing, where rotational symmetries reinforce efficient detection and redundancy-free computations [29, 65]. It also applies to parallel graph algorithms, which divide-and-conquer data into symmetric chunks for faster processing [64].

Corollary 4.6 ($\mathbb{Z}_6$ Symmetry Exploitation of Angular Sector Decompositions for Parallel Graph Speedups). Under the action of the cyclic subgroup $\mathbb{Z}_6$ (generated by rotations $\mathcal{R}_{k\pi/3}$ for $k \in \mathbb{Z}_6$), the inner zone subgraph $\Lambda_{-,r}$, boundary zone subgraph $\Lambda_{T,r}$, and outer zone subgraph $\Lambda_{+,r}$ are invariant (since the Euclidean norm $\|\vec{v}_i\|$ is preserved), and the distribution of phase pairs $\phi(\vec{v}_i)$ is preserved up to cycling (as established for the angular sectors S_t in Lemma 3.10). This enables symmetry-reduced analysis: computations on one angular sector S_t extend to the full radial dual triangular lattice graph Λ_r by applying the group actions of $\mathbb{Z}_6$. This potentially reduces the computational effort by a factor of up to six in parallel algorithms that fully leverage the symmetry, such as idealized parallel traversals with balanced sectors (e.g., BFS at $O(|V|/6 + C)$, per Subsection 4.1).

Remark 4.7. *This $\mathbb{Z}_6$ rotational invariance effectuates the optimization of computational resources. For problems that can be divided, tuned, and framed to respect this symmetry—such as uniform vertex labels or evenly distributed data—one angular sector S_t suffices to infer the results for the entire Λ_r by applying the group actions of $\mathbb{Z}_6$, given that these actions hold without disruptions or interference from boundary irregularities or finite truncation distortions. Thus, we can distinguish two key cases of symmetry exploitation:*

- ***Fully Symmetric Exploitation**: Global aggregates (e.g., total counts or sums in balanced truncations of Λ_r^R) emerge efficiently as sixfold replications of computations on a single angular sector S_t, due to the subgroup decompositions following Proposition 4.2. For example, with uniformly distributed vertices, these aggregates can be computed once on one S_t and rotated outward via $\mathbb{Z}_6$ actions to cover the full structure without recomputation.*
- ***Partial Symmetry Exploitation**: If data asymmetries or finite truncations with edge effects disrupt the mod 6 partitioning (e.g., as observed in the $O(1/R^2)$ gaps of Table 7), then we can still pursue load-balanced parallel radial sweeps over the angular sectors S_t with modest synchronization overhead $C = O(\sqrt{|V|})$. Yet for problems demanding exhaustive traversals (e.g., comprehensive brute-force enumeration of all unique vertices in Λ_r^R)—the symmetry may still boost efficiency via partitioning into independent sets from the upcoming equivariant encodings (Definition 5.2)—but certain sub-solution specifics cannot be programmatically derived (and replicated) across angular sector boundaries—though the TQF can still trim the computational burden via such encodings (e.g., as observed in our ∼2x mirroring speedup results of Section 6, which is assured by $\mathbb{T}_{24}$-equivariant bijections per Corollary 4.19).*

Subgroups are "mini-groups" inside $\mathbb{Z}_6$ that serve as the fundamental building blocks for its structure. The central order-2 rotation subgroup $\mathbb{Z}_2 = \langle \mathcal{R}_\pi \rangle$ is generated by the rotation $\mathcal{R}_\pi$ by π radians about the origin—known as the *half-turn* because it sends every point to its diametrically opposite point, completing exactly half of a full rotation. The half-turn pairs opposite angular sectors like a balanced seesaw across the origin (e.g., $\{S_0, S_3\}$, $\{S_1, S_4\}$, and $\{S_2, S_5\}$ via $\mathcal{R}_\pi$ operations).

The half-turn earns its central role from three converging perspectives. Geometrically, $\mathcal{R}_\pi$ is the unique nontrivial rotation of $\mathbb{C}$ that coincides with the linear map $\vec{x} \mapsto -\vec{x}$ (point reflection through the origin), which is why it pairs each lattice point with its antipode and induces the antipodal pairing on the six primitive direction vectors $\vec{e}_k \leftrightarrow \vec{e}_{k+3 \bmod 6}$ that already underwrites the edge framing of Definition 3.39. Algebraically, $\mathcal{R}_\pi$ is the unique element of order 2 inside the rotation subgroup $\mathbb{Z}_6 \leq D_6$; it lies in the center of D_6 and in fact realizes the direct-product decomposition $D_6 \cong D_3 \times \langle \mathcal{R}_\pi \rangle$ exploited in the upcoming discussion of $\mathbb{T}_{24}$ following Corollary 4.19. Computationally, half-turn pairings are particularly useful for symmetry-reduced algorithms: any quantity computed on three "representative" sectors $\{S_0, S_1, S_2\}$ extends to the remaining three sectors $\{S_3, S_4, S_5\}$ by a single application of $\mathcal{R}_\pi$, halving the work in problems where the order-6 cyclic symmetry can't be fully exploited but the order-2 antipodal symmetry can (e.g., for data with even/odd parity along the radial direction, or for algorithms whose third-of-a-rotation cost is dominated by their half-of-a-rotation cost).

Note that this is the central $\mathbb{Z}_2 \leq \mathbb{Z}_6 \leq D_6$, distinct from the axis-reflection $\mathbb{Z}_2$ subgroups of D_6 considered in the upcoming Corollary 4.25 (where reflection σ_0 across the real axis yields a different sector pairing $\{S_0, S_5\}, \{S_1, S_4\}, \{S_2, S_3\}$). $\mathbb{Z}_3$ (order-3 rotations) cycles every third S_t like a three-way switch to generate interleaved triples (e.g., $\{S_0, S_2, S_4\}$ and $\{S_1, S_3, S_5\}$ via $\mathcal{R}_{2\pi/3}$ operations, which partition the hexagonal structure into two equilateral triangular cycles that rotate symmetrically). These subgroups create nested half-turn pairings or cycles while preserving the overall combinatorial duality. For instance, $\mathbb{Z}_2 = \langle \mathcal{R}_\pi \rangle$ enables binary decompositions like splitting graph rays into half-turn-paired halves for efficient searches (e.g., check one pair and then map via $\mathcal{R}_\pi$). $\mathbb{Z}_3$ supports ternary clustering, such as grouping angular sectors for load-balanced processing in multi-core systems. By varying the admissible r (e.g., scaling outward to larger symmetric cycles or inverting inward for self-similar copies), this enables recursive divide-and-conquer strategies—breaking a graph into halves or thirds for faster searching—which is useful in hierarchical data structures or multi-level algorithms.

Corollary 4.8 (Exploiting Hierarchical Decompositions from Binary-Ternary Cyclic Subgroup Actions). The subgroups $\mathbb{Z}_2 = \langle \mathcal{R}_\pi \rangle$ (the central order-2 rotation) and $\mathbb{Z}_3 = \langle \mathcal{R}_{2\pi/3} \rangle$ (order-3 rotations) of $\mathbb{Z}_6$ induce finer dualities and partitions that preserve phase pair constancy and zone separation. This enables hierarchical graph decompositions. Specifically, $\mathbb{Z}_2 = \langle \mathcal{R}_\pi \rangle$ provides a binary partitioning by pairing opposite angular sectors (e.g., $\{S_0, S_3\}$, $\{S_1, S_4\}$, and $\{S_2, S_5\}$ via $\mathcal{R}_\pi$)—it creates half-turn pairs across the origin. $\mathbb{Z}_3$ enables ternary partitioning by cycling the angular sectors into interleaved triples (e.g., $\{S_0, S_2, S_4\}$ and $\{S_1, S_3, S_5\}$ via $\mathcal{R}_{2\pi/3}$)—it divides the structure into three symmetric subsets like a three-way cycle or switch. These rotational $\mathbb{Z}_2$ and $\mathbb{Z}_3$ subgroups of $\mathbb{Z}_6$ are distinct from the reflection-based $\mathbb{Z}_2$ subgroups generated by axis reflections in D_6 (treated separately in Corollary 4.25).

Remark 4.9. *The binary ($\mathbb{Z}_2$) and ternary ($\mathbb{Z}_3$) partitions can be combined to generate a hierarchical structure because the direct product $\mathbb{Z}_6 \cong \mathbb{Z}_2 \times \mathbb{Z}_3$ allows nested decompositions. By combining radial layering with angular sector decomposition, we achieve a systematic approach to organize Λ_r for efficient, recursive processing—see Algorithm Pseudocode 4 for an example of such an approach. First, we select a sequence of admissible radii $r_1 < r_2 < r_3$ to define concentric shells/layers (e.g., the region between r_1 and r_2 forms one layer, the region between r_2 and r_3 forms the next, etc.), where each shell inherits the full six angular sectors. Within each layer, we begin at the root with all six S_t, then apply the ternary split to branch into two interleaved groups (e.g., $\{S_0, S_2, S_4\}$ and $\{S_1, S_3, S_5\}$), followed by binary pairing at the leaves (e.g., we match the opposites within each group and adjust for the subgroup action). Intriguingly, the circular inversion ι_r allows us to extend the tree structure symmetrically inward, where (results from) the outer layers are mapped directly to the inner layers without recomputation. This carefully devised arrangement supports divide-and-conquer algorithms, such as distributing ternary groups across processors for load balancing or three-way switching, then recursing for a binary split within subgroups. It unlocks scalable traversals or optimizations in symmetric lattice graphs.*

ALGORITHM PSEUDOCODE 4. Simplified Symmetric Processing in the Tri-Quarter Framework: Symmetry-Reduced Computation via Rotational and Inversive Bijections for Angular and Radial Mirroring

```
function SimplifiedSymmetricProcessing(graph, base_sector, rotation_map, inversion_map):
    # Step 1: Compute fully on base sector (leaf-level work)
    # Task-specific application e.g., sum(node.value for all nodes in base_sector)
    base_result = ComputeOnSector(base_sector)

    # Step 2: Mirror base to other sectors via rotational bijections
    # (exploit Z_6 symmetry with no recompute)
    outer_results = {base_sector: base_result}  # Start with base
    for sector in other_sectors:  # Cycle through remaining 5 sectors
        # Mirror via rotation (preserves values exactly under bijection)
        mirrored = MirrorSector(base_result, rotation_map, from=base_sector, to=sector)
        outer_results[sector] = mirrored

    # Step 3: Aggregate outer zone results
    # (e.g., sum or merge dictionaries across sectors, union lookups, etc.)
    outer_total = CombineResults(outer_results)

    # Step 4: Mirror outer to inner via radial inversion bijection
    # (exact, no recompute)
    inner_total = MirrorResult(outer_total, inversion_map)  # Preserve values via zone swap

    # Step 5: Return combined dual-zone aggregate (e.g., for zone-invariant metrics)
    return CombineResults(outer_total, inner_total)  # Final global sum or merged output

# Helper: Mirror sector via rotation (bijection preserves values)
function MirrorSector(sector_result, rotation_map, from_sector, to_sector):
    mirrored = 0  # Or {} for dictionary-based results
    for node, value in sector_result.items():
```

```
        rot_node = rotation_map.get(node)  # Bijection via rotation (e.g., pi/3 shift)
        if rot_node: mirrored += value  # Or mirrored[rot_node] = value
    return mirrored

# Helper: Mirror outer to inner via inversion (bijection preserves values)
function MirrorResult(outer_result, inversion_map):
    inner_result = 0  # Or {} for dictionary-based results
    for node, value in outer_result.items():
        inv_node = inversion_map.get(node)
        if inv_node: inner_result += value  # Or inner_result[inv_node] = value
    return inner_result

# Helper: Leaf-level computation (task-specific -- we can replace as needed)
function ComputeOnSector(sector):
    # Example application: sum -- could be BFS(G_sector, start) distances
    return sum(node.value for node in sector.nodes)
```

$\Lambda_{T,r}$ forms a D_6-invariant symmetric set that partitions Λ_r into $\Lambda_{-,r}$ and $\Lambda_{+,r}$—like a fence separating two territories along a symmetric perimeter. For the smallest admissible $r = 1$, it has size $|\Lambda_{T,1}| = 6$ and forms a cycle. As r increases, the vertex count $|\Lambda_{T,r}|$ tends to grow (though sporadically, as multiples of six), driven by additional Eisenstein integer representations of $r^2 = N$ as $a^2 + ab + b^2$ (e.g., 12 vertices for $N = 7$, 18 for $N = 49$). Even with denser $\Lambda_{T,r}$ at larger r, the set remains small relative to the $O(r^2)$ scale of the surrounding zones, which cleanly divides $\Lambda_{-,r}$ from $\Lambda_{+,r}$. Thus, Λ_r is well-suited for min-cut algorithms like Karger's [66], where symmetry and boundary size far below the $O(r^2)$ area enable efficient probabilistic cuts through repeated runs—boosting applications in practices such as network optimization [67] and clustering [68].

Corollary 4.10 (D_6-Invariant Symmetric Vertex Separator). The boundary zone subgraph $\Lambda_{T,r}$ (induced on the symmetric vertex set $V_{T,r}$ for admissible $r > 0$) forms a vertex separator between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ that is invariant under the action of D_6 (the point symmetry group of the base triangular lattice L). This structure fits min-cut algorithms [66] because its symmetry enables efficient probabilistic cuts with expected time scaling as $O(|\Lambda_{T,r}|)$.

Remark 4.11. *For the minimal admissible case $r = 1$, $V_{T,1}$ is a single D_6-orbit of size 6, so it is the smallest D_6-invariant vertex separator between $\Lambda_{-,1}$ and $\Lambda_{+,1}$ of this form. Smaller (non-D_6-invariant) separators may exist combinatorially, but $\Lambda_{T,r}$ is the canonical D_6-equivariant choice that supports the symmetry-reduced cut and flow algorithms referenced in Corollary 4.10.*

Assigning a vertex $\vec{v}_i \in \Lambda_r$ to a zone is as simple as checking its norm $||\vec{v}_i||$ against r, like sorting items by size with one measurement. Moreover, if we precompute the vertex phases mod 6, then we achieve even faster angular sector classification and thus more efficient processing of larger truncated Λ_r^R (with larger R).

With this combinatorial duality established, we now extend it to incorporate reflective symmetry through circle inversion on Λ_r. This builds on TQF's foundational separation and directional labeling mechanisms to achieve exact zone swapping while preserving key properties.

4.2 Escher Reflective Duality on Λ_r

Building on the radial separation provided by combinatorial duality, we formalize the Escher reflective duality to enable exact bijective zone swapping.

We extend the circle inversion map ι_r from the continuous X of Subsection 2.4 to the discrete Λ_r as follows:

Definition 4.12 (Circle Inversion Map on Λ_r). For any admissible $r > 0$, the *circle inversion map* $\iota_r : \Lambda_r \to \Lambda_r$ is defined as [15]

$$\iota_r(\vec{v}_i) := \frac{r^2 \vec{v}_i}{||\vec{v}_i||^2}. \tag{6}$$

Remark 4.13. *The inverted images land exactly on discrete vertices (which are distinct for admissible r, where they accumulate near the origin for inner mappings to $\Lambda_{-,r}$).*

Definition 4.14 (Escher Reflective Duality on Λ_r). Let Λ_r be the radial dual triangular lattice graph on $X = \mathbb{C} \setminus \{(0,0)_C\}$. For any admissible $r > 0$ such that $\Lambda_{T,r}$ is induced on a symmetric set $V_{T,r}$, define $\Lambda_{-,r}$, $\Lambda_{T,r}$, and $\Lambda_{+,r}$ as above, where $\Lambda_{T,r}$ is combinatorially dual to $\Lambda_{-,r}$ and $\Lambda_{+,r}$ as in Proposition 4.2. Then $\Lambda_{T,r}$ exhibits *Escher reflective duality* between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ if there exists a map ι_r (inducing a graph isomorphism between the zones) that satisfies the following conditions, analogous to the continuous case in Subsection 2.4:

(1) $\iota_r(\vec{v}_i) = \vec{v}_i$ for all $\vec{v}_i \in \Lambda_{T,r}$, which implies that $\Lambda_{T,r}$ is fixed under inversion.

(2) $\iota_r(\Lambda_{-,r}) = \Lambda_{+,r}$ and $\iota_r(\Lambda_{+,r}) = \Lambda_{-,r}$, which implies that the $\Lambda_{-,r}$ and $\Lambda_{+,r}$ are swapped.

(3) $\phi(\iota_r(\vec{v}_i)) = \phi(\vec{v}_i)$ for all $\vec{v}_i \in \Lambda_r$, which implies that the phase pairs are preserved to maintain directional consistency.

(4) $\iota_r \circ \iota_r = \mathrm{id}$, which implies that the map is its own inverse for reversible transformations.

Proposition 4.15 (Escher Reflective Duality on Λ_r). Let Λ_r be the radial dual triangular lattice graph on $X = \mathbb{C} \setminus \{(0,0)_C\}$. For any admissible $r > 0$, we define $\Lambda_{-,r}$, $\Lambda_{T,r}$, $\Lambda_{+,r}$ as above, where $\Lambda_{T,r}$ is combinatorially dual as in Proposition 4.2. Then the circle inversion map ι_r establishes Escher reflective duality across $\Lambda_{T,r}$ between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ by satisfying the conditions of Definition 4.14.

Proof. Let Λ_r be the radial dual triangular lattice graph on $X = \mathbb{C} \setminus \{(0,0)_C\}$. For any admissible $r > 0$, we define $\Lambda_{-,r}$, $\Lambda_{T,r}$, $\Lambda_{+,r}$ as above, where $\Lambda_{T,r}$ is combinatorially dual as in Proposition 4.2. We proceed in steps to verify each condition of Definition 4.14:

- **Step 1: Verify Fixed Boundary under Inversion.** For $\vec{v}_i \in \Lambda_{T,r}$, $||\vec{v}_i|| = r$, so $\iota_r(\vec{v}_i) = \frac{r^2 \vec{v}_i}{r^2} = \vec{v}_i$, which fixes the boundary vertices as required, analogous to the continuous case on T_r in Subsection 2.4. So condition (1) of reflective duality holds. ☑
- **Step 2: Verify Zone Swapping and Bijectivity.** The bijectivity between $\Lambda_{+,r}$ and $\Lambda_{-,r}$ follows from the radial dual construction in Subsection 3.3, where $\Lambda_{-,r} = \iota_r(\Lambda_{+,r})$ and vice versa by involution, which ensures exact swaps without overlaps or gaps for admissible r. So condition (2) of reflective duality holds. ☑
- **Step 3: Verify Phase Pair Preservation.** Phase pair preservation holds across Λ_r under ι_r, as the real scalar $\frac{r^2}{||\vec{v}_i||^2} > 0$ scales the magnitude but maintains the direction, so $\langle \iota_r(\vec{v}_i) \rangle = \langle \vec{v}_i \rangle$ and $\phi(\iota_r(\vec{v}_i)) = \phi(\vec{v}_i)$, just as in the continuous case on X. So condition (3) of reflective duality holds. ☑
- **Step 4: Verify Involution Property.** The involution holds algebraically: $\iota_r(\iota_r(\vec{v}_i)) = \vec{v}_i$ for vertices in Λ_r, as derived in the continuous case on X and preserved discretely due to Λ_r's rational coordinates and exact mappings. So condition (4) of reflective duality holds. ☑

The bijectivity holds for all such $\Lambda_{T,r}$, where ι_r maps graph rays proportionally while preserving angular sector alignments. Thus, ι_r establishes reflective duality across $\Lambda_{T,r}$ between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ [69]. □

Definition 4.16 (Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group). The TQF's *Inversive Hexagonal Dihedral Symmetry Group* $\mathbb{T}_{24}$ is the order-24 direct product $D_6 \times \mathbb{Z}_2$, where D_6 is the order-12 dihedral group that captures the Λ_r's order-6 rotations and reflections, and $\mathbb{Z}_2 = \mathrm{gen}(\iota_r)$ is the order-2 group generated by the circle inversion ι_r. Here, ι_r commutes with every element of D_6 ($\iota_r \circ \mathcal{R}_\theta = \mathcal{R}_\theta \circ \iota_r$ for all rotations, and $\iota_r \circ \sigma = \sigma \circ \iota_r$ for all reflections $\sigma \in D_6$), because ι_r preserves phases exactly ($\langle \iota_r(\vec{v}_i) \rangle = \langle \vec{v}_i \rangle$, per Proposition 4.15) and acts only on radial magnitudes. Thus ι_r lies in the center of $\mathbb{T}_{24}$, and $\mathbb{T}_{24}$ is a direct product with trivial cross-action between its factors.

Remark 4.17 (Relation to the Centrosymmetric Hexagonal Point Group). *As an* abstract *group, $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$ is not new. It is the centrosymmetric hexagonal point group that is cataloged in crystallography as $6/mmm$ in Hermann–Mauguin notation and D_{6h} in Schoenflies notation—one of the 32 crystallographic point groups [14, 70]. Taking the direct product of a point group with the order-2 group generated by a central involution is the standard way to build the centrosymmetric point groups from their rotation/reflection subgroups [14], and $D_6 \times \mathbb{Z}_2$ is the hexagonal case. We note this so that the abstract isomorphism type is not mistaken for a contribution of this work. What is specific to the TQF is the realization. In the crystallographic D_{6h}, the central $\mathbb{Z}_2$ factor comes from a spatial symmetry—a horizontal mirror or a 3D point inversion. Here it is instead generated by the circle inversion ι_r, a nonlinear, radial, geometrically non-spatial involution acting on Λ_r. The contribution is this action of $\mathbb{T}_{24}$ on Λ_r and its role as the symmetry group that governs the dualities and equivariant encodings developed below. The name "Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group" refers to that realized object, not to the abstract group, which already carries the standard names above.*

Remark 4.18. *The centrality of ι_r is a structural feature rather than a limitation. Because the $\mathbb{Z}_2 = \mathrm{gen}(\iota_r)$ factor commutes with all spatial symmetries in D_6, the inversive duality acts as a* global involution *that is decoupled from the spatial D_6-action—independent of, yet fully compatible with, the lattice's rotational and reflective symmetries. This decoupling enables factorized analyses: representations, equivariant encodings, and invariants of $\mathbb{T}_{24}$ split cleanly along the D_6 and $\mathbb{Z}_2$ factors, which simplifies invariance proofs (each factor verified independently) and separates spatial transformations from the duality involution in downstream computations. As an abstract group, $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$ is* not *isomorphic to D_{12} (the dihedral group of order 24): although both have order 24, D_{12} contains an element of order 12 (a primitive rotation) while $\mathbb{T}_{24}$ does not, since every element of $D_6 \times \mathbb{Z}_2$ has order dividing $\mathrm{lcm}(6, 2) = 6$. Equivalently, since $|D_6|/2 = 6$ and D_6 admits the decomposition $D_6 \cong D_3 \times \mathbb{Z}_2$ (generated by the central half-turn $\mathcal{R}_\pi$), we have $\mathbb{T}_{24} \cong D_3 \times \mathbb{Z}_2 \times \mathbb{Z}_2$. Because ι_r is central and acts by swapping the two zones while fixing $V_{T,r}$ pointwise, it induces a canonical $\mathbb{Z}_2$-grading $V(\Lambda_r) \setminus V_{T,r} = V_{+,r} \sqcup V_{-,r}$, with $V_{T,r}$ as the ι_r-fixed locus (the grading-trivial subset). And because ι_r is an involution ($\iota_r^2 = \mathrm{id}$), it splits every $\mathbb{T}_{24}$-equivariant function f on Λ_r into an ι_r-even (symmetric) part and an ι_r-odd (antisymmetric) part, just as an ordinary function decomposes into even and odd components: $f = f_+ + f_-$ with $f_\pm = \frac{1}{2}(f \pm f \circ \iota_r)$. This decomposition is canonical whenever 2 is invertible in the codomain—in particular for any characteristic-zero field, such as $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$-vector spaces. Over $\mathbb{Z}_2$, where $2 = 0$ and this averaging is unavailable, the even/odd split breaks down. The parity content that remains is instead tracked by the Tate cohomology of the order-2 group $\langle \iota_r \rangle \cong \mathbb{Z}_2$ (see Remark 5.30).*

Corollary 4.19 (Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group). The circle inversion ι_r extends Λ_r's symmetry group from D_6 to the TQF's Inversive Hexagonal Dihedral Symmetry Group $\mathbb{T}_{24}$. This full order-24 group harnesses the Escher reflective duality while maintaining bijectivity and phase pair assignments across transformations.

Remark 4.20. *$\mathbb{T}_{24}$ is useful for proving the invariance of properties (e.g., encodings or path lengths) under the complete set of symmetries, which unleashes the potential for algorithmic efficiency. For instance, in parallel traversals across Λ_r, the additional inversion symmetry allows load-balanced processing across the dual zones with reduced synchronization because computations in one zone can be directly mapped to the other without recomputation (as benchmarked in the simulations of Section 6 with ∼2x speedups).*

For larger admissible r with $|\Lambda_{T,r}| = 6k$ and $k > 1$, $\mathbb{T}_{24}$ scales invariantly, as ι_r normalizes boundary orbits regardless of k, which preserves bijectivity and phase pair constancy.

Building on the Escher reflective duality established via ι_r (Subsection 4.2), we derive further corollaries that highlight symmetric isomorphisms and reversible operations.

In the realm of graphs, an isomorphism between two graphs implies that their structure matches precisely—that is, their vertices connect identically, like twins. Here, the circle inversion property of ι_r generates such "reflective twins" between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ by keeping their connections and phase pair assignments intact over zone swapping. This drives operations that "flip" data/images symmetrically without information loss.

Corollary 4.21 (Transported Edge Graph Isomorphism Induced by Escher Reflective Duality). The Escher reflective duality induces a transported edge graph isomorphism (Definition 3.25) between $\Lambda_{+,r}$ and $\Lambda_{-,r}$ via ι_r (and vice versa by involution)—which preserves adjacency and phase pair assignments—and enables reversible graph transformations with $O(1)$ per-vertex reversal.

This isomorphism provides a structured way to mirror graph properties, which can support efficient data replication and symmetry checks in computational tasks. For instance, as demonstrated in the upcoming inversion-based path mirroring simulations in Section 6, this isomorphism promotes 1.6–1.8x speedups by mapping computations in $\Lambda_{+,r}$ directly to $\Lambda_{-,r}$ via the bijection ι_r, which thereby avoids redundant traversals in symmetry-aware applications such as robotic path planning or multi-agent coordination on Λ_r. This offers potential benefits for algorithm design in network analysis, where the preserved adjacency and phase pair assignments facilitate load-balanced parallel processing across zones, which reduce synchronization overhead in distributed systems.

In this reflective duality, $\Lambda_{T,r}$ consists entirely of fixed vertices, which remain anchored as a stable "backbone" under swap operations. This is handy for anchor-based algorithms, where the vertices of $\Lambda_{T,r}$ serve as consistent, reliable reference points across reversible data transformations or symmetry detection in networks. For example, in min-cut computations (as noted in Corollary 4.10 to Proposition 4.2), the fixed separator $\Lambda_{T,r}$ fosters efficient probabilistic cuts—scaling as $O(|\Lambda_{T,r}|)$—to support flow optimization in symmetric networks or clustering tasks without recomputing zone-internal structures under ι_r.

Corollary 4.22 (Anchored Boundary Vertex Invariance). The Escher reflective duality ensures all vertices in $\Lambda_{T,r}$ remain fixed under ι_r, with phase pair invariance, which enables $\Lambda_{T,r}$ to serve as a stable reference for symmetric graph operations like anchor-based traversals, where these fixed boundary vertices act as starting points for searches by leveraging invariance for efficient exploration.

Remark 4.23. *The ι_r-fixed locus $V_{T,r}$ has the structure of an order-$6k$ vertex set (Definition 3.14) that carries an unbroken D_6-action (since ι_r acts trivially on it and D_6 acts as the full point symmetry group of L on it). Equivalently, $V_{T,r}$ is the unique maximal $\mathbb{T}_{24}$-invariant subset of $V(\Lambda_r)$ on which the central $\mathbb{Z}_2 = \langle \iota_r \rangle$ factor degenerates—i.e., it is the "diagonal" of the ι_r-induced $\mathbb{Z}_2$-grading $V(\Lambda_r) \setminus V_{T,r} = V_{+,r} \sqcup V_{-,r}$ described in the remark following Definition 4.16. This categorical role makes $\Lambda_{T,r}$ the natural locus for any algorithm or labeling that needs to be invariant under the full $\mathbb{T}_{24}$-action (not merely D_6-equivariant), since the $\mathbb{Z}_2$ factor contributes no nontrivial action there.*

The anchored-vertex invariance of $\Lambda_{T,r}$ secures a trustworthy, dependable foundation upon which to facilitate graph operations on Λ_r, armed with the potential to streamline graph traversals and improve reliability in symmetric computational frameworks.

Reversible means you can *undo* the swap exactly—like a graph permutation that's its own inverse (an involution). Here, the inversion "toggles" $\Lambda_{+,r}$ and $\Lambda_{-,r}$ back and forth—without information loss, while always preserving the structure via the induced graph isomorphism. This supports undoable operations in data structures or error-correcting codes (e.g., in symmetric or lattice-based schemes), where states toggle symmetrically without recomputation.

Corollary 4.24 (Reversible Zone Swapping). The involution property of ι_r enables exact, bijective swapping of $\Lambda_{-,r}$ and $\Lambda_{+,r}$, which preserves vertex adjacency *(via the induced graph isomorphism from Corollary 4.21)*, phase pairs assignments, and angular sector indices. Reversible lattice graph transformations with $O(1)$ per-vertex reversal holds.

This reversible swapping allows for flexible zone interchanges, which can advance in the development of adaptive algorithms—where transformations are easily reversed—to contribute to more robust data handling in dynamic systems.

Having established this Escher-inspired radial, nonlinear reflective duality via circle inversion, we now extend the TQF to encompass the linear mirror symmetries of D_6 to achieve paired decompositions that complement the bijective zone swaps for finer-grained algorithmic applications.

4.3 Extended Linear Mirror-Symmetric Decompositions

While the Escher reflective duality centers on nonlinear, radial swapping via circle inversion, it naturally extends through $\mathbb{T}_{24}$ of Definition 4.16 and Corollary 4.19 to incorporate the linear reflections of D_6. These linear mirror symmetries—via the $\mathbb{Z}_2$ subgroup actions of Corollary 4.25—decompose Λ_r into paired angular sectors, which complement the bijective zone swaps by promoting refined algorithmic applications—such as balanced traversals and matchings—that preserve the overall duality.

Linear mirror symmetry splits Λ_r into halves that match under reflection—like folding a symmetric network. The subgroups (e.g., reflections as $\mathbb{Z}_2$) create paired subsets, which are useful for mirrored data replication or balanced binary trees, where each half can be processed independently (on separate threads, processors, or nodes) and then recombined to finalize the results of a symmetry-aware parallelizable task.

Corollary 4.25 (Linear Mirror-Symmetric Decompositions). The reflection symmetries in D_6 (which contains $\mathbb{Z}_6$) decompose the graph rays and their assigned phase pairs into mirror-paired classes. Specifically, each reflection σ generates a $\mathbb{Z}_2$ action whose specific sector pairing depends on the reflection axis (e.g., reflection across the real axis pairs $\{S_t, S_{5-t \bmod 6}\}$, giving $\{S_0, S_5\}, \{S_1, S_4\}, \{S_2, S_3\}$), while the central rotation $\mathcal{R}_\pi \in \mathbb{Z}_6$ pairs $\{S_t, S_{t+3 \bmod 6}\}$. These pairings map primary graph rays and their assigned phase pairs to their symmetric counterparts while preserving the combinatorial duality (Proposition 4.2) and Escher reflective duality (Proposition 4.15). This decomposition enables binary-symmetric lattice graph algorithms, such as reflected traversals (where a traversal in one paired class is mirrored to the other without recomputation) or paired matchings (e.g., constructing auxiliary bipartite graphs from the paired orbits for efficient perfect matchings or load-balanced parallel processing).

Mirror-symmetric decompositions enable paired analyses that can simplify binary operations and serve as a useful tool for balanced processing in graph-based computations. For instance, the following example demonstrates how to apply this decomposition to construct paired matchings under a specific reflection.

Example 4.26 (Paired Matchings via Real Axis Linear Mirror-Symmetric Decomposition). Let's consider the truncated radial dual triangular lattice graph Λ_r^R (with admissible inversion radius $r = 1$ and truncation radius $R = 4$) as visualized in Figure 3). We select the reflection σ across the real axis (the primary graph ray at phases 0 and π, per Table 5), which induces a $\mathbb{Z}_2$ action that pairs the angular sectors as $\{S_0, S_5\}$, $\{S_1, S_4\}$, and $\{S_2, S_3\}$ (per Corollary 4.25).
The orbits under σ consist of:

- Fixed vertices: Each vertex $\vec{v}_i$ on the real axis graph ray (e.g., outer vertices like $(2, 0)_C$ at phase 0 or $(-2, 0)_C$ at phase π, and their inverted inner twins), which lie on the reflection axis and satisfy $\sigma(\vec{v}_i) = \vec{v}_i$.
- Paired orbits: For each vertex $\vec{v}_i$ off the axis (e.g., in S_0), the pair $\{\vec{v}_i, \sigma(\vec{v}_i)\}$ maps to the symmetric vertex in the paired angular sector (e.g., S_5), which transforms the assigned phase pairs consistently with the linear reflection symmetry while preserving the combinatorial duality (Proposition 4.2) and Escher reflective duality (Proposition 4.15).

We construct the auxiliary bipartite graph B for paired matchings as follows (where A and A' form the two parts of a bipartition, like "upper" and "lower" halves reflected across the real axis, while B connects them with matching edges to exploit mirror symmetry for algorithms):

(1) First, we define the bipartition sets A as the union of vertices in angular sectors $S_0 \cup S_1 \cup S_2$ and A' as their images under the reflection σ, namely $S_5 \cup S_4 \cup S_3$.

(2) Second, for each paired orbit $\{\vec{v}_i, \sigma(\vec{v}_i)\}$ under the reflection σ, we add the edge $\{\vec{v}_i, \sigma(\vec{v}_i)\}$ to B (with $\vec{v}_i \in A$ and $\sigma(\vec{v}_i) \in A'$) to form a matching that perfectly covers all non-fixed vertices. Fixed vertices (those that are invariant under σ) remain unmatched and thus may be treated as isolated vertices in B (added to either A or A' as appropriate) to preserve bipartiteness without self-loops.

(3) Third (and optionally), we may include the original edges from Λ_r within A and within A' separately (as disjoint components) to support specific algorithms while avoiding cross-pair edges and preserving bipartiteness.

The auxiliary bipartite graph B enables efficient perfect matchings (e.g., via the Hopcroft–Karp algorithm [71] in $O(\sqrt{|V|}|E|)$ time) or load-balanced parallel processing (by assigning A and A' to separate threads and synchronizing only at matched edges). For instance, a BFS traversal in A can be reflected to A' without recomputation via the linear mirror symmetry (Corollary 4.25) to reduce the computational effort by approximately 50% in symmetric cases because our construction preserves zone separations, graph rays, and equivariant encodings (in the upcoming Section 5).

These linear mirror-symmetric decompositions extend the utility of the Escher reflective duality by providing tools for binary partitioning, which integrate seamlessly with the bijective self-duality and dual metrics that we explore next.

4.4 Dual Metrics

With Escher reflective duality (and the extended linear mirror-symmetric decompositions) in place, we now pivot to achieve bijective self-duality under inversion. Given any admissible $r > 0$ and for any $\vec{v}_i \in \Lambda_r$, the Euclidean norm satisfies the relation

$$||\vec{v}_i|| = \frac{r^2}{||\iota_r(\vec{v}_i)||},$$

which enables the bijective mapping and preserves the properties of the TQF such as phase pair assignments, angular sector memberships, and graph isomorphism between zones.

This radial focus (from the origin) is intentional for the framework's emphasis on central symmetry and inversion-based duality. For example, it enables exact zone swaps and scale-invariant computations along graph rays. In other words, $\Lambda_{-,r}$ mirrors $\Lambda_{+,r}$ exactly under circular inversion. It preserves distances via the product r^2 (a constant like a seesaw where one side's increase balances the other side's decrease). To ensure exact preservation under inversion while maximizing TQF's utility, we define dual metrics that handle both discrete and continuous distances bijectively—with no approximations. These enable exact distance preservation under inversion, which aligns with our bijective requirements.

We start with the discrete dual metric, which aligns naturally with the graph-theoretic dualities from prior subsections. The full metric between any two vertices $\vec{v}_i$ and $\vec{v}_j$ in the same zone can be extended to graph distance (shortest-path hops in the induced subgraph for a purely combinatorial zone-internal metric). It is not defined across different zones. The zones are disjoint subgraphs separated by the boundary. Cross-zone distances would require additional bridging (e.g., via boundary edges). This is outside the current scope of radial duality. For the sake of simplicity and exactness, our paper keeps it zone-internal. This connects directly to the norm.

Definition 4.27 (Discrete Dual Metric). The *discrete dual metric* is defined for pairs of vertices $\vec{v}_i, \vec{v}_j$ in the same zone (either both in $\Lambda_{-,r}$ or both in $\Lambda_{+,r}$) as the graph distance (shortest-path distance) $d_{hops}(\vec{v}_i, \vec{v}_j)$ in the respective induced subgraph, i.e., the minimum number of hops (edges) along any path that connects vertices $\vec{v}_i$ and $\vec{v}_j$ (that satisfy the standard metric axioms: non-negativity, symmetry, and

triangle inequality, where we assume that the induced subgraph is connected). Any cross-zone extensions will require additional mapping.

Lemma 4.28 (Zone Subgraph Connectivity). For every admissible inversion radius $r > 0$, the induced subgraphs $\Lambda_{+,r}$ and $\Lambda_{-,r}$ are each connected, so the discrete dual metric d_{hops} of Definition 4.27 is well-defined on every pair of same-zone vertices.

Proof.

- **Step 1: Connectivity of $\Lambda_{+,r}$.** Let $\vec{v}_i, \vec{v}_j \in V_{+,r}$. Choose any integer $K > \max(\|\vec{v}_i\|, \|\vec{v}_j\|, 2r/\sqrt{3})$ (so in particular $K > r$, and the inscribed-circle radius $K\sqrt{3}/2$ of the hexagonal ring at radius K strictly exceeds r), and consider the hexagonal ring at radius K in L formed by the six corner vertices $K\vec{e}_0, K\vec{e}_1, \ldots, K\vec{e}_5$ (where $\vec{e}_k = \mathcal{R}_{k\pi/3}(\vec{\omega}_0)$, as in Definition 3.39) joined by sequences of unit-length lattice edges along each side; concretely, the side from $K\vec{e}_k$ to $K\vec{e}_{k+1 \bmod 6}$ consists of the $K+1$ lattice points $K\vec{e}_k + l(\vec{e}_{k+1 \bmod 6} - \vec{e}_k)$ for $l = 0, 1, \ldots, K$, with consecutive points differing by the unit primitive direction vector $\vec{e}_{k+1 \bmod 6} - \vec{e}_k$ (also in $\{\vec{e}_0, \ldots, \vec{e}_5\}$ by the closure of the primitive direction set under D_6). Every point on this hexagonal ring lies on the boundary of the regular hexagon inscribing the circle of radius $K\sqrt{3}/2$ about the origin, so it has Euclidean norm at least $K\sqrt{3}/2 > r$ by the choice of K; hence the entire ring lies in $V_{+,r}$ and is internally connected via its constituent unit edges. It suffices to show that any $\vec{v} \in V_{+,r}$ is connected to this outer ring by a path entirely within $V_{+,r}$. Writing $\vec{v} = a\vec{\omega}_0 + b\vec{\omega}_1$ for $(a, b) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$, identify the angular sector S_t containing $\vec{v}$ via the index $t = \lfloor 6\langle\vec{v}\rangle/(2\pi) \rfloor \bmod 6$ (Definition 3.9), and select the primitive direction vector $\vec{p}_t \in \{\vec{e}_0, \ldots, \vec{e}_5\}$ whose phase lies in S_t (so $\langle\vec{p}_t\rangle \in [t\pi/3, (t+1)\pi/3)$); concretely, $\vec{p}_t = \vec{e}_t$. Consider the lattice ray $\vec{v}, \vec{v} + \vec{p}_t, \vec{v} + 2\vec{p}_t, \ldots$: each step is a unit lattice edge (so consecutive vertices are adjacent in $\Lambda_{+,r}$), and the squared norm $\|\vec{v} + k\vec{p}_t\|^2 = \|\vec{v}\|^2 + 2k\langle\vec{v}, \vec{p}_t\rangle + k^2$ is monotonically increasing for $k \geq 0$ whenever $\langle\vec{v}, \vec{p}_t\rangle \geq 0$ (which holds since $\vec{v}$ and $\vec{p}_t$ both have phases in S_t, giving an angular separation strictly less than $\pi/3 < \pi/2$ and hence a non-negative Euclidean inner product). Therefore every vertex on the ray satisfies $\|\vec{v} + k\vec{p}_t\| \geq \|\vec{v}\| > r$ and lies in $V_{+,r}$. Continue along this ray until reaching a vertex with norm at least K. Given the adjacency along the ring (which contains all six corner vertices $K\vec{e}_k$ and their connecting paths), the reached vertex is thus connected within $V_{+,r}$ to any corner $K\vec{e}_t$ via at most one ring side. Applying this construction to both $\vec{v}_i$ and $\vec{v}_j$ and joining the two terminal vertices via the ring yields a path entirely within $V_{+,r}$. Thus $\Lambda_{+,r}$ is connected. ☑
- **Step 2: Connectivity of $\Lambda_{-,r}$ by Transport.** The edges of $\Lambda_{-,r}$ are defined by transport along ι_r from edges of $\Lambda_{+,r}$ (Subsection 3.3), so the map $\iota_r : \Lambda_{+,r} \to \Lambda_{-,r}$ is a graph isomorphism (see Proposition 4.15). Therefore, connectivity of $\Lambda_{+,r}$ implies connectivity of $\Lambda_{-,r}$. ☑

□

Corollary 4.29 (Self-Duality of the Discrete Dual Metric). The discrete dual metric of Definition 4.27 is *self-dual* under the circular inversion map ι_r as

$$d_{hops}(\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)) = d_{hops}(\vec{v}_i, \vec{v}_j) \tag{7}$$

exactly, which is preserved by the graph isomorphism (Corollary 4.21 to Proposition 4.38), for admissible $r > 0$.

Remark 4.30. *The discrete dual metric d_{hops} is defined solely for zone-specific distances in the induced subgraphs $\Lambda_{-,r}$ and $\Lambda_{+,r}$, where self-duality holds via the isomorphism induced by reflective duality (Proposition 4.15). For distances in the full graph Λ_r that include cross-zone paths across $\Lambda_{T,r}$'s twin edges, self-duality does not apply without extensions because these connections introduce radial asymmetries that are not preserved under circle inversion. Applications that require global metrics may be extended via dual*

boundary mappings (e.g., weighting twins by inverse norms), noting that bijectivity remains zone-specific per Proposition 4.38.

Building on this discrete foundation, we next introduce the continuous dual metric, which embeds each zone subgraph isometrically in the hyperbolic plane $\mathbb{H}^2$ for exact zone-internal invariance under inversion, with ι_r acting as the canonical isometry between the two copies. To achieve a metric on which circle inversion acts as an isometry, we represent each zone separately in $\mathbb{H}^2$ in geodesic polar coordinates about the boundary circle T_r, where ι_r takes the form of an orientation-preserving isometry that exchanges the two zone embeddings. For each vertex $\vec{v}_i \in \Lambda_r$ (in either zone or on the boundary) we define the (signed) *boundary-relative depth*

$$\tilde{u}(\vec{v}_i) := \log\left(\frac{\|\vec{v}_i\|}{r}\right), \tag{8}$$

which satisfies $\tilde{u} = 0$ exactly on T_r, is strictly positive on $\Lambda_{+,r}$, strictly negative on $\Lambda_{-,r}$, and transforms as $\tilde{u} \mapsto -\tilde{u}$ under ι_r. The intrinsic (unsigned) depth used for the hyperbolic embedding is then

$$u(\vec{v}_i) := |\tilde{u}(\vec{v}_i)| = \left|\log\left(\frac{\|\vec{v}_i\|}{r}\right)\right| \geq 0, \tag{9}$$

together with the phase $\langle \vec{v}_i \rangle \in [0, 2\pi)$.

Definition 4.31 (Continuous Dual Metric). Embed each open zone into the hyperbolic plane $\mathbb{H}^2$—modeled in geodesic polar coordinates (u, θ) about a fixed basepoint, where $ds^2 = du^2 + \sinh^2 u\, d\theta^2$ is the metric tensor of constant Gaussian curvature -1 [69]—via the coordinate maps

$$\Phi_+ : V_{+,r} \to \mathbb{H}^2, \qquad \Phi_- : V_{-,r} \to \mathbb{H}^2, \qquad \Phi_\pm(\vec{v}_i) := \big(u(\vec{v}_i),\, \langle \vec{v}_i \rangle\big),$$

where $u(\vec{v}_i)$ is the non-negative boundary-relative depth of Equation 9 and $\langle \vec{v}_i \rangle \in [0, 2\pi)$ is the phase. For two vertices $\vec{v}_i, \vec{v}_j$ in the same open zone (either both in $\Lambda_{+,r}$ or both in $\Lambda_{-,r}$), the *continuous dual metric* $d_{\mathbb{H}}(\vec{v}_i, \vec{v}_j)$ is the pullback under $\Phi_\pm$ of the $\mathbb{H}^2$ geodesic distance, given in closed form by the hyperbolic law of cosines

$$d_{\mathbb{H}}(\vec{v}_i, \vec{v}_j) := \operatorname{acosh}\left(\cosh u(\vec{v}_i) \cosh u(\vec{v}_j) - \sinh u(\vec{v}_i) \sinh u(\vec{v}_j) \cos \delta\theta(\vec{v}_i, \vec{v}_j)\right), \tag{10}$$

where the angular separation is

$$\delta\theta(\vec{v}_i, \vec{v}_j) := \min\left(|\langle \vec{v}_i \rangle - \langle \vec{v}_j \rangle|,\ 2\pi - |\langle \vec{v}_i \rangle - \langle \vec{v}_j \rangle|\right). \tag{11}$$

On each open zone $\Phi_\pm$ is injective—because $u > 0$ strictly there (Equation 9) and the phase $\langle \vec{v}_i \rangle$ is well-defined—so $\Phi_\pm$ is an isometric embedding and $d_{\mathbb{H}}$ is a genuine metric on each open zone, which inherits non-negativity, symmetry, and the triangle inequality from $\mathbb{H}^2$. The two zones $\Lambda_{+,r}$ and $\Lambda_{-,r}$ each carry their own copy of this metric (denoted $d_{\mathbb{H},+}$ and $d_{\mathbb{H},-}$ when disambiguation is needed), and the two copies share the common polar coordinate origin $u = 0$ corresponding to T_r. This is the canonical zone-internal metric under which ι_r acts as a boundary fixing isometry: since $u(\iota_r(\vec{v}_i)) = u(\vec{v}_i)$ (Equation 9) and $\langle \iota_r(\vec{v}_i) \rangle = \langle \vec{v}_i \rangle$ (Proposition 4.15), the inversion fixes the coordinates (u, θ) pointwise and merely exchanges which copy of $\mathbb{H}^2$ a vertex occupies.

Remark 4.32. *In the $\mathbb{H}^2$-embedding of Definition 4.31, the $u = 0$ locus is the polar coordinate origin (a single point at which θ is undefined and where polar coordinates have their standard origin singularity), and the boundary circle $T_r \subset X$ is identified with this origin point under the embedding (so T_r collapses to the polar origin rather than embedding as a geodesic of positive circumference). Since the two zones $\Lambda_{+,r}$ and $\Lambda_{-,r}$ are each represented by their own copy of this same $\mathbb{H}^2$ embedding sharing the common polar origin $u = 0$, the circle inversion ι_r acts as the canonical isometric identification $\mathbb{H}^2_+ \to \mathbb{H}^2_-$ between the two copies, fixing the common origin pointwise (since $u(\vec{v}_i) = u(\iota_r(\vec{v}_i))$ by Equation 9 and the phases $\langle \vec{v}_i \rangle$ are preserved by Proposition 4.15). Equivalently, ι_r is the identity on the polar coordinates (u, θ) of the embedded zones and*

effects the zone swap purely at the level of which copy of $\mathbb{H}^2$ a vertex belongs to. Consequently $d_{\mathbb{H}}$ is a genuine metric on each open zone $V_{\pm,r}$ but degenerates to a pseudo-metric *on $V_{\pm,r} \cup V_{T,r}$, since all boundary vertices in $V_{T,r}$ are identified to the single polar coordinate origin $u = 0$ under this embedding (so $d_{\mathbb{H}}(\vec{v}_i, \vec{v}_j) = 0$ whenever both vertices lie in $V_{T,r}$). Hence, $d_{\mathbb{H}}$ is used in this paper only between same-open-zone vertex pairs, where the boundary vertices are excluded from its domain of applicability.*

Corollary 4.33 (Self-Duality of the Continuous Dual Metric). Proposition 4.15 implies that the continuous dual metric of Definition 4.31 is *self-dual* under the circular inversion map ι_r as

$$d_{\mathbb{H}}(\iota_r(\vec{v}_i), \iota_r(\vec{v}_j)) = d_{\mathbb{H}}(\vec{v}_i, \vec{v}_j) \tag{12}$$

for admissible $r > 0$, whenever $\vec{v}_i, \vec{v}_j$ lie in a common open zone (i.e., both in $\Lambda_{+,r}$ or both in $\Lambda_{-,r}$, excluding $\Lambda_{T,r}$, so that both $\iota_r(\vec{v}_i)$ and $\iota_r(\vec{v}_j)$ lie in the opposite open zone). Equivalently, ι_r acts trivially on the polar coordinates (u, θ) in the $\mathbb{H}^2$-embedding of either zone, so $d_{\mathbb{H}}$ depends only on those coordinates.

Remark 4.34. *The continuous dual metric provides exact, symbolically closed-form geodesic distances within each zone, with self-duality under ι_r following directly from the fact that ι_r leaves (u, θ) unchanged and only swaps which $\mathbb{H}^2$-copy contains a vertex. For radial paths (along the same ray with $\delta\theta = 0$) between vertices in the same zone, the distance reduces to*

$$d_{\mathbb{H}} = |u(\vec{v}_i) - u(\vec{v}_j)| = \left| \log\left(\frac{\|\vec{v}_i\|}{\|\vec{v}_j\|} \right) \right|. \tag{13}$$

For "same-shell" circumferential paths between vertices $\vec{v}_i$ and $\vec{v}_j$ at the same Euclidean norm (i.e., $\|\vec{v}_i\| = \|\vec{v}_j\|$ and thus $u(\vec{v}_i) = u(\vec{v}_j) = u$), the distance reduces equivalently to

$$d_{\mathbb{H}} = 2 \operatorname{asinh}\left(\sinh u \, \sin\left(\frac{\delta\theta}{2} \right) \right). \tag{14}$$

All evaluations are symbolically exact (closed-form expressions are computable via norms and phases) and don't require approximations. For implementations, use exact arithmetic (e.g., integer squared norms) to avoid floating-point error.

Remark 4.35. *For an exciting exercise, we invite the interested reader to consider any alternative or additional metrics that further enrich the structure of Λ_r and $\mathbb{T}_{24}$ for real-world applications.*

Building on the inversion-invariant dual metric, vertices can be quickly looked up using a key from their angular sector and distance, like a dictionary where flips don't change the key. This allows rapid access even after swapping zones—to drive efficient lookups in databases or caches in symmetric systems.

Corollary 4.36 (Mod 6 Sector and Radial-Bin Bijection Hashing). Vertices of Λ_r can be hashed via the pair (angular sector S_t index t mod 6, dual norm bin), where S_t is as defined in Definition 3.9 and the dual norm bin is the floor-indexed bin $\lfloor u(\vec{v}_i)/b \rfloor$, with $b > 0$ the bin width and $u(\vec{v}_i) = |\log(\|\vec{v}_i\|/r)|$ the inversion-invariant (non-negative) boundary-relative depth from Equation 9. The inversion ι_r preserves this hash (since $u(\iota_r(\vec{v}_i)) = u(\vec{v}_i)$ by construction), which facilitates dual-indexed data structures for rapid $O(1)$ lookups across zones. Note that all boundary vertices $\vec{v}_i \in \Lambda_{T,r}$ share the radial bin $\lfloor 0/b \rfloor = 0$ (since $u(\vec{v}_i) = 0$ on T_r) and are distinguished only by their angular sector index t, which is consistent with $\Lambda_{T,r}$ being the ι_r-fixed locus.

This hashing upholds the reflective duality and expedites rapid data retrieval in structures that maintain efficiency across transformations. This is useful in applications that require high-throughput access to dynamic Λ_r.

Remark 4.37. *This discretizes the radial distance in a scale-invariant manner under inversion. Logarithmic (geometric) binning of a radial coordinate is a standard technique for handling data where its density varies with the radius, such that the bins widen at larger radii to balance the per-bin counts. This is used routinely in, for example, radial-profile photometry [72, 73], and we adapt it here to the inversion invariant boundary-relative depth. Key invariants under ι_r include:*

(1) the angular sector S_t index t (preserved via phase constancy, as per Proposition 4.15),

(2) the dual norm bin $\lfloor u/b \rfloor$ (invariant under ι_r since u is the non-negative boundary-relative depth $|\log(\|\vec{v}_i\|/r)|$ from Equation 9), and

(3) the overall hash pair $(t, \lfloor u/b \rfloor)$ (a composite key combination of the above two invariants that is invariant across ι_r-twins, as per Proposition 4.38).

For implementation, use exact integer computations on squared norms to avoid floating-point error and assure reproducibility. Use empirical data to tune b to achieve balanced bin distributions.

These dual metrics align with bijective self-duality (Proposition 4.38), which yield exact, reversible computations in lattice-based models. In practice, the discrete dual metric (Definition 4.27) suits combinatorial tasks like shortest-path traversals in the upcoming simulations of Section 6, while the continuous dual metric (Definition 4.31) extends to geometric applications like Euclidean embeddings in signal processing or robotics.

Proposition 4.38 (Bijective Self-Duality). The circle inversion map ι_r is a bijection $\iota_r : \Lambda_{-,r} \to \Lambda_{+,r}$ (and vice versa by involution). It preserves phase pairs $\phi(\iota_r(\vec{v}_i)) = \phi(\vec{v}_i)$ and graph ray directions (Definition 3.31).

Proof. Consider an admissible $r > 0$ to ensure that the symmetric boundary set $\Lambda_{T,r}$ facilitates gap-free mappings. The graph isomorphism induced by the Escher reflective duality of Proposition 4.15 preserves the path lengths measured as hop counts (as per the discrete dual metric in Definition 4.27) where Λ_r's order-6 rotational symmetry bijectively maps structures. This sends the inner paths to their twin outer paths with zero deviation to ensure the bijection as follows:

- **Step 1: Setup and Preservation of Key Structures.** The circle inversion map ι_r acts on the zones via the construction $\Lambda_{-,r} = \iota_r(\Lambda_{+,r})$ to confirm that the phase pairs $\phi(\iota_r(\vec{v}_i)) = \phi(\vec{v}_i)$ and graph ray phases are preserved exactly, as established in Proposition 4.15. Since ι_r is an involution on X (it is its own inverse, and therefore bijective on its domain), distinct outer vertices in $\Lambda_{+,r}$ map to distinct image points in X, so the construction yields a well-defined bijection of vertex sets without overlaps or collisions. ☑
- **Step 2: Injectivity.** Since ι_r is an involution on X (Proposition 4.15, condition 4), it is bijective on X, hence injective on the subset $V_{+,r} \subset X$. Therefore distinct $\vec{v}_i, \vec{v}_j \in V_{+,r}$ have distinct images $\iota_r(\vec{v}_i), \iota_r(\vec{v}_j) \in V_{-,r}$, with phase preservation $\langle \iota_r(\vec{v}_i) \rangle = \langle \vec{v}_i \rangle$ guaranteed by Proposition 4.15 (condition 3). Consequently, distinct outer graph rays in $\Lambda_{+,r}$ map to distinct inner graph rays in $\Lambda_{-,r}$. ☑
- **Step 3: Surjectivity.** Surjectivity follows immediately from the definition $\Lambda_{-,r} = \iota_r(\Lambda_{+,r})$ (Definition 3.18): every inner vertex has a preimage in $\Lambda_{+,r}$ by construction. The involution property $\iota_r \circ \iota_r = \mathrm{id}$ (condition 4 of Definition 4.14) further guarantees that every inner vertex has a *unique* preimage in the outer zone. ☑
- **Step 4: No Duplicates and Exactness.** Injectivity (Step 2) and the involution property of ι_r together imply that no duplicates exist: distinct outer graph rays map to distinct inner graph rays, where the resulting rational coordinates avoid overlaps. Exactness holds because the bijection aligns the mappings to discrete vertices, which preserves combinatorial properties like hop counts (via the discrete dual metric). ☑

- **Step 5: Infinite and Finite Cases.** In the countably infinite (non-truncated) Λ_r, the bijection is exact. In finite truncated Λ_r^R, the bijection remains exact between the finite truncated zones, though these approximate the infinite zones with gaps scaling as $O(1/R^2)$ (Table 7). For example, the path mapping in Figure 3 demonstrates the bijection, and the graph ray at phase $\pi/3$ in Subsection 3.2 has its phase preserved under ι_r. ☑

□

Data on $\Lambda_{-,r}$ can be perfectly reconstructed from data on $\Lambda_{+,r}$ via ι_r. It is like undoing a transformation with zero information loss. It keeps all counts and paths intact. This is useful for making copies/backups or reversing computations in graphs.

Corollary 4.39 (Reversible Information Preservation). Given the circle inversion map $\iota_r : \Lambda_{-,r} \to \Lambda_{+,r}$ (and vice versa by involution, per the bijective self-duality in Proposition 4.38), any function $f : \Lambda_{-,r} \to \mathcal{C}$ (where $\mathcal{C}$ is any codomain set) induces a corresponding function $f' : \Lambda_{+,r} \to \mathcal{C}$ defined explicitly by $f'(\vec{v}_j) = f(\iota_r^{-1}(\vec{v}_j))$ for all $\vec{v}_j \in \Lambda_{+,r}$, where $\iota_r^{-1}(\vec{v}_j) = \vec{v}_i$ is the unique bijective twin in the inner zone $\Lambda_{-,r}$ with $\vec{v}_j = \iota_r(\vec{v}_i)$. Equivalently, since ι_r is an involution, this simplifies to $f'(\vec{v}_j) = f(\iota_r(\vec{v}_j))$ for $\vec{v}_j \in \Lambda_{+,r}$. Moreover, f can be perfectly reconstructed from f' via $f(\vec{v}_i) = f'(\iota_r(\vec{v}_i))$ for all $\vec{v}_i \in \Lambda_{-,r}$, which preserves combinatorial properties such as distances under the discrete dual metric (Definition 4.27) and the continuous dual metric (Definition 4.31), along with the upcoming equivariant encodings (Section 5).

Remark 4.40. *Data in $\Lambda_{-,r}$ can be perfectly reconstructed from data in $\Lambda_{+,r}$ (and vice versa) via the circle inversion map ι_r (which is its own inverse, as an involution per Proposition 4.15), similar to undoing a transformation without information loss while preserving vertex counts and graph ray structures (Definition 3.31). This supports applications such as state backups in iterative graph algorithms (e.g., checkpointing traversal states for fault tolerance) or reversible computations on graphs, thereby facilitating the consistent handling of data across transformations and securing integrity in computational models that involve symmetric mappings (e.g., such as symmetry-aware parallel algorithms or lattice-based cryptography).*

Corollary 4.41 (Constant Product Invariance). For any $\vec{v}_i \in \Lambda_{-,r} \cup \Lambda_{+,r}$, the Euclidean norms satisfy $\|\vec{v}_i\| \cdot \|\iota_r(\vec{v}_i)\| = r^2$, which maintains normalized radial distances (scaled relative to the boundary radius r) under bijective self-duality (Proposition 4.38) and promotes scale-invariant computations across zones. This invariance underpins the discrete and continuous dual metrics of Definitions 4.27 and 4.31, which supports computations in unweighted Λ_r where hop distances remain consistent across dual zones via the induced graph isomorphism (Corollary 4.21).

The constant product offers a useful conservation property for radial distances. It facilitates computations that remain consistent across varying scales in graph-based analyses.

As the admissible r increases, $\Lambda_{T,r}$ extends farther from the origin, thereby partitioning Λ_r such that $\Lambda_{+,r}$ encompasses more vertices while the map ι_r preserves the core duality properties—such as the exact bijections between $\Lambda_{-,r}$ and $\Lambda_{+,r}$. The symmetries of $\mathbb{T}_{24}$ that are native to TQF facilitate per-vertex adjustments—such as $O(1)$ zone reclassifications via norm trichotomy checks—which allows algorithms to adapt to these larger instances of Λ_r without full recomputation.

Corollary 4.42 (Adaptive Inversion Scaling). For an increasing sequence of admissible inversion radii r_k, the circle inversion map ι_{r_k} preserves the combinatorial duality (Proposition 4.2), Escher reflective duality (Proposition 4.15), and bijective self-duality (Proposition 4.38) across the corresponding radial dual triangular lattice graph Λ_{r_k}.

Remark 4.43. *The preservation in Corollary 4.42 enables radial scaling of ι_{r_k} to adapt the bijection between Λ_{+,r_k} and $\Lambda_{-,r_k} = \iota_{r_k}(\Lambda_{+,r_k})$ without recomputing the dualities. More specifically, for each $\vec{v}_i \in \Lambda_{+,r_k}$, the per-vertex adjustment computes the new inverted position $\iota_{r_k}(\vec{v}_i) = r_k^2 \vec{v}_i / \|\vec{v}_i\|^2$ in $O(1)$ time (by precomputing*

the squared Euclidean norm $\|\vec{v}_i\|^2$), to yield a total rescaling cost of $O(|\Lambda_{r_k}|)$ to update the bijection and induced edges across the complete $\Lambda_r = \Lambda_{-,r_k} \cup \Lambda_{T,r_k} \cup \Lambda_{+,r_k}$. This supports adaptive algorithms for dynamically expanding truncated $\Lambda_{r_k}^R$, where the truncation radius $R \gg r_k$ is adjusted "on the fly" to maintain balanced zones.

This adaptive scalability provides a flexible approach to handle expanding structures—like enlarging truncated Λ_r^R as needed without rebuilding the structure each time. This supports the design of algorithms where Λ_r^R needs to grow in steps (e.g., adaptive networks or evolving simulations). We computationally combine these dual metrics to simultaneously model the combinatorial (discrete) and geometric (continuous) aspects of distances and thereby enable practical encodings. For example, given a finite set of target vertices, as we traverse along Λ_r^R we can simultaneously model, with respect to each target vertex, the hop counts along the shortest paths and the radially symmetric potentials. All this keeps the bijective mappings intact—no approximations required—which strengthens the practical simulation capabilities on Λ_r.

Example 4.44 (Modular Hashing with the Continuous Dual Metric). For vertex $\vec{v}_i$, compute the hash $h(\vec{v}_i) = t + 6 \times \lfloor u(\vec{v}_i)/b \rfloor$, where $t = \lfloor 6\langle \vec{v}_i \rangle/(2\pi) \rfloor \bmod 6$ is the angular sector S_t index from Definition 3.9 and Lemma 3.10, $u(\vec{v}_i) = |\log(\|\vec{v}_i\|/r)|$ is the non-negative boundary-relative depth from Equation 9 (which serves as the radial coordinate of the $\mathbb{H}^2$ embedding in Definition 4.31), and $b > 0$ is the bin width. Under inversion, the S_t index t is preserved by the phase constancy in Proposition 4.15, and u is preserved by the self-duality of the continuous dual metric (Corollary 4.33), which together maintain hash consistency for data structures.

Remark 4.45. *For exact integer bins in implementations, use $\lfloor \log_2(\|\vec{v}_i\|^2/r^2) \rfloor$ for outer vertices and $\lfloor \log_2(r^2/\|\vec{v}_i\|^2) \rfloor$ for inner vertices (with $r^2 \in \mathbb{N}$ admissible per Definition 3.14) as a discrete alternative to the natural logarithm in the continuous dual metric. Base-2 enables exact computation via bit shifts or $\lfloor \log_2 n \rfloor$ for $n \in \mathbb{N}$, without floating-point. Each expression is non-negative and the two expressions agree on $\Lambda_{T,r}$ (where both evaluate to zero) and on ι_r-twins (by the constant-product invariance $\|\vec{v}_i\| \cdot \|\iota_r(\vec{v}_i)\| = r^2$), so the resulting bins are inversion-symmetric across $\Lambda_{-,r}$ and $\Lambda_{+,r}$ while tying to the self-dual continuous metric (Definition 4.31).*

ALGORITHM PSEUDOCODE 5. Invariant Hashing with Dual Metrics. We assume BoundaryRelativeDepth computes the non-negative boundary-relative depth $u(\vec{v}_i) = |\log(\|\vec{v}_i\|/r)|$ from Equation 9 using exact arithmetic (e.g., via Python's mpmath for precision, or via the integer-based alternative $\lfloor \log_2 \max(\|\vec{v}_i\|^2/r^2, r^2/\|\vec{v}_i\|^2) \rfloor$) to avoid floating-point error. This hash is invariant under circle inversion ι_r, which enables $O(1)$ rapid lookups across the dual zones.

```
function InvariantHash(vertex_v[i], b):
    sector = floor(6 * arg(vertex_v[i]) / (2 * pi)) mod 6
    dual_norm = BoundaryRelativeDepth(vertex_v[i])  # Compute |log(||vertex_v|| / r)|
    norm_bin = floor(dual_norm / b)
    return sector + (6 * norm_bin)
```

With TQF's dualities firmly established—complete with its symmetries and invertible bijective self-duality via the Escher reflective duality—we've unlocked a powerful foundation for applications like pattern recognition or path optimization. To contextualize this within the broader landscape, how does our origin-centered radial approach align with established dualities? Let's examine contrasts with key examples, such as planar or matroid dualities, to highlight where TQF distinguishes itself or addresses specific gaps.

4.5 Comparison to Related Dualities

Planar dualities—such as Whitney's approach [21]—depend on graph embeddings to establish correspondences between vertices and faces, while mapping cycles to cuts primarily for planarity testing. In Whitney's

context, the duality requires a specific planar representation to define these swaps and focuses on the topological properties inherent to the embedding. By contrast, our TQF's combinatorial duality, as formalized in Subsection 4.1, operates independently of any embedding. It emphasizes radial separation through the norm trichotomy to assure exact bijective swaps between $\Lambda_{-,r}$ and $\Lambda_{+,r}$ without relying on vertex-face or cycle-cut mappings. Our Λ_r's embedding-independent nature allows for direct application to real-world structures with lattice properties, where the origin-centered norm provides a natural partition that is analogous to dividing a symmetric space by distance thresholds. Λ_r's dualities leverage the symmetries and bijective mappings to drive advanced graph operations and analyses, as contrasted with related dualities in Table 10.

Our TQF also diverges from matroid dualities [74], which abstract linear independence and extend to orthogonal complements in vector spaces or graphic/cographic pairs in graphs. Matroids generalize combinatorial structures but they do not inherently incorporate geometric constraints like origin-centered radiality. Unlike these, our Escher reflective duality, as proven in Proposition 4.15, introduces a bijective mechanism that preserves directional labels (phase pairs) and symmetries to design reversible transformations that apply to radial geometries. The absence of such radial bijectivity in matroid dualities highlights a key gap that our approach addresses, particularly in applications requiring exact zone inversions.

Traditional dualities often "flip" graphs in a manner reminiscent of inverting a planar map, transforming internal structures outward while maintaining combinatorial invariants. TQF's approach complements this by radially mirroring the zones, similar to Escher's reflections across a spherical surface, which supports efficient operations in hub-centric networks where paths emanate symmetrically from a central point. This radial mirroring preserves Λ_r's order-6 rotational symmetry and reinforces modular decompositions that align with angular sectors for balanced computations.

Furthermore, our radial zone swaps—executed via the circle inversion map ι_r in Definition 4.12—align well with angle-preserving discrete conformal mappings [37], which also utilize circular-based patterns to maintain local angles during transformations. While conformal mappings excel in preserving angular structure without explicit radial partitioning, TQF's bijective swaps add a complementary layer for hybrid scenarios. For instance, in computational geometry tasks that simultaneously require both smooth deformations *and* symmetric radial inversions, our TQF integration could advance processes like mesh optimization or pattern recognition on lattice-based structures.

The key invariance properties that are secured under these dualities are summarized in Table 11 to exemplify their utility for algorithmic efficiency.

These dual metrics and bijective mappings—which support invariant data representations—set the stage for equivariant encodings that harness the power of symmetry to forge robust computational structures in the next section.

5 Equivariant Functions and Encodings

Building on the symmetries, dualities, and bijective mappings from previous sections, we now pivot to further upgrade TQF with equivariant functions and encodings. These are mappings that assign values or transformations to vertices so they predictably and deterministically align with the framework's symmetries—this includes rotations, reflections, and circle inversions—under the actions of TQF's core algebraic structure $\mathbb{T}_{24}$. In essence, $\mathbb{T}_{24}$ fuses Λ_r's D_6 symmetries with inversive bijections to deliver consistent, symmetry-compliant tools for computational tasks. This approach leverages the key properties of Λ_r—including its radial dual structure, phase pair constancy along graph rays, and bijective self-duality under inversion—to support efficient algorithms in areas like graph hashing, pattern recognition, and parallel processing, all while maintaining the exactness and reversibility that are central to the framework.

5.1 Definitions and Structure

We begin by defining equivariant functions generally, as mappings on Λ_r that respect the symmetries of $\mathbb{T}_{24}$. These functions operate on vertices and their associated data, where their outputs transform consistently under group actions. Encodings are then introduced as a special case for labeling tasks.

TABLE 10. Contrasts with Related Dualities

Duality Type	**Basis/Mapping**	**Key Difference from TQF**	**Applications**
TQF Radial	• Origin-centered circle inversion map ι_r • Exact bijective zone swaps via Escher reflective duality	• N/A	• Symmetric graph ops • Radial networks • Lattice-based cryptography • Multi-agent coordination
Planar (Whitney [21])	• Embedding • Vertex-face swap	• Embedding-dependent with vertex-face swaps • Lacks embedding-independent radial separation via norm trichotomy	• Planarity testing • Embeddings
Matroid (Oxley [74])	• Abstract dependence • Orthogonal complements	• Generalizes independence • No geometric radial structure or origin-centered bijective self-duality	• Optimization • Coding theory
Discrete Conformal (Kharevych [37])	• Circle patterns • Angle preservation	• Typically no radial zone swaps • Focus on angle preservation rather than exact bijective mappings via radial inversion	• Simulations • Angle-preserving maps
Self-Dual Graphs (Graver [46])	• Isomorphic to planar dual • Vertex-face	• Limited to self-isomorphic embeddings • No radial bijective self-duality under order-6 symmetry	• Cataloging polyhedra • Symmetry analysis

TABLE 11. Key Invariance Properties under Dualities

Property	**Description**	**Application Example**
Path Invariance	• Preserves graph ray counts and phase pair assignments	• Symmetric traversals (BFS/DFS)
Rotational Invariance	• Zones and phase pair labels unchanged under rotations	• Reduced computation (theoretical up to 6x in idealized cases, practical 1.7-2x as demonstrated in simulations (Section 6); analogous to benchmarks in [32])
Fixed-Point Invariance	• Boundary subgraph $\Lambda_{T,r}$ stable under ι_r	• Anchor-based searches
Reversible Swapping	• Exact bijective inner-outer zone swap	• Undoable operations in data structures
Mirror Decompositions	• Paired classes via $\mathbb{Z}_2$ subgroup actions	• Binary-symmetric algorithms
Duality Hashing	• Angular sector index and dual norm bin preserved	• $O(1)$ lookups across zones

Definition 5.1 (Equivariant Function). An *equivariant function* is a mapping $f : V(\Lambda_r) \times \mathcal{D} \to \mathcal{D}'$, where $\mathcal{D}$ and $\mathcal{D}'$ are data spaces (e.g., scalars, vectors, labels, functions, or matrices), such that

$$f(g \cdot \vec{v}_i, g \cdot x) = \rho(g)\big(f(\vec{v}_i, x)\big) \tag{15}$$

for all $g \in \mathbb{T}_{24}$, $\vec{v}_i \in V(\Lambda_r)$, $x \in \mathcal{D}$, where $\mathbb{T}_{24}$ acts on $V(\Lambda_r)$ via its symmetries (Definition 4.16) and on $\mathcal{D}$ accordingly, and $\rho : \mathbb{T}_{24} \to \mathrm{GL}(\mathcal{D}')$ is a representation of $\mathbb{T}_{24}$ on $\mathcal{D}'$.

An equivariant encoding is a specific kind of equivariant function, which assigns discrete labels to vertices.

Definition 5.2 (Equivariant Encoding). An *equivariant encoding* is a special case of an equivariant function (Definition 5.1) that assigns labels from a finite set $\mathcal{C}$ (e.g., $\{0, 1\}$ for binary decompositions or $\{0, \ldots, 5\}$ for modular partitions aligned with the order-6 rotational symmetry of $\mathbb{Z}_6$) to the vertices of Λ_r, specifically $e : V(\Lambda_r) \to \mathcal{C}$, while respecting the actions of $\mathbb{T}_{24}$ for symmetry-preserving labeling tasks such as graph coloring or angular sector partitioning.

A natural encoding arises from the angular sectors defined in Lemma 3.10, which assigns each vertex to an angular sector index mod 6.

Definition 5.3 (Angular Sector-Based Six-Encoding). The *angular sector-based six-encoding* is the map $s_6 : V(\Lambda_r) \to \mathbb{Z}_6$ defined by $s_6(\vec{v}_i) := \lfloor 6\langle \vec{v}_i \rangle / (2\pi) \rfloor \bmod 6$, which partitions Λ_r into six classes that correspond to the six angular sectors S_t for $t \in \mathbb{Z}_6$. For a lattice vertex with integer coordinates (a, b) this index is computed exactly, without any floating-point phase, as the unique sector whose bounding primary rays satisfy the integer cross product sign tests $\mathrm{cross}(\vec{d}_t, \vec{v}_i) \geq 0$ and $\mathrm{cross}(\vec{v}_i, \vec{d}_{t+1}) > 0$ (where $\vec{d}_t$ is the primary ray direction at phase $t\pi/3$ and $\mathrm{cross}((p, q), (s, t)) = pt - qs$). This integer realization agrees with the floor-based definition for every off-ray vertex and assigns each on-ray vertex to its counterclockwise sector, which makes s_6 exactly equivariant under the order-6 rotation.

This encoding aligns with Λ_r's order-6 rotational symmetry of $\mathbb{Z}_6$ and preserves the adjacency properties because any two distinct adjacent vertices $\vec{v}_i, \vec{v}_j \in \Lambda_r$ differ by 0 or ± 1 mod 6 along certain directions (as formalized in the upcoming Theorem 5.4).

5.2 Equivariance Properties

Equivariant functions and encodings respect the symmetries of the framework's transformations and establish consistent behavior across the dual zones up to the group actions of $\mathbb{T}_{24}$.

Theorem 5.4 (Equivariance of the Angular Sector-Based Six-Encoding). The angular sector S_t-based six-encoding $s_6 : V(\Lambda_r) \to \mathbb{Z}_6$ (Definition 5.3) is equivariant under the actions of $\mathbb{T}_{24}$, where:

(1) rotations $\mathcal{R}_{k\pi/3}$ (for $k \in \mathbb{Z}_6$) act by $s_6(\mathcal{R}_{k\pi/3}(\vec{v}_i)) = (s_6(\vec{v}_i) + k) \bmod 6$,

(2) reflections in D_6 act by permutations of the labels that preserve the boundaries of the angular sectors S_t (for $t \in \mathbb{Z}_6$), and

(3) the circle inversion map ι_r acts trivially, i.e., $s_6(\iota_r(\vec{v}_i)) = s_6(\vec{v}_i)$ for all $\vec{v}_i \in V(\Lambda_r)$, due to phase preservation.

Proof. To verify the equivariance of the angular sector-based six-encoding $s_6 : V(\Lambda_r) \to \mathbb{Z}_6$ (Definition 5.3) under the action of the Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group $\mathbb{T}_{24}$ (Definition 4.16), recall from Definition 5.1 that equivariance requires $s_6(g \cdot \vec{v}_i) = \rho(g)\big(s_6(\vec{v}_i)\big)$ for all $g \in \mathbb{T}_{24}$ and $\vec{v}_i \in V(\Lambda_r)$, where $\rho : \mathbb{T}_{24} \to \mathrm{Aut}(\mathbb{Z}_6)$ is the natural representation acting by affine transformations (e.g., shifts or permutations preserving the cyclic structure) on the labels—the six possible output values $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ of the S_t-based six-encoding s_6. We proceed in steps to confirm this

for the generators of $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$, where D_6 acts via rotations $\mathcal{R}_{k\pi/3}$ ($k \in \mathbb{Z}_6$) and reflections, and $\mathbb{Z}_2 = \langle \iota_r \rangle$ acts via the circle inversion map ι_r (Definition 4.12).

- **Step 1: Equivariance under Rotations $\mathcal{R}_{k\pi/3}$ ($k \in \mathbb{Z}_6$).** A rotation $\mathcal{R}_{k\pi/3}$ maps each angular sector S_t to $S_{t+k \bmod 6}$ (Lemma 3.10), which induces the representation $\rho(\mathcal{R}_{k\pi/3}) : \mathbb{Z}_6 \to \mathbb{Z}_6$ defined by the affine shift $\rho(\mathcal{R}_{k\pi/3})(l) = (l + k) \bmod 6$. Thus, for any $\vec{v}_i \in V(\Lambda_r)$ with $s_6(\vec{v}_i) = t \in \mathbb{Z}_6$, we have $s_6(\mathcal{R}_{k\pi/3}(\vec{v}_i)) = (t + k) \bmod 6 = \rho(\mathcal{R}_{k\pi/3})(s_6(\vec{v}_i))$, confirming equivariance under the cyclic subgroup $\mathbb{Z}_6 \leq D_6$. ☑
- **Step 2: Equivariance under Reflections in D_6.** The full point symmetry group D_6 has six reflection axes through the origin: three *corner-axes* at phases $k\pi/3 \bmod \pi$ for $k = 0, 1, 2$ (i.e., the three distinct lines through opposite pairs of primary rays at phases $0, \pi$; $\pi/3, 4\pi/3$; and $2\pi/3, 5\pi/3$) and three *edge-midpoint-axes* at phases $\pi/6 + k\pi/3 \bmod \pi$ for $k = 0, 1, 2$ (i.e., the three distinct lines through opposite pairs of edge-midpoint directions at phases $\pi/6, 7\pi/6$; $\pi/2, 3\pi/2$; and $5\pi/6, 11\pi/6$). Each such reflection maps angular sectors via an orientation-reversing involution. As a canonical example, the reflection σ_0 across the primary ray at phase 0 (the positive real axis) sends a point with phase $\theta \in [0, 2\pi)$ to one with phase $(-\theta) \bmod 2\pi$, and hence maps S_t to $S_{-t-1 \bmod 6}$: explicitly, S_t has phase range $[t\pi/3, (t+1)\pi/3)$ (closed at $t\pi/3$, open at $(t+1)\pi/3$ per Definition 3.9), and under $\theta \mapsto -\theta$ this interval maps set-theoretically to $(-(t+1)\pi/3, -t\pi/3]$ (the closed/open endpoints swap because the reflection reverses orientation); after reducing modulo 2π (with the convention that 0 rather than 2π represents the wrap point), this becomes $[(5-t)\pi/3, (6-t)\pi/3) \bmod 2\pi$, which is exactly $S_{5-t \bmod 6} = S_{-t-1 \bmod 6}$. The previously closed lower endpoint $t\pi/3 \in S_t$ now maps to $-t\pi/3 \equiv (6-t)\pi/3 \bmod 2\pi$, which is the (open) upper endpoint of S_{5-t} and hence the (closed) lower endpoint of $S_{6-t \bmod 6}$, consistent with σ_0 exchanging $S_t \leftrightarrow S_{5-t}$ as set-theoretic blocks of the half-open partition (no overlap, no omission). This induces the representation $\rho(\sigma_0) : \mathbb{Z}_6 \to \mathbb{Z}_6$ as the permutation $\rho(\sigma_0)(l) = -l - 1 \bmod 6$, which is a (sector-relabeling) automorphism of $\mathbb{Z}_6$. The five remaining reflections in D_6 are obtained as compositions $\mathcal{R}_{k\pi/3} \circ \sigma_0$ for $k = 1, \ldots, 5$, which (by the standard identity that pre-composing a reflection through angle β with a rotation by α yields a reflection through angle $\beta + \alpha/2$) are exactly the reflections across the six reflection axes of D_6 at phases $k\pi/6$ for $k = 1, \ldots, 5$ (covering both the remaining two corner-axes and all three edge-midpoint-axes enumerated above). Each induces a representation of the form $\rho(\mathcal{R}_{k\pi/3} \circ \sigma_0)(l) = (k - l - 1) \bmod 6$. For $\vec{v}_i \in V(\Lambda_r)$ with $s_6(\vec{v}_i) = t$, we therefore have $s_6(\sigma_0(\vec{v}_i)) = -t - 1 \bmod 6 = \rho(\sigma_0)(s_6(\vec{v}_i))$, and analogously for the other reflections. Thus, equivariance holds under the full D_6. ☑
- **Step 3: Invariance under Circle Inversion ι_r.** The circle inversion map ι_r preserves phases exactly (condition 3 of Definition 4.14 and Proposition 4.15), so $\langle \iota_r(\vec{v}_i) \rangle = \langle \vec{v}_i \rangle$ for all $\vec{v}_i \in V(\Lambda_r)$, implying $s_6(\iota_r(\vec{v}_i)) = s_6(\vec{v}_i)$. This corresponds to the trivial representation $\rho(\iota_r) = \mathrm{id}_{\mathbb{Z}_6}$. This confirms invariance (a special case of equivariance) under the generator of $\mathbb{Z}_2$. ☑
- **Step 4: Overall Equivariance under $\mathbb{T}_{24}$.** Since $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$ is a direct product whose factors commute (Definition 4.16), every element decomposes uniquely as $g = h \cdot \iota_r^\epsilon$ for $h \in D_6$ and $\epsilon \in \{0, 1\}$, with the two factors acting independently on labels. The representation factors accordingly: $\rho(g) = \rho(h) \circ \rho(\iota_r^\epsilon)$. From Steps 1–2, $\rho(h)$ is an automorphism of $\mathbb{Z}_6$ (affine shifts for rotations, sign-reversal for reflections); from Step 3, $\rho(\iota_r) = \mathrm{id}_{\mathbb{Z}_6}$ is trivial. Their composition is therefore an automorphism, and equivariance extends to the full group: $s_6(g \cdot \vec{v}_i) = \rho(h)\big(s_6(\iota_r^\epsilon(\vec{v}_i))\big) = \rho(h)\big(s_6(\vec{v}_i)\big) = \rho(g)\big(s_6(\vec{v}_i)\big)$ holds for all $g \in \mathbb{T}_{24}$ and $\vec{v}_i \in V(\Lambda_r)$, as required by Definition 5.1. ☑

□

Corollary 5.5 ($\mathbb{T}_{24}$-Symmetric Equivariant Function Extension). Equivariant functions (Definition 5.1) are compatible with the discrete dual metric (Definition 4.27), the continuous dual metric (Definition 4.31), and bijective self-duality (Proposition 4.38) because they commute with the actions of $\mathbb{T}_{24}$ (Definition 4.16). This drives symmetry-respecting operations, such as convolutions on graph rays or reductions that commute with these actions.

Example 5.6. For instance, let's consider a scalar equivariant function f (Definition 5.1) that aggregates vertex attributes along graph rays (Definition 3.31)—such as summing edge weights or node values to compute a metric like cumulative signal strength in a symmetrically designed Wi-Fi network. The Escher reflective duality (Proposition 4.15) and inversion invariance under $\mathbb{T}_{24}$ (Definition 4.16) ensure that f remains stable under circle inversion, which allows us to rapidly reflect/mirror the outer zone results directly to the inner zone via bijective self-duality (Proposition 4.38) without fully recomputing the remaining inner half. (Note: this stability assumes that the per-vertex or per-edge attributes themselves transport along ι_r—as guaranteed by Corollary 4.39—and is therefore an invariance of the *combinatorial* aggregate, not of geometric edge lengths, which differ between zones by the variable scaling of ι_r.) This approach aligns with network flow optimization (e.g., using $\Lambda^R_{T,r}$ as a minimal cut, as per Corollary 4.10 and [66]), where symmetric flows between $\Lambda^R_{-,r}$ and $\Lambda^R_{+,r}$ can be computed in one zone and reflected, thereby reducing redundancy in hierarchical decompositions (e.g., layering truncated Λ^R_r by angular sectors) or parallel traversals (e.g., where computational workloads are uniformly distributed across processors and the resulting outputs are flipped via reversible zone swapping from Corollary 4.24).

Building on the equivariance of the angular sector-based six-encoding s_6 under the symmetries of $\mathbb{T}_{24}$, we can further refine these encodings through modular reductions to create coarser partitions that still respect $\mathbb{T}_{24}$'s actions. This adaptive approach allows us the flexibility to group the angular sectors into fewer classes while maintaining consistency across rotations, reflections, and inversions. This is similar to filtering or simplifying a color arrangement/scheme without losing any symmetry information.

Corollary 5.7 ($\mathbb{T}_{24}$-Adaptive Modular Decompositions). The modular encodings $s_m(\vec{v}_i) = s_6(\vec{v}_i) \mod m$, derived from the angular sector-based six-encoding s_6 for divisors m of 6 (i.e., $m \in \{1, 2, 3, 6\}$, such as $m = 2$ yielding a binary even-odd decomposition of angular sectors per Corollary 5.8), are equivariant encodings under the symmetries of $\mathbb{T}_{24}$. These yield modular decompositions of Λ_r into m equivariant classes aligned with the subgroup actions of $\mathbb{Z}_6$.

Such modular equivariant encodings enable hierarchical decompositions of Λ_r, where finer six-class partitions can be aggregated into binary or ternary structures for tasks like load-balanced processing. For example, in robotics path planning on symmetric warehouse layouts modeled by Λ_r, a binary decomposition could assign even and odd angular sectors to separate processing threads to allow concurrent traversal of independent subgraphs to halve the computation time while maintaining a structured orientation via phase pair assignments. This sets the stage for specific applications, such as the binary decomposition that follows, which further leverages these properties for parallel algorithms and symmetry-aware optimizations.

Corollary 5.8 (Binary Even-Odd Angular Sector Decomposition). The binary encoding $e_2(\vec{v}_i) = s_6(\vec{v}_i) \bmod 2$, obtained via modular reduction from the angular sector-based six-encoding s_6 (Definition 5.3), decomposes Λ_r into two interleaved classes: the even class $\bigcup_{t \equiv 0 \bmod 2} S_t = S_0 \cup S_2 \cup S_4$ and the odd class $\bigcup_{t \equiv 1 \bmod 2} S_t = S_1 \cup S_3 \cup S_5$, which correspond to the two $\mathbb{Z}_3$-orbits under the order-3 rotational subgroup action (as per the Corollary on subgroup decompositions following Proposition 4.2). These classes are preserved under the Escher reflective duality (Proposition 4.15) via the bijective mapping across zones (Proposition 4.38), with invariance under circle inversion ι_r yielding $e_2(\iota_r(\vec{v}_i)) = e_2(\vec{v}_i)$. This decomposition is suitable for parallel processing, such as load-balanced traversal of even/odd angular sectors.

Equivariant functions and encodings drive the design and implementation of symmetry-exploiting algorithms such as modular hashing and conflict-free access.

Example 5.9 (Parallel Processing with Binary Decomposition). Utilizing the binary equivariant encoding $e_2(\vec{v}_i) = s_6(\vec{v}_i) \bmod 2$ (as per Corollary 5.8), we decompose the angular sectors into the even class $\bigcup_{t \equiv 0 \bmod 2} S_t = S_0 \cup S_2 \cup S_4$ and the odd class $\bigcup_{t \equiv 1 \bmod 2} S_t = S_1 \cup S_3 \cup S_5$ for thread assignment. This equips us with coarse grained parallelism at the class level (though cross-class edges may require an additional synchronization overhead cost).

These equivariant decompositions equip TQF with foundational tools achieving robust, efficient computations on Λ_r. To further enhance parallelism and facilitate conflict-free operations across independent sets, we now introduce the trihexagonal six-coloring.

5.3 Trihexagonal Six-Coloring

Building on the angular sector-based six-encoding and leveraging the honeycomb lattice as the dual of our triangular lattice L (as referenced in Subsection 3.2), we introduce a trihexagonal six-coloring as an equivariant function. This extends the proper three-colorability of the triangular lattice L [18]—upon which Λ_r is based—to six colors, which facilitates enhanced parallel computations through finer independent set decompositions.

Λ_r admits a proper three-coloring $c : \Lambda_r \to \{0, 1, 2\}$, where no adjacent vertices share the same color, that is equivariant under the order-6 rotational symmetry up to permutation of colors (because Λ_r's triangular structure allows cyclic shifts of the colors under D_6).

Definition 5.10 (Trihexagonal Six-Coloring). The *trihexagonal six-coloring* $e_6 : \Lambda_r \to \{0, \ldots, 5\}$ is defined as

$$e_6(\vec{v}_i) := 2 \cdot c(\vec{v}_i) + (s_6(\vec{v}_i) \bmod 2), \tag{16}$$

which merges the lattice three-coloring with angular sector parity to yield six distinct classes.

Remark 5.11. *In Definition 5.10, the three-coloring $c : \Lambda_{+,r} \to \{0, 1, 2\}$ is a proper coloring of $\Lambda_{+,r}$, which inherits the standard proper three-colorability of the base triangular lattice L (as per Subsection 3.2). This coloring is then induced exactly on $\Lambda_{-,r}$ via the graph isomorphism of Corollary 4.21 (due to the Escher reflective duality of Proposition 4.15), which ensures that the adjacency and chromatic properties are preserved across zones without approximation.*

Remark 5.12. *The "trihexagonal" part of the name in Definition 5.10 is inspired by the trihexagonal tiling (also known as the kagome lattice), which is the medial graph of the triangular lattice L and exhibits related coloring properties. In such lattice graphs, equivariant functions (per Definition 5.1) extend six-colorings to operations on independent sets to support computational tasks like matching or decomposition in network algorithms. More structurally, the medial graph of Λ_r—combinatorially the kagome lattice with its three sublattices indexed by the $\mathbb{Z}_3$ direction-pair classes $\{0, 3\}, \{1, 4\}, \{2, 5\}$ of Remark 3.42—inherits a six-coloring from e_6 via edge-to-vertex incidence, where the central inversion ι_r trivially acts on the resulting color labels (as it does on e_6 itself, per Theorem 5.15). This medial structure is the natural setting for edge-based equivariant labelings on Λ_r that respect the full $\mathbb{T}_{24}$-action.*

Corollary 5.13 (Proper Coloring). The trihexagonal six-coloring e_6 (Definition 5.10) is a proper six-coloring of Λ_r because any two adjacent vertices have distinct c values (that differ by either 1 or 2 mod 3), so their $2 \cdot c$ differs by at least 2 mod 6, which secures distinct e_6 even if the angular sector parities match. Thus, each color class is an independent set, and therefore suitable for conflict-free parallel algorithms.

Remark 5.14 (Why six classes rather than three?). *The underlying triangular lattice is three-chromatic, and the three-coloring $c = (a - b) \bmod 3$ alone already properly colors Λ_r. And as we made explicit in the proof of Corollary 5.13, the properness of e_6 is derived* entirely *from c, because the angular-sector parity term $s_6 \bmod 2$ does not contribute any additional separation. Thus, the six classes of e_6 are therefore not a chromatic minimum but a deliberate refinement of the proper three-coloring, where each c-class is split along the $\mathbb{Z}_6$ angular-sector parity so that the resulting labeling is exactly $\mathbb{T}_{24}$-equivariant (Theorem 5.15) and its six classes align one-to-one with the order-6 rotational structure that is exploited by the orbit-transversal computations. This $\mathbb{Z}_6$-alignment—not a reduction in the number of colors—is what positions e_6 as the*

natural conflict-free schedule for the symmetry-reduced parallel algorithms of Section 6. A bare three-coloring would suffice for independence but would not carry the rotational structure that the framework exploits.

Theorem 5.15 (Equivariance of Trihexagonal Six-Coloring). The trihexagonal six-coloring $e_6 : \Lambda_r \to \{0, \dots, 5\}$ (Definition 5.10) is equivariant under the action of $\mathbb{T}_{24}$ (Definition 4.16), where rotations and reflections in D_6 each act by a fixed permutation of the six labels (i.e., the action factors through a homomorphism $D_6 \to S_6$, not generally a cyclic shift), and the circle inversion ι_r operates trivially (i.e., $e_6(\iota_r(\vec{v}_i)) = e_6(\vec{v}_i)$ for all $\vec{v}_i \in \Lambda_r$), which thereby supports the proper six-coloring property across all transformations.

Proof. We proceed in steps to verify equivariance of the trihexagonal six-coloring e_6 under the action of $\mathbb{T}_{24}$:

- **Step 1: Verify Equivariance of the Proper Three-Coloring c under D_6.** The proper three-coloring $c : \Lambda_r \to \{0, 1, 2\}$ of Λ_r (induced from the base L via the graph isomorphism of Corollary 4.21) is preserved under the symmetries of D_6 up to permutation of the colors, which it inherits from the equilateral structure and order-6 rotational symmetry of L (Subsection 3.2). Concretely, taking the standard proper three-coloring $c(a\vec{\omega}_0 + b\vec{\omega}_1) = (a - b) \bmod 3$ and using the lattice-coordinate action $\mathcal{R}_{\pi/3} : (a, b) \mapsto (-b, a + b)$, direct computation gives $c \mapsto -(a - b) \bmod 3 = -c \bmod 3$ under $\mathcal{R}_{\pi/3}$. Therefore rotations $\mathcal{R}_{k\pi/3}$ (for $k \in \mathbb{Z}_6$) act on the colors as $c \mapsto (-1)^k c \bmod 3$, i.e., $\mathcal{R}_{\pi/3}$ swaps colors 1 and 2 while fixing color 0, and $\mathcal{R}_{2\pi/3}$ fixes c pointwise. Equivalently, the induced D_6-action on $\{0, 1, 2\}$ factors through the quotient $D_6 \twoheadrightarrow \mathbb{Z}_2$ (consistent with the edge-direction analogue in Remark 3.42). The six reflections in D_6 (across the three corner-axes through primary lattice radial rays and the three edge-midpoint-axes through nearest-neighbor edge midpoints, as enumerated in Step 2 of the proof of Theorem 5.4) likewise permute the three colors via this $\mathbb{Z}_2$-quotient. ☑
- **Step 2: Verify Equivariance of the Angular Sector-Based Six-Encoding s_6 and Angular Sector Parity under D_6.** The angular sector-based six-encoding $s_6 : \Lambda_r \to \mathbb{Z}_6$ (Definition 5.3) is equivariant under D_6 by Theorem 5.4, where each rotation adds k mod 6 to the angular sector index and each reflection maps sectors via a fixed permutation (e.g., the reflection σ_0 across the positive real axis gives $S_t \mapsto S_{-t-1 \bmod 6}$, equivalently $S_{5-t \bmod 6}$). The angular sector parity $(s_6(\vec{v}_i) \bmod 2)$ inherits this equivariance: rotations $\mathcal{R}_{k\pi/3}$ shift parity by k mod 2 (a uniform flip for odd k, identity for even k), and reflections $\mathcal{R}_{k\pi/3} \circ \sigma_0$ send parity $p \mapsto (k + p + 1) \bmod 2$ (uniform flip when k is even, identity when k is odd, as worked out explicitly in Step 4 below). In all cases the action on parity is a uniform $\mathbb{Z}_2$-action (either the identity or the global involution $p \mapsto p + 1 \bmod 2$), not a sector-by-sector relabeling. ☑
- **Step 3: Verify Behavior of e_6 under Rotations in D_6.** Under $\mathcal{R}_{k\pi/3}$, we have $c \mapsto (-1)^k c \pmod 3$ (Step 1) and $s_6 \mapsto s_6 + k \pmod 6$ (Theorem 5.4), so $(s_6 \bmod 2) \mapsto (s_6 + k) \bmod 2$. Hence
$$e_6(\mathcal{R}_{k\pi/3}(\vec{v}_i)) = 2\big((-1)^k c(\vec{v}_i) \bmod 3\big) + \big((s_6(\vec{v}_i) + k) \bmod 2\big) \pmod 6,$$
which depends only on the pair $(c(\vec{v}_i), s_6(\vec{v}_i) \bmod 2)$, and hence only on $e_6(\vec{v}_i)$. The induced map on labels is therefore a well-defined permutation of $\{0, 1, 2, 3, 4, 5\}$. For example, identifying each label $l \in \{0, \dots, 5\}$ with the pair $(c, s_6 \bmod 2) = (\lfloor l/2 \rfloor, l \bmod 2)$, the rotation $\mathcal{R}_{\pi/3}$ induces the involution (0 1)(2 5)(3 4) on labels (since $(c, p) \mapsto (-c \bmod 3,\ p + 1 \bmod 2)$), and $\mathcal{R}_{2\pi/3}$ acts trivially on labels (since it fixes c and adds 2 to s_6, preserving parity). The action of the full rotation subgroup $\mathbb{Z}_6 \leq D_6$ therefore factors through $\mathbb{Z}_6 \twoheadrightarrow \mathbb{Z}_2 = \langle \mathcal{R}_{\pi/3} \rangle / \langle \mathcal{R}_{2\pi/3} \rangle$, generated by the label involution above. ☑
- **Step 4: Verify Behavior of e_6 under Reflections in D_6.** Each reflection $\sigma \in D_6$ induces a permutation of $\{0, 1, 2\}$ on c (Step 1: D_6 acts on c via the quotient $D_6 \twoheadrightarrow \mathbb{Z}_2$, and reflections lie in nontrivial $\mathbb{Z}_2$-cosets accordingly) and a fixed permutation of $\mathbb{Z}_6$ on s_6 (Theorem 5.4: e.g., σ_0 sends $s_6 \mapsto -s_6 - 1 \bmod 6$, which uniformly flips the parity $(s_6 \bmod 2) \mapsto (s_6 + 1) \bmod 2$ for every vertex; analogously, the reflection $\mathcal{R}_{k\pi/3} \circ \sigma_0$ sends $s_6 \mapsto (k - s_6 - 1) \bmod 6$, which acts on parity by $(s_6 \bmod 2) \mapsto (k + s_6 + 1) \bmod 2$—uniformly flipping parity when k is even and preserving parity when

k is odd). The resulting action on $e_6 = 2c + (s_6 \bmod 2)$ depends only on the pair $(c(\vec{v}_i), s_6(\vec{v}_i) \bmod 2)$ and therefore descends to a well-defined permutation of the six labels $\{0, \ldots, 5\}$. Distinctness across adjacent vertices is preserved because the underlying three-coloring c is proper (Step 6 below). ☑

- **Step 5: Verify Behavior of e_6 under Circle Inversion ι_r.** Under circle inversion ι_r, phase preservation (Proposition 4.15) implies $s_6(\iota_r(\vec{v}_i)) = s_6(\vec{v}_i)$ and the induced isomorphism on c (Corollary 4.21) implies $c(\iota_r(\vec{v}_i)) = c(\vec{v}_i)$, so $e_6(\iota_r(\vec{v}_i)) = e_6(\vec{v}_i)$. ☑
- **Step 6: Verify Overall Equivariance under $\mathbb{T}_{24}$ and Preservation of Proper Six-Coloring.** The direct product structure of $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$—where $\mathbb{Z}_2 = \langle \iota_r \rangle$ is central and acts trivially on labels (Step 5)—assures that the combined action remains consistent because the circle inversion map ι_r commutes with every element of D_6 and acts as the identity on both the parity and three-coloring components. Hence, e_6 is equivariant under the full $\mathbb{T}_{24}$, and the proper six-coloring property—that no two adjacent vertices share the same color—is preserved, since the scaling by two separates the three-color classes by at least 2 mod 6, which bypasses conflicts even under parity reversal. ☑

□

The trihexagonal six-coloring e_6 (Definition 5.10) partitions Λ_r into six independent sets, which establishes conflict-free queue processing across threads (or nodes or processors) in parallel BFS for theoretical speedups up to 6x (via partitioning into six independent sets) in idealized symmetry-reduced workloads, which is analogous to coloring-based parallel BFS benchmarks in [32].

ALGORITHM PSEUDOCODE 6. Six-way parallel breadth-first search (BFS) via the trihexagonal six-coloring e_6 on the radial dual triangular lattice graph Λ_r (level-synchronous scheduling for full traversal of Λ_r, where the six independent sets induced by the proper coloring support conflict-free parallel processing per level). Note: Full concurrent implementations require locks or atomic updates for shared structures like distances, queue, and visited. The preprocessing to assign vertices to color lists via e_6 has $O(|V|)$ runtime, which is amortized over multiple traversals. We assume efficient data structures (e.g., hash sets for $O(1)$ average-case set intersections), and the coloring's equivariance under ι_r (Theorem 5.15) preserves bijections across zones. In practice, this yields $O(|V|/6 + C)$ runtime across the six color classes, where C denotes synchronization overhead cost.

```
function ParallelBFSviaTH(graph Λ_r, start_vertices):
    # Preprocess: Assign trihexagonal six-colors and initialize global structures
    queue = start_vertices  # Global queue (multi-source support)
    distances = {}  # Global distances
    colors = array of 6 empty lists
    visited = set()  # Global visited set
    for v in Λ_r:
        col = e_6(v)  # Compute trihexagonal six-coloring
        colors[col].append(v)

    # Level-synchronous traversal
    while queue not empty:
        level_queue = queue.copy()  # Process current level
        queue = []  # Next level queue
        parallel for col in 0 to 5:  # Process each independent set independently (no
    intra-color edges)
            local_level = set intersection of level_queue and colors[col]
            for v in local_level:
                for neighbor in neighbors(v):  # Update distances and enqueue next level
                    if not visited(neighbor):
                        lock(global_structures)  # Lock global structures before update
                        distances[neighbor] = distances[v] + 1
                        queue.append(neighbor)
                        visited.add(neighbor)
                        unlock(global_structures)  # Unlock after update
```

```
    return distances
```

For conflict-free access in concurrent systems (e.g., multi-agent coordination on Λ_r), the trihexagonal six-coloring schedules non-adjacent operations across its independent sets to totally eliminate the risk of edge conflicts for each timestep [32].

This equivariant encoding leverages the $\mathbb{T}_{24}$ symmetries to induce practical partitions of Λ_r into six independent sets, thereby enhancing algorithmic efficiency in conflict-free parallel computations (with no approximations).

5.4 Face Structure and Face Zone Trichotomy

We close this section with two complementary structural additions to Λ_r. The first, developed in this subsection, is a face complex that extends the vertex zone trichotomy to 2-cells. The second, developed in Subsection 5.5, is a $\mathbb{Z}_2$-valued chain complex that organizes vertex, edge, and face labelings into a single $\mathbb{T}_{24}$-equivariant algebraic object. Together with the consistent edge framing of Definition 3.39, these auxiliary structures support direction-aware, cycle-aware, and face-aware computations on Λ_r (e.g., planar-embedding algorithms, cycle space algorithms, and network reliability analyses), all while remaining compatible with the dualities, equivariant encodings, and bijective self-duality of the preceding sections.

Why introduce faces at all? Up to this point in the paper, Λ_r has lived as a 1-complex: just vertices and edges. This is enough for adjacency, shortest paths, colorings, and the dualities of Sections 4 and 5. It is not enough, however, for any algorithm that needs to reason about 2D structure—what's inside versus outside a cycle, which region a vertex belongs to, or whether two cycles bound the same enclosed area. Lifting Λ_r to a 2-complex by attaching its triangular faces gives us that capability without changing anything established so far: every vertex-based result remains intact, and we additionally gain a face set $F_r = F_{+,r} \sqcup F_{T,r} \sqcup F_{-,r}$ that mirrors the vertex zone trichotomy exactly. The price is one bookkeeping step (Definition 5.16, Definition 5.17, Definition 5.18) and one bijection (Corollary 5.20). The payoff is that any face-level or cycle-level computation on $F_{+,r}$—region growing, planar cycle enumeration, dual-graph traversal, region-based clustering—extends to $F_{-,r}$ via ι_r without recomputation, exactly the same symmetry-reduction we already established for vertex-level computations.

The base triangular lattice L admits a canonical planar embedding in $X = \mathbb{C} \setminus \{(0,0)_C\}$, in which the bounded 2-cells (faces) are exactly the equilateral triangles whose three vertices lie in L at mutual unit Euclidean distance. We use this canonical embedding to equip $\Lambda_{+,r}$, and then by transport $\Lambda_{-,r}$, with a face structure.

Definition 5.16 (Outer Face Set). The *outer face set* $F_{+,r}$ is the set of bounded triangular 2-cells of the canonical planar embedding of L all three of whose vertices lie in $V_{+,r}$. Each face $f \in F_{+,r}$ is identified by its (unordered) triple of bounding vertices $\{\vec{v}_a, \vec{v}_b, \vec{v}_c\} \subset V_{+,r}$ with pairwise unit Euclidean distance. Because $\Lambda_{+,r}$ is the induced subgraph of L on $V_{+,r}$, the three bounding edges of such a triangle automatically lie in $E_{+,r}$, so each $f \in F_{+,r}$ is a triangular face of $\Lambda_{+,r}$ as a graph.

Definition 5.17 (Inner Face Set). The *inner face set* $F_{-,r}$ is defined by transport along ι_r as

$$F_{-,r} := \{\, \iota_r(f) \mid f \in F_{+,r} \,\}, \qquad \iota_r(\{\vec{v}_a, \vec{v}_b, \vec{v}_c\}) := \{\iota_r(\vec{v}_a), \iota_r(\vec{v}_b), \iota_r(\vec{v}_c)\},$$

so each inner face is the ι_r-image of a unique outer face (its *twin*).

Definition 5.18 (Boundary Face Set). The *outer side boundary face set* $F^+_{T,r}$ is the set of bounded triangular 2-cells of the canonical planar embedding of L that have at least one vertex in $V_{T,r}$ and all remaining vertices in $V_{T,r} \cup V_{+,r}$. The *inner side boundary face set* $F^-_{T,r}$ is defined by transport along ι_r as $F^-_{T,r} := \{\iota_r(f) : f \in F^+_{T,r},\ f \notin V_{T,r}\}$, where ι_r fixes $V_{T,r}$ pointwise and maps $V_{+,r}$ to $V_{-,r}$ componentwise on vertex triples. The *boundary face set* is $F_{T,r} := F^+_{T,r} \cup F^-_{T,r}$. Equivalently, $F_{T,r}$ consists of those triangular 2-cells of the 2-complex on Λ_r that touch $V_{T,r}$, where the outer side faces are inherited from L and the inner side

faces are obtained by transport along ι_r. By construction, every face in $F_{T,r}$ has all three bounding edges in E_r (since edges of the canonical L-embedding with both endpoints in $V_{T,r} \cup V_{+,r}$ lie in E_r by Definition 3.21, and edges of the inner side faces lie in E_r via transport).

Remark 5.19. *By construction, $F_{+,r}$ and $F^+_{T,r}$ are face sets of the canonical planar embedding of L, while $F_{-,r} = \iota_r(F_{+,r})$ and $F^-_{T,r}$ are transported face sets obtained from $F_{+,r}$ and $F^+_{T,r}$ respectively along ι_r. The three sets $F_{+,r}$, $F_{T,r}$, $F_{-,r}$ are pairwise disjoint by construction and together form $F_r := F_{+,r} \sqcup F_{T,r} \sqcup F_{-,r}$, the face set of the 2-complex on Λ_r used in Subsection 5.5. The cardinalities satisfy $|F_{+,r}| = |F_{-,r}|$ and $|F^+_{T,r}| = |F^-_{T,r}|$ exactly by the bijection of Definition 5.17 (where we note that any all-boundary faces with $f \subset V_{T,r}$ are ι_r-fixed and contribute to $F^+_{T,r}$ only, with $F^-_{T,r}$ excluding them by construction).*

Corollary 5.20 (Face Trichotomy Bijection). The circle inversion ι_r induces a bijection $F_{+,r} \leftrightarrow F_{-,r}$ on outer and inner faces, defined by mapping each face's vertex triple componentwise. This bijection commutes with $\mathbb{T}_{24}$ in the sense that for every $g \in \mathbb{T}_{24}$ and every $f \in F_{+,r}$,

$$g \cdot \iota_r(f) = \iota_r(g \cdot f),$$

where g acts on faces componentwise on their vertex triples.

Proof. Bijectivity follows from bijectivity of ι_r on $V_{+,r} \to V_{-,r}$ (Proposition 4.38): distinct vertex triples in $V^3_{+,r}$ map to distinct vertex triples in $V^3_{-,r}$, and the construction $F_{-,r} = \iota_r(F_{+,r})$ is surjective by definition. Equivariance with $\mathbb{T}_{24}$ follows because each $g \in \mathbb{T}_{24}$ acts componentwise on vertex triples and commutes with ι_r on individual vertices (Definition 4.16: ι_r is central in $\mathbb{T}_{24}$), so the action on faces respects the same commutativity. □

Remark 5.21. *The face trichotomy $F_r = F_{+,r} \sqcup F_{T,r} \sqcup F_{-,r}$ parallels the vertex trichotomy $V_r = V_{+,r} \sqcup V_{T,r} \sqcup V_{-,r}$ and is similarly compatible with $\mathbb{T}_{24}$, which makes face-based and cycle-based algorithms (e.g., planar cycle enumeration, dual-graph traversals, region-based clustering) directly inherit the same symmetry-reduction benefits as the vertex-based algorithms of Sections 4 and 5. In particular, face-level computations on $F_{+,r}$ extend immediately to $F_{-,r}$ by the bijection of Corollary 5.20, without recomputation.*

5.5 Equivariant Chain Complex of Λ_r over $\mathbb{Z}_2$

Building on the vertex, edge, and face sets of Λ_r, we now organize $\mathbb{Z}_2$-valued labelings of these three layers into a chain complex that is compatible with the actions of $\mathbb{T}_{24}$. Many lattice graph algorithms can be phrased as questions about $\mathbb{Z}_2$-valued state: which edges are "on" versus "off" in a configuration (network reliability under random edge failures [49]), which vertices belong to a parity class (checkerboard-style scheduling and/or bipartite-coloring constraints), or which faces are marked versus unmarked (cycle space and cut space decompositions for max-flow / min-cut analyses [21]). These three questions are usually handled by three separate ad-hoc bookkeeping schemes. The chain complex unifies them by stacking the three layers by dimension—faces as 2-cells (C_2), edges as 1-cells (C_1), and vertices as 0-cells (C_0)—so that a single boundary calculus $\partial_2 : \text{faces} \to \text{edges}$ and $\partial_1 : \text{edges} \to \text{vertices}$ ties them together. The fundamental identity $\partial_1 \circ \partial_2 = 0$ (Corollary 5.26) then guarantees that the resulting cycle space, cut space, and boundary space are well-defined regardless of which layer the algorithm starts on. The entire construction is $\mathbb{T}_{24}$-equivariant (Proposition 5.25), so every $\mathbb{Z}_2$-labeling algorithm—not just those originally designed with the framework in mind—inherits the same symmetry-reduction benefits that are already established for vertex-based computations. The chain complex itself is standard graph homology [18]. Our contribution is the $\mathbb{T}_{24}$-equivariance, which makes that standard construction directly compatible with TQF's dualities and encodings.

Before the formal definitions, here is a brief chain complex "warmup." A k-chain is just a bit-vector indexed by the k-cells of Λ_r: a 0 or 1 on each vertex ($k = 0$), each edge ($k = 1$), or each face ($k = 2$),

which marks that cell as either off or on. Addition is over $\mathbb{Z}_2$, so summing two chains is bitwise XOR, which is the symmetric difference of the two marked cell sets. The boundary operator sends a region to its rim, which takes an edge to its two endpoints and a triangle to its three edges. Everything in this subsection is assembled from those two moves: XOR and "take-the-rim." We note one caution regarding a reused symbol: the $\mathbb{Z}_2$ that supplies the coefficients here is the two element scalar ring $\{0, 1\}$, which is a different role from the $\mathbb{Z}_2 = \mathrm{gen}(\iota_r)$ factor inside $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$, even though both are the order two object. As a first contact, the single edge $\{\vec{v}_0, \vec{v}_1\}$ has the boundary $\partial_1(\{\vec{v}_0, \vec{v}_1\}) = \vec{v}_0 + \vec{v}_1$, and any triangular face $\{\vec{v}_a, \vec{v}_b, \vec{v}_c\}$ has a boundary that is equal to its three bounding edges.

Definition 5.22 ($\mathbb{Z}_2$-Valued Chains on Λ_r). For each $k \in \{0, 1, 2\}$, the space of k-chains on Λ_r with $\mathbb{Z}_2$ coefficients is

$$C_0(\Lambda_r; \mathbb{Z}_2) := \mathbb{Z}_2^{V_r}, \qquad C_1(\Lambda_r; \mathbb{Z}_2) := \mathbb{Z}_2^{E_r}, \qquad C_2(\Lambda_r; \mathbb{Z}_2) := \mathbb{Z}_2^{F_r},$$

where V_r, E_r, and F_r are the vertex, edge, and face sets of Λ_r (with $F_r = F_{+,r} \cup F_{T,r} \cup F_{-,r}$ as in Subsection 5.4). Each chain space is a $\mathbb{Z}_2$-vector space whose basis is the corresponding set of cells. For a set S of k-cells, we write $[S] \in C_k(\Lambda_r; \mathbb{Z}_2)$ for its *indicator chain*, the bit-vector equal to 1 on the cells of S and 0 elsewhere. For instance, the boundary hexagon $\Lambda_{T,1}$ gives the 1-chain $[\Lambda_{T,1}]$ that marks its six edges and no others.

Definition 5.23 (Boundary and Coboundary Operators). Define the $\mathbb{Z}_2$-linear *boundary operators*

$$\partial_1 : C_1(\Lambda_r; \mathbb{Z}_2) \to C_0(\Lambda_r; \mathbb{Z}_2), \qquad \partial_2 : C_2(\Lambda_r; \mathbb{Z}_2) \to C_1(\Lambda_r; \mathbb{Z}_2)$$

by their action on basis elements:

$$\partial_1(\{\vec{v}_i, \vec{v}_j\}) := \vec{v}_i + \vec{v}_j, \qquad \partial_2(\{\vec{v}_a, \vec{v}_b, \vec{v}_c\}) := \{\vec{v}_a, \vec{v}_b\} + \{\vec{v}_b, \vec{v}_c\} + \{\vec{v}_a, \vec{v}_c\},$$

where addition is in $\mathbb{Z}_2$. The *coboundary operators* $\partial_0^* : C_0 \to C_1$ and $\partial_1^* : C_1 \to C_2$ are the transpose (adjoint) maps with respect to the standard bases.

Definition 5.24 (Cycle, Cut, and Boundary Spaces). Within $C_1(\Lambda_r; \mathbb{Z}_2)$ we single out three subspaces, the standard $\mathbb{Z}_2$ cycle, bond, and boundary spaces of algebraic graph theory [18, 50]:

- the *cycle space* $\ker \partial_1$ (often written Z_1), the link configurations that close up;
- the *cut space*, or bond space, $\mathrm{im}\, \partial_0^*$, the edge cuts of vertex subsets; and
- the *boundary space* $\mathrm{im}\, \partial_2$ (often written B_1), the cycles that bound a set of faces.

A chain x lies in the cycle space exactly when $\partial_1 x = 0$, which says every vertex meets an even number of active edges, so the configuration has no loose ends—these are the even subgraphs that drive network reliability analysis [49]. The cut space collects $\partial_0^*[S]$ over vertex subsets S, the edges crossing the boundary of S, which is exactly what one severs to disconnect S from the rest of the graph—these bonds are the objects behind max-flow / min-cut and planar duality [21]. The boundary space sits inside the cycle space, $\mathrm{im}\, \partial_2 \subseteq \ker \partial_1$ by Corollary 5.26, and collects the loops that are rims of some set of triangular faces, the ones that can be "filled in." These three spaces are why the boundary calculus is valuable and applicable, because a wide range of lattice graph algorithms reduce to a membership or projection test against one of these three. For a concrete instance, take any one triangular face of Λ_1: its three edges form a cycle, since each of the face's three vertices meets exactly two of them, and that same edge triple is the boundary ∂_2 of the face, so it is a boundary as well. Separately, the edges that are incident to a single vertex $\vec{v}$ form the cut $\partial_0^*[\{\vec{v}\}]$ that isolates $\vec{v}$ from the rest of the graph.

Proposition 5.25 ($\mathbb{T}_{24}$-Equivariance of the Chain Complex). For every $g \in \mathbb{T}_{24}$, let $\rho_k(g) : C_k(\Lambda_r; \mathbb{Z}_2) \to C_k(\Lambda_r; \mathbb{Z}_2)$ denote the linear extension of the g-action on k-cells (vertices for $k = 0$, edges for $k = 1$, faces for $k = 2$). Concretely, g permutes the k-cells, so it permutes the entries of a bit-vector, and $\rho_k(g)$ is precisely that permutation realized as a $\mathbb{Z}_2$-linear map. The family $\{\rho_k(g)\}_{g \in \mathbb{T}_{24}}$ is a linear representation of $\mathbb{T}_{24}$ on C_k. The equivariance below then secures the fact: taking the boundary and then applying the symmetry yields the same chain as applying the symmetry and then taking the boundary. Then the boundary operators ∂_1 and ∂_2 of Definition 5.23 commute with the action:

$$\rho_0(g) \circ \partial_1 = \partial_1 \circ \rho_1(g), \qquad \rho_1(g) \circ \partial_2 = \partial_2 \circ \rho_2(g),$$

for all $g \in \mathbb{T}_{24}$. Consequently, the cycle space $\ker \partial_1$, the cut space $\operatorname{im} \partial_0^*$, and the boundary space $\operatorname{im} \partial_2$ (Definition 5.24), and their analogues in higher dimensions, are all $\mathbb{T}_{24}$-invariant subspaces of the corresponding chain spaces.

Proof. By linearity it suffices to verify the commutation relations on basis elements.

- **Step 1: Commutation for ∂_1.** Let $\{\vec{v}_i, \vec{v}_j\} \in E_r$ and $g \in \mathbb{T}_{24}$. By the componentwise action on edges (Subsection 5.4 for the analogous face case, where the edge case is identical), $\rho_1(g)(\{\vec{v}_i, \vec{v}_j\}) = \{g \cdot \vec{v}_i, g \cdot \vec{v}_j\}$. Applying ∂_1 gives $\{g \cdot \vec{v}_i\} + \{g \cdot \vec{v}_j\} = \rho_0(g)(\{\vec{v}_i\} + \{\vec{v}_j\}) = \rho_0(g)(\partial_1(\{\vec{v}_i, \vec{v}_j\}))$. As a concrete instance on Λ_1, the rotation $g = \mathcal{R}_{\pi/3}$ sends $\vec{v}_t \mapsto \vec{v}_{t+1}$ around the boundary hexagon, so $\rho_1(g)$ advances each edge $\{\vec{v}_t, \vec{v}_{t+1}\}$ to $\{\vec{v}_{t+1}, \vec{v}_{t+2}\}$, and both routes send that edge to $\vec{v}_{t+1} + \vec{v}_{t+2}$ in C_0. ☑
- **Step 2: Commutation for ∂_2.** The case for ∂_2 is analogous, using the componentwise face action and the fact that each g permutes the three boundary edges of a face among themselves. ☑
- **Step 3: Invariance of Derived Subspaces.** Invariance of cycle, cut, and boundary spaces then follows since each is the kernel or image of an equivariant linear map. ☑

□

Corollary 5.26 (Chain Complex Axiom). The composition $\partial_1 \circ \partial_2 : C_2(\Lambda_r; \mathbb{Z}_2) \to C_0(\Lambda_r; \mathbb{Z}_2)$ is zero. Hence $(C_\bullet(\Lambda_r; \mathbb{Z}_2), \partial_\bullet)$ is a $\mathbb{T}_{24}$-equivariant chain complex with well-defined $\mathbb{Z}_2$-homology groups $H_k(\Lambda_r; \mathbb{Z}_2) := \ker \partial_k / \operatorname{im} \partial_{k+1}$ that carry induced $\mathbb{T}_{24}$-actions.

Proof. By $\mathbb{Z}_2$-linearity it suffices to verify the identity on a basis face $\{\vec{v}_a, \vec{v}_b, \vec{v}_c\} \in F_r$:

$$\begin{aligned} \partial_1\big(\partial_2(\{\vec{v}_a, \vec{v}_b, \vec{v}_c\})\big) &= \partial_1\big(\{\vec{v}_a, \vec{v}_b\} + \{\vec{v}_b, \vec{v}_c\} + \{\vec{v}_a, \vec{v}_c\}\big) \\ &= (\vec{v}_a + \vec{v}_b) + (\vec{v}_b + \vec{v}_c) + (\vec{v}_a + \vec{v}_c) \\ &= 2\vec{v}_a + 2\vec{v}_b + 2\vec{v}_c = 0 \text{ in } \mathbb{Z}_2. \end{aligned}$$

Equivariance of the homology groups under $\mathbb{T}_{24}$ then follows from Proposition 5.25, since each $\rho_k(g)$ is a linear isomorphism on $C_k(\Lambda_r; \mathbb{Z}_2)$ that preserves $\ker \partial_k$ and $\operatorname{im} \partial_{k+1}$ and therefore descends to an action on the quotient. □

Remark 5.27. *The quotient in H_k is key to this entire construction. Two cycles represent the same homology class exactly when they differ by a boundary, that is, by an element of $\operatorname{im} \partial_{k+1}$. Thus, taking a cycle modulo $\operatorname{im} \partial_2$ implies "XOR-ing away" any fillable region and keeping only what cannot be filled, which is the familiar move of quotienting by an equivalence: discard the detailed information that does not matter (e.g., loops you can cap off with faces) and retain the key invariant information that does matter. The identity $\partial_1 \circ \partial_2 = 0$ is what makes this well-defined, and it has a one line geometric reading, namely that the rim of a filled region has no endpoints, so every boundary is automatically a cycle. For a concrete instance on Λ_1, let's take the hexagon cycle $[\Lambda_{T,1}]$ and XOR in the boundary $\partial_2 f$ of a single triangular face f. The result is*

a different 1-chain, but it lies in the same H_1 class as $[\Lambda_{T,1}]$, which is precisely because the two differ by the element $\partial_2 f \in \operatorname{im}\partial_2$.

Example 5.28 (A $\mathbb{Z}_2$ Link-State Labeling on a Networked Device Cluster). Consider a cluster of communicating devices placed at the vertices of a truncated Λ_1^R, with a bidirectional communication link along each edge and no device at the origin (the puncture models an absent or failed central hub). A snapshot of which links are currently carrying traffic is recorded as a 1-chain $x \in C_1(\Lambda_r;\mathbb{Z}_2)$ whose active links are the edges with coefficient 1. The boundary calculus, the ι_1-action, and the puncture class each contribute something distinct to reasoning about such a snapshot.

(1) **Layout.** Take admissible $r = 1$. The six boundary devices sit at the unit Eisenstein vertices $V_{T,1} = \{\vec{v}_0, \ldots, \vec{v}_5\}$ at phases $t\pi/3$ for $t \in \mathbb{Z}_6$, each with integer norm $a^2 + ab + b^2 = 1$, and together they form the regular hexagon $\Lambda_{T,1}$. The outer devices occupy $V_{+,1}$ (norm > 1), and each one has a unique inner twin in $V_{-,1}$ that is obtained via its exact inversion image under ι_1, so $|\Lambda^R_{+,1}| = |\Lambda^R_{-,1}|$.

(2) **Closed loop test via ∂_1.** Suppose the active links are exactly the six hexagon edges $\{\vec{v}_t, \vec{v}_{t+1}\}$ with indices mod 6, so $x = [\Lambda_{T,1}]$. Applying the boundary operator gives

$$\partial_1 x = \sum_{t\in\mathbb{Z}_6} (\vec{v}_t + \vec{v}_{t+1}) = 2\sum_{t\in\mathbb{Z}_6} \vec{v}_t = 0 \text{ in } \mathbb{Z}_2,$$

because every boundary device is the endpoint of exactly two active links and $1 + 1 = 0$. A vanishing boundary $\partial_1 x = 0$ implies that x is a closed routing configuration with no loose ends, that is, $x \in \ker\partial_1$. Adding or dropping a single link would make $\partial_1 x$ precisely indicate the existence of two remaining devices with an odd number of active links.

(3) **Orbit dispatch via ι_1.** A reachability or reliability computation over the outer devices does not need to be repeated on the inner side. The inversion ι_1 pairs each outer link with its corresponding inner twin link and fixes the six boundary devices pointwise (Remark 5.30), so a result computed on $\Lambda^R_{+,1}$ transports verbatim to $\Lambda^R_{-,1}$, which only leaves the ι_1-fixed locus $V_{T,1}$ to be handled once on its own. Therefore, an implementation splits the work into one outer zone pass, a transport step to the inner zone, and a single boundary pass, where no cross zone message exchange is needed to reconcile the two halves.

(4) **Local loop versus loop around the missing hub.** Two closed configurations can look alike yet mean different things. For instance, a loop that bounds a single triangular region is the ∂_2-image of that face, so it lies in $\operatorname{im}\partial_2$ and represents the trivial class in $H_1(\Lambda_r;\mathbb{Z}_2)$: it does not encircle a missing device. The hexagon $x = [\Lambda_{T,1}]$ bounds no face (Remark 5.31) and represents the nontrivial generator, so a configuration equal to $[\Lambda_{T,1}]$ modulo $\operatorname{im}\partial_2$ is one whose active links circulate all the way around the absent hub. Thus, testing a closed configuration against the generator separates the traffic that merely loops within a finite region from the traffic that wraps around the unreachable center, a key distinction that plain adjacency bookkeeping cannot make.

The same three ingredients—the boundary test, the ι_1-orbit split, and the puncture class—reappear for any $\mathbb{Z}_2$-valued labeling on Λ_r, which is the substance of the three remarks that follow.

Remark 5.29. *We work with $\mathbb{Z}_2$ coefficients throughout this subsection so that the 2-complex on Λ_r carries no orientation data. The boundary operators are then well-defined on unordered cells (vertex sets, edge sets, face sets), and the $\mathbb{T}_{24}$-action is automatically orientation-free. Extending to $\mathbb{Z}$ or $\mathbb{Z}[\mathbb{T}_{24}]$ coefficients would require a coherent orientation on the faces, namely a choice of cyclic ordering of each face's vertex triple, which the edge framing of Definition 3.39 supports but which lies beyond the present chain-complex treatment. Here, the advantage for implementations is that any algorithm that traverses or modifies the boundary calculus on Λ_r can ignore orientation entirely, because both the equivariance and the boundary identity hold without any choice of sign convention.*

Remark 5.30. *The central involution ι_r also organizes the $\mathbb{Z}_2$-homology groups $H_k(\Lambda_r;\mathbb{Z}_2)$, but not through the even/odd eigenspace splitting that one would use over a field of characteristic zero. That splitting is unavailable over $\mathbb{Z}_2$ because $+1 = -1$ collapses the two involution-eigenspaces into one. Instead, the chain-level action $\rho_k(\iota_r)$ induces a canonical short exact sequence on each chain space as*

$$0 \longrightarrow \operatorname{Im}(\rho_k(\iota_r)+\mathrm{id}) \longrightarrow \ker(\rho_k(\iota_r)+\mathrm{id}) \longrightarrow \widehat{H}_k(\langle\iota_r\rangle;C_k) \longrightarrow 0,$$

where $\widehat{H}_k(\langle\iota_r\rangle;C_k)$ is the Tate cohomology of the order-2 action on C_k over $\mathbb{Z}_2$. It is known that $\widehat{H}_k$ is the algebraic ledger of that order two symmetry, which separates the chains that ι_r leaves fixed, from the chains that ι_r swaps (e.g., pairs are swapped). This Tate cohomology refines the bookkeeping that an integral-coefficient eigendecomposition would provide: the ι_r-fixed chains, the ι_r-orbit sums, and their combination split the cycle, cut, and boundary spaces into independent symmetry-reduced subspaces that remain compatible with the full actions of $\mathbb{T}_{24}$, because $\langle\iota_r\rangle$ commutes with D_6. The practical consequence matches the integral case. Any $\mathbb{Z}_2$-chain algorithm that respects the ι_r-action—one operating on ι_r-orbits (the paired outer/inner cells) or on the ι_r-fixed locus $V_{T,r}$—decomposes into independent subproblems that carry the symmetry-reduction benefits of the full actions of $\mathbb{T}_{24}$. This is the orbit split that is exercised in Example 5.28, where an implementation dispatches one pass over the ι_r-orbits and one pass over $V_{T,r}$ across the processors with no redundant communication between the two zones.

Remark 5.31. *Removing the origin from the plane punches a hole, and a loop that encircles that hole can neither be shrunk to a point nor filled in by triangular faces. Such a loop is the* puncture class*: it lets a $\mathbb{Z}_2$-labeling detect whether a configuration wraps the absent center, which is a global fact that the local adjacency bookkeeping cannot see. Because Λ_r lives on the origin-punctured plane $X = \mathbb{C}\setminus\{(0,0)_C\}$, the cycle space $\ker\partial_1$ carries a topologically nontrivial class: any cycle that encloses the origin represents it. The minimal admissible case $r = 1$ directly shows the simplest of such cycles. There the boundary subgraph $\Lambda_{T,1}$ is itself a regular hexagon through the six vertices of $V_{T,1}$, and its indicator chain $[\Lambda_{T,1}] \in C_1(\Lambda_r;\mathbb{Z}_2)$ lies in $\ker\partial_1$ while staying out of $\operatorname{im}\partial_2$, since no single face has $\Lambda_{T,1}$ for its $\mathbb{Z}_2$-boundary. So $[\Lambda_{T,1}]$ is a canonical generator of an $H_1(\Lambda_r;\mathbb{Z}_2)$ class that is pointwise fixed by ι_r (which thus pointwise fixes $V_{T,r}$) and equivariantly permuted under the spatial D_6-action. That makes it the natural $\mathbb{T}_{24}$-invariant homological anchor for any $\mathbb{Z}_2$-labeling that has to keep track of a topologically nontrivial class around the puncture, and it complements the geometric role of $\Lambda_{T,r}$ as the ι_r-fixed separator (Corollary 4.10). Larger admissible r carry analogous generators, which are built from cycles in $\Lambda_{+,r}$ or $\Lambda_{-,r}$ that enclose the origin, but the minimal case is the cleanest to identify. Step 4 of Example 5.28 puts this class to work by distinguishing between a cycle that bounds a finite region and a cycle that winds around the punctured origin, which is precisely the cohomological obstruction that a consistent $\mathbb{Z}_2$-labeling scheme must detect.*

Remark 5.32. *The indicator vectors of the six trihexagonal color classes (Definition 5.10) all lie in $C_0(\Lambda_r;\mathbb{Z}_2)$, and their $\mathbb{Z}_2$-span is a $\mathbb{T}_{24}$-invariant 6D subspace of C_0, on which the action permutes the six basis indicators up to the proper-coloring permutation of Theorem 5.15. More generally, Proposition 5.25 together with Corollary 5.26 places every vertex, edge, and face $\mathbb{Z}_2$-labeling on Λ_r inside one equivariant homological picture, so an algorithm that is built on cycle space or cut space decompositions—for network reliability [49], planar duality [21], or cycle basis cover construction—picks up the same symmetry reduction that the TQF already gives vertex-based computations. This dualizing carries this to the cochain side: the cochain complex $(C^\bullet(\Lambda_r;\mathbb{Z}_2),\partial^\bullet)$ inherits the $\mathbb{T}_{24}$-equivariance, the natural pairing $C_k \otimes C^k \to \mathbb{Z}_2$ is $\mathbb{T}_{24}$-invariant, and the dual cell labelings (vertex/edge/face against face/edge/vertex).*

6 Simulation Experiments and Runtime Performance

In this section, we conduct experiments and present the resulting empirical benchmarks to demonstrate the practical efficiency gains of the TQF through its core dualities and symmetries. Specifically, we evaluate three representative applications:

- inversion-based path mirroring, which exploits the Escher reflective duality for reversible zone swapping;
- symmetry-reduced clustering, which exploits the order-6 rotational symmetry for orbit-based computations; and
- conflict-free data parallel relaxation, which exploits the equivariant trihexagonal six-coloring to schedule independent updates directly onto GPU hardware.

These simulations use truncated radial dual triangular lattice graphs Λ_1^R—where the admissible $r = 1$ yield a symmetric hexagonal boundary with $|\Lambda_{T,1}^R| = 6$ vertices—with varying truncation radii $R \gg r$ to approximate the infinite structure while preserving exact bijections.

A word on what "exact" means across these benchmarks, since exactness—rather than raw speed—is TQF's central claim. Every framework-level quantity that is exercised below is *combinatorial* or integer-valued, and is computed bitwise-exactly: the shortest-path hop counts (the discrete dual metric), the zone-membership and boundary tests (integer Eisenstein-norm comparisons $a^2 + ab + b^2$ against r^2), the angular-sector index and the trihexagonal six-coloring that it feeds (the sector index is decided from the integer coordinate pair (a, b) via the sign of the lattice cross product, and the color residue is $c = (a - b) \bmod 3$—both exact integer operations, with no floating-point phase), and the inner/outer vertex bijection. The one real-valued aggregate we report—the average local clustering coefficient—is itself an exact rational number, and we compute it in exact rational arithmetic (`fractions.Fraction`), so the orbit-reduced and full graph values agree not merely to a floating-point tolerance but as the *identical* rational (verified by an exact `==` check at every R). Floating-point arithmetic enters in exactly one place, the relaxation kernel of the six-coloring benchmark, where the CPU and GPU backends are verified to agree to $\sim 10^{-13}$—a property of the diffusion update itself, not of the framework's combinatorial structure. Below, each measured speedup is therefore an exactness-preserving elimination of redundant work, not an approximation.

All benchmarks were implemented in Python. The path mirroring and clustering benchmarks (Tables 12 and 13) use NetworkX for graph operations and NumPy for the vectorized orbit construction, and were executed on a laptop with an Intel Core i7-14650HX CPU (16 GB DDR4 RAM). The trihexagonal six-coloring benchmark (Table 14) additionally uses a CUDA-enabled PyTorch build for the GPU backend, which is executed on the same laptop's NVIDIA GeForce RTX 4060 Laptop GPU (8 GB). Run counts, inner loop timing repeats, and—for the six-coloring benchmark—the number of independent sessions are noted per table. These runtimes are approximate and may vary by hardware, where the absolute CPU times scale as $O(R^2)$.

6.1 Inversion-Based Path Mirroring Benchmarks

To demonstrate TQF's practical utility, we simulate an inversion-based path mirroring benchmark experiment that exploits the Escher reflective duality from Subsection 4.2. As established in Proposition 4.15 and Corollary 4.24, the circle inversion bijection ι_r enables direct path mirroring without recomputation. This approach leverages the phase pair constancy along graph rays (Subsection 3.2) to map paths from the truncated outer zone subgraph $\Lambda_{+,r}^R$ to the truncated inner zone subgraph $\Lambda_{-,r}^R$, which bypasses the need to recompute results for the latter in symmetric balanced finite truncations of Λ_r^R and yields a measured 1.6–1.8x speedup under the discrete dual metric, with the mirrored result verified bitwise-identical to full recomputation.

For admissible $r = 1$ consider truncated Λ_1^R (recalling that $|\Lambda_{+,1}^R| = |\Lambda_{-,1}^R|$ exactly by the bijective construction of Subsection 3.3, and $|\Lambda_{+,1}^R| \sim O(R^2)$ asymptotically, which excludes $\Lambda_{T,1}^R$). The standard approach computes single source shortest paths (e.g., from a random start vertex) in $\Lambda_{+,1}^R$, then recomputes them in $\Lambda_{-,1}^R$ from the corresponding inverted start vertex $\iota_r(\text{start})$. (Note: For weighted graphs, the continuous dual metric from Subsection 4.4 can normalize the edge lengths, which extends this to geometric applications like lattice-based optimization in operations research.) The TQF approach computes single source shortest paths in $\Lambda_{+,1}^R$, then quickly reflects/mirrors them directly to $\Lambda_{-,1}^R$ via ι_1 (as per Proposition 4.38 and the induced graph isomorphism of Corollary 4.21).

ALGORITHM PSEUDOCODE 7. Inversion-Based Path Mirroring via Bijection

```
function MirrorPaths(outer_paths, inversion_map):
    # outer_paths is dictionary of target vertices in outer subgraph
    # via discrete dual metric
    mirrored = {}  # Dictionary for reflected paths in inner subgraph
    for target, length in outer_paths.items():
        if target in inversion_map:
            inv_target = inversion_map[target]
            mirrored[inv_target] = length  # Preserve hop count via discrete dual metric
    return mirrored
```

Intuitively, the circle inversion map ι_1 reflects $\Lambda^R_{+,1}$'s shortest-path lengths (under the discrete dual metric) to $\Lambda^R_{-,1}$ via the Escher reflective duality, which preserves phase constancy along the graph rays and tracks the hop distances without recomputing a full BFS on $\Lambda^R_{-,1}$.

We implemented our two Python simulation scripts `simulation_02_benchmark_standard_path_mirroring.py` and `simulation_03_benchmark_triquarter_path_mirroring.py` in Appendix C, which are freely available online [56]. To ensure the comparison is fair, both scripts solve the identical dual zone problem, and the TQF script verifies that its inversion mirrored inner zone result is bitwise-identical to an independent recomputation before timing. Benchmarks (averaged over 20 runs, each with 100 inner loop timing repeats) for $R = 5, 10, 15, 20, 25$ show a consistent 1.6–1.8x speedup (Table 12) to reflect the elimination of the second single source traversal—the theoretical ceiling is 2x, where the gap attributable to the $O(|\Lambda^R_{+,1}|)$ construction of the mirrored distance map is what replaces that traversal. While tested up to $R = 25$ (with $|\Lambda^{25}_{-,1}| = |\Lambda^{25}_{+,1}| = 2256$ per zone), extensions to larger R or noisy graphs (e.g., with random edge perturbations) would likely maintain the relative speedups due to the preservation of symmetries under the Escher reflective duality (Proposition 4.15), though absolute times scale as $O(R^2)$.

TABLE 12. Inversion-Based Path Mirroring Benchmark Comparisons (Laptop)

R	$\lvert\Lambda^R_{-,1}\rvert = \lvert\Lambda^R_{+,1}\rvert$	**Standard Time (ms)**	**TQF Time (ms)**	**Speedup**
5	84	0.054 ± 0.006	0.030 ± 0.000	1.8x
10	360	0.263 ± 0.002	0.149 ± 0.001	1.8x
15	816	0.628 ± 0.010	0.380 ± 0.009	1.7x
20	1452	1.164 ± 0.017	0.678 ± 0.014	1.7x
25	2256	1.831 ± 0.020	1.100 ± 0.011	1.7x

Note. Zone counts exclude the boundary vertices of $|\Lambda^R_{T,1}| = 6$ (for admissible $r = 1$). Benchmarks averaged over 20 runs with 100 timing repeats each. Executed on a laptop with an Intel Core i7-14650HX CPU (16 GB DDR4 RAM). The TQF approach was verified to produce inner zone distances bitwise-identical to independent recomputation at every R. The theoretical ceiling is 2x (one of two equal traversals eliminated). The measured 1.6–1.8x reflects the $O(|\Lambda^R_{+,1}|)$ cost of the mirrored distance map. The absolute times scale as $O(R^2)$.

6.2 Symmetry-Reduced Clustering Benchmarks

To further illustrate the practical utility of the TQF by exploiting its order-6 rotational symmetry (as detailed in Subsection 4.1), we benchmark symmetry-reduced computations of the average local clustering coefficient on truncated Λ^R_r. This approach leverages a $\mathbb{Z}_6$-orbit transversal (built by traversing vertices and skipping those reachable from earlier representatives via the $\mathbb{Z}_6$ action) to compute the coefficient on one representative per orbit and replicate the result across the orbit via the group action, which bypasses the

need to recompute the remaining $\sim 5/6$ of Λ_r^R. As established in Lemma 3.10 and Corollary 4.6, the $\mathbb{Z}_6$ orbits enable this direct replication while preserving phase pair constancy along graph rays, which ensures exact consistency across orbits in symmetric, balanced finite truncations of Λ_r^R. Our simulations (detailed in Subsection 6.2 and Table 13) observe a steady $\sim$3.5–4.0x speedup (peaking near $\sim$4.0x at $R = 50$), below the idealized $\sim$6x $\mathbb{Z}_6$-symmetry ceiling—the gap reflecting the uniform per-vertex cost of exact rational arithmetic together with the TQF approach's own per-vertex constants (orbit-transversal construction and replication bookkeeping)—while the orbit-replicated coefficient reproduces the full graph rational coefficient *exactly* (bitwise) at every R.

Again, with admissible $r = 1$, consider truncated Λ_1^R (recalling that $|\Lambda_{+,1}^R| = |\Lambda_{-,1}^R|$ exactly by the bijective construction of Subsection 3.3, and $|\Lambda_{+,1}^R| \sim O(R^2)$ asymptotically, which excludes $\Lambda_{T,1}^R$). The standard approach computes local clustering coefficients (triangle density in neighborhoods) for all vertices in Λ_1^R and loops over the full vertex set. The TQF approach precomputes the $\mathbb{Z}_6$-orbit transversal ($O(|\Lambda_1^R|)$ once), computes the coefficients on one representative per orbit ($\sim|\Lambda_1^R|/6$), and then replicates via rotations (preserving exact values under equivariance, as per Theorem 5.4).

ALGORITHM PSEUDOCODE 8. Symmetry-Reduced Clustering via Orbit Replication

```
function OrbitReducedClustering(graph $\Lambda_r$, orbits):
    # Precompute: orbits = get_symmetry_orbits($\Lambda_r$)  # O(|V|) once
    total_clust = 0
    total_weight = 0
    for orbit in orbits:
        rep = orbit[0]  # Representative
        neigh = neighbors(rep)  # O(deg) = O(1)
        if |neigh| < 2: continue
        common = 0
        for i,j in combinations(neigh, 2):  # O(deg^2) = O(1)
            if has_edge(i, j): common += 1
        clust_rep = (2 * common) / (|neigh| * (|neigh| - 1))  # Local coeff
        orbit_size = |set(orbit)|  # ~6, < for fixed points
        total_clust += clust_rep * orbit_size
        total_weight += orbit_size
    return total_clust / total_weight  # Average
```

The $\mathbb{Z}_6$ action replicates clustering values across orbits, which preserves the invariant without recomputing on the full graph.

We implemented our two Python simulation scripts `simulation_04_benchmark_standard_clustering.py` and `simulation_05_benchmark_triquarter_clustering.py` in Appendix D, which are freely available online [56]. Both scripts compute the average local clustering coefficient in exact rational arithmetic (`fractions.Fraction`)—because the order-6 rotation is a graph automorphism of Λ_r^R, every vertex in a $\mathbb{Z}_6$ orbit shares the identical exact rational local coefficient, so the orbit-replicated average reproduces the full graph average not just to floating-point tolerance but as the *same rational number*. At every truncation radius, the script verifies this by an exact `==` comparison of the two `Fraction` results (PASS at all R), which confirms that the symmetry reduction is lossless. Benchmarks (averaged over 20 runs) for $R = 10, 20, 50, 100, 150, 200$ show a consistent $\sim$3.5–4.0x speedup, which peak near $\sim$4.0x at $R = 50$ (Table 13)—see Subsection 6.4 for the asymptotic interpretation. The orbit transversal is a one time precomputation that can be built either in pure Python or with NumPy-vectorized rotation matrices; the two constructions yield bitwise-identical orbits, and at $R = 200$ the NumPy-vectorized build completes in $\sim$416 ms. While tested up to $R = 200$ ($|\Lambda_1^{200}| \approx 290094$), extensions to larger R or noisy graphs (e.g., with random edge perturbations) would likely maintain the relative speedups due to the preservation of symmetries under the $\mathbb{Z}_6$ action (Lemma 3.10), though absolute times scale as $O(R^2)$.

TABLE 13. Symmetry-Reduced Clustering Coefficient Benchmark Comparisons (Exact Rational Arithmetic, Laptop)

R	$\lvert\Lambda_1^R\rvert$	**Standard Time (ms)**	**TQF Time (ms)**	**Speedup**
10	726	1.50 ± 0.02	0.39 ± 0.02	3.9x
20	2910	6.16 ± 0.09	1.56 ± 0.04	3.9x
50	18114	40.71 ± 0.37	10.07 ± 0.20	4.0x
100	72582	161.00 ± 1.25	45.41 ± 1.15	3.5x
150	163242	371.28 ± 1.70	104.91 ± 1.48	3.5x
200	290094	693.53 ± 4.28	189.61 ± 1.01	3.7x

Note. Boundary vertices $\lvert\Lambda_{T,1}^R\rvert = 6$ are included in the full graph $\lvert\Lambda_1^R\rvert$. Both approaches compute the average local clustering coefficient in exact rational arithmetic (`fractions.Fraction`), so each reported value is an exact rational. At every R, the orbit-replicated coefficient equals the full graph coefficient *bitwise* (verified by exact `==` comparison, not a floating-point tolerance), which confirms that the symmetry reduction is lossless. For example, the exact coefficients are 1788/4235 at $R = 10$ and 2036351/5076645 at $R = 200$. Benchmarks averaged over 20 runs and were executed on a laptop with an Intel Core i7-14650HX CPU (16 GB DDR4 RAM). The idealized $\mathbb{Z}_6$ ceiling is ∼6x—the measured speedups are a steady ∼3.5–4.0x (peaking near 4.0x at $R = 50$)—the gap reflects the uniform per-vertex cost of exact rational arithmetic together with the orbit-transversal and replication bookkeeping. Orbit-transversal construction is a one time precomputation excluded from the timed region. The absolute times scale as $O(R^2)$.

6.3 Conflict-Free Parallel Processing via the Trihexagonal Six-Coloring

The path mirroring and clustering benchmarks above exploit TQF's dualities and rotational symmetry on a single processor. We now demonstrate that the equivariant trihexagonal six-coloring e_6 (Definition 5.10) translates TQF's symmetry directly into data parallelism on a GPU. By Corollary 5.13, e_6 is a proper six-coloring of Λ_r, so each of its six color classes is an independent set: no two vertices in the same class are adjacent. Independence is precisely the property a parallel scheduler requires, since vertices in one class can be updated simultaneously with no read-write conflicts and no locks. Because e_6 is $\mathbb{T}_{24}$-equivariant (Theorem 5.15), the partition into conflict-free classes is an exact structural property of Λ_r rather than the output of a general purpose graph coloring heuristic.

To exercise this, we benchmark one color-ordered relaxation sweep over Λ_1^R, where every vertex updates its scalar state to a weighted combination of its own state and the mean of its neighbors' states. This is the computational kernel shared by graph diffusion, iterative smoothing, label propagation, and the message-passing layers of graph neural networks [75]. The sweep visits the six color classes in turn, such that the updates are mutually independent within each class (by Corollary 5.13) and are issued as a single batched, vectorized operation. The color classes are thus the unit of parallelism.

ALGORITHM PSEUDOCODE 9. Conflict-Free Color-Ordered Relaxation Sweep

```
function ColorOrderedSweep(graph $\Lambda_r$, state, color_classes):
    # color_classes = partition of V by e_6 into 6 independent sets
    for class in color_classes:            # 6 classes, visited in turn
        # All vertices in this class are mutually non-adjacent, so
```

```
        # their updates are independent and run as one batched op
        neighbor_mean[class] = mean over neighbors of state   # gather + reduce
        state[class] = alpha * state[class] \
                       + (1 - alpha) * neighbor_mean[class]
    return state
```

We implemented this benchmark in the Python script `simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py` in Appendix E, which is freely available online [56]. The script compares two backends on the identical workload: a single threaded CPU baseline using NumPy array operations versus a GPU backend using PyTorch tensor operations. Before timing, the script verifies that e_6 is a proper coloring on the constructed graph and that the GPU result agrees with the CPU result to within floating-point tolerance—both checks pass at every R. The results are reported in Table 14.

TABLE 14. Trihexagonal Six-Coloring Conflict-Free Parallel Benchmark Comparisons (Laptop CPU vs. GPU with medians over five sessions)

R	$\lvert\Lambda_1^R\rvert$	$\lvert E\rvert$	**CPU (ms/sweep)**	**GPU (ms/sweep)**	**Speedup (med; range)**
50	18114	53658	0.68	1.19	0.54x (0.28–0.57)
100	72582	216366	2.43	0.94	2.59x (2.28–2.72)
150	163242	487650	5.72	1.03	5.52x (5.23–7.00)
200	290094	867522	11.53	1.39	8.04x (6.65–8.64)

Note. Boundary vertices $\lvert\Lambda_{T,1}^R\rvert = 6$ are included in $\lvert\Lambda_1^R\rvert$. CPU and GPU columns report the *median* milliseconds per relaxation sweep over five independent sessions (each session: 10 runs × 20 inner loop repeats × 10 sweeps). The last column gives the median session speedup with its observed min–max range. CPU baseline: single threaded NumPy on an Intel Core i7-14650HX; GPU backend: NVIDIA GeForce RTX 4060 Laptop GPU (8 GB) with CUDA-enabled PyTorch. The six-coloring was verified proper and the CPU/GPU results were verified in agreement (to $\sim 10^{-13}$) in all 20 invocations. The GPU runtime stays nearly flat as the graph grows (the device is far from saturated) while the CPU cost scales linearly, so the speedup widens with scale. At $R = 50$ the GPU is slower because its fixed launch and transfer overhead is not yet amortized. Per-session variance (notably session-to-session CUDA context warm up and thermal state) is the reason for reporting medians and ranges rather than a single run.

The benchmark exhibits a clean crossover-and-scaling pattern. At $R = 50$ (∼18k vertices) the GPU is slower than the CPU (median 0.54x): the fixed cost of launching kernels and moving data to the device has not yet been amortized. As the graph grows, the CPU sweep time scales linearly with the vertex and edge counts (median 0.68 → 11.53 ms/sweep from $R = 50$ to $R = 200$), whereas the GPU sweep time stays nearly flat (∼0.9–1.4 ms/sweep), because even the largest tested graph (∼290k vertices, ∼868k edges) leaves the device far from saturated. Consequently, the speedup grows monotonically past the crossover near ∼100k vertices to hit a median of 8.0x at $R = 200$ (range 6.6–8.6x across the five sessions). The flat GPU profile indicates the advantage would continue to widen at larger R, which is ultimately bounded by the device memory (8 GB) rather than by the framework: the six-coloring supplies as much conflict-free parallelism as the hardware can absorb. This result is the practical payoff of the equivariant encoding—TQF's exact group structure becomes, with no approximation, a partition that consumer GPU hardware can directly execute. We emphasize that the baseline is a single threaded CPU implementation, so the comparison measures the

data parallelism that the exact conflict-free coloring unlocks on commodity GPU hardware, not a speedup over a maximally optimized multi-core baseline.

6.4 Benchmark Results and Discussion

The simulation results validate the theoretical dualities and symmetries of the TQF through empirically observed efficiency gains across three complementary regimes. In the inversion-based path mirroring benchmarks (Subsection 6.1), the Escher reflective duality (Proposition 4.15) yields consistent 1.6–1.8x speedups across truncation radii $R = 5$ to 25 while preserving exact bijections under the circle inversion map ι_r—and, crucially, the mirrored result is verified to be bitwise-identical to full recomputation, so the speedup is an exactness-preserving elimination of redundant work rather than an approximation. The headline figure here is correctness as much as speed: the duality lets one of the two equal traversals be replaced by an $O(|\Lambda^R_{+,1}|)$ map with no information loss, which caps the achievable speedup at 2x and explains the measured 1.6–1.8x speedup once the cost of constructing that map is accounted for.

In the symmetry-reduced clustering benchmarks (Subsection 6.2), the $\mathbb{Z}_6$ orbits (Lemma 3.10 and Corollary 4.6) let the expensive computation run on a single representative per orbit and replicate via rotational bijections. Because both the full graph and orbit-reduced computations are carried out in exact rational arithmetic and the order-6 rotation is a graph automorphism, the orbit-replicated average clustering coefficient equals the full graph coefficient *as the same exact rational number* at every R (verified by exact == comparison, not a floating-point tolerance). The measured speedup is a steady ∼3.5–4.0x, that peaks near ∼4.0x at $R = 50$, which is below the idealized $\mathbb{Z}_6$ ceiling of ∼6x. That ∼6x theoretical upper bound is based on the symmetry argument with zero implementation overhead. The measured ∼3.5–4.0x is what remains after two costs are paid: the uniform per-vertex cost of the exact rational arithmetic, and the TQF approach's own constant factors from the orbit transversal construction and the per-orbit replication bookkeeping. The orbit transversal is a one time precomputation, so its cost amortizes quickly. At $R = 200$, the ∼416 ms vectorized build is repaid after roughly one clustering query, which saves ∼500 ms over the baseline against each subsequent query.

The trihexagonal six-coloring benchmark (Subsection 6.3) is where the framework's symmetry delivers its largest practical payoff. Because e_6 partitions Λ_r into six conflict-free independent sets (Corollary 5.13) by an exact, equivariant rule (Theorem 5.15), a color-ordered relaxation sweep maps directly onto data parallel hardware with no locks and no approximation. On a consumer NVIDIA GeForce RTX 4060 laptop GPU, the speedup over a single threaded CPU baseline grows monotonically past a crossover near ∼100k vertices to reach a median 8.0x at ∼290k vertices (range 6.6–8.6x across five independent sessions), where the GPU runtime remains nearly flat while the CPU cost scales linearly. This is the qualitative jump that the encodings were designed to unleash: the same exact group structure that yields bounded constant factor gains on a single core becomes, on parallel hardware, an advantage that widens with problem size and is limited by the device rather than the framework.

These experimental findings underscore TQF's value in symmetric networks and lattice-based models, such as efficient dual zone queries, motif analysis, and path optimizations. The applications closest to what we benchmark are:

- In network optimization [67], the D_6-invariant symmetric separator $\Lambda_{T,r}$ is a constant-size, perfectly balanced inner/outer vertex cut (Theorem 3.28, Corollary 3.30), so cut-based and flow-based routines across the two zones only pay a separation overhead cost that is independent of R—useful for repeated min-cut or community detection queries on evolving topologies like web graphs, power grids, or social networks.

- In motif analysis on symmetric graphs (e.g., biological networks [1]), the $\mathbb{Z}_6$ orbit replication benchmarked in Subsection 6.2 computes clustering coefficients on one representative per orbit and replicates exactly, specifically the lossless reduction demonstrated above.

- In graph neural networks [75], the trihexagonal six-coloring (Definition 5.10) schedules the equivariant message-passing kernel as the conflict-free color-ordered sweep benchmarked in Subsection 6.3, which supports data parallel inductive learning on lattice-structured graphs.

Advancing onward, the same structure has natural but as-yet-untested connections to lattice-based cryptography [76], symmetry-invariant signal and image processing on lattice signals [29, 30], robotics path planning via reflective zone mirroring, and multi-agent coordination [48]. These are prime directions for future work.

On the other hand, the observed speedups assume balanced zones ($|\Lambda^R_{-,r}| = |\Lambda^R_{+,r}|$ ensured by bijective self-duality of Proposition 4.38) and idealized symmetry under $\mathbb{T}_{24}$ (Definition 4.16), which may not hold in noisy real-world graphs with perturbations deviating from Λ^R_r's ideal symmetries. Thus, further validation on diverse datasets (e.g., graphs with edge weight perturbations or irregular vertex distributions) and hardware configurations is needed to confirm broader applicability because these benchmarks are preliminary and focused on controlled settings.

6.5 Per Operation Cost Model

Table 15 summarizes the asymptotic cost and the arithmetic model of TQF's core operations. The recurring pattern is that every structural query is either constant-time or constant-time per vertex, and is carried out in exact integer or exact rational arithmetic rather than floating-point. (Floating-point only enters the optional relaxation kernel.) Consequently, the only term that scales with graph size is a single $O(|V|)$ orbit-transversal precomputation, which is built once and amortized across all subsequent symmetry-reduced queries (Subsection 6.4).

TABLE 15. Per Operation Cost and Arithmetic Model of Core TQF Operations

Operation	**Cost**	**Arithmetic / Guarantee**
Zone membership and boundary test (per vertex)	$O(1)$	exact integer (Eisenstein norm $a^2 + ab + b^2$ vs. r^2)
Angular-sector index (per vertex)	$O(1)$	exact integer cross product on (a, b); sectors exactly uniform
Circle inversion ι_r (per vertex)	$O(1)$	exact rational, coordinates in $\mathbb{Q}(\sqrt{3})$
Trihexagonal six-coloring e_6 (per vertex)	$O(1)$	exact integer; proper by Corollary 5.13, exactly $\mathbb{T}_{24}$-equivariant
Inner/outer vertex bijection (whole graph)	$O(\|V\|)$	exact; balanced, $\|V^R_{-,r}\| = \|V^R_{+,r}\|$
Balanced inner/outer separator $V_{T,r}$	size $6k$, R-independent	exact integer; Theorem 3.28
$\mathbb{Z}_6$ orbit-transversal build (one time)	$O(\|V\|)$, once	exact integer; Lemma 3.10
Symmetry-reduced clustering coefficient	$O(\|V\|/6)$ + replication	exact rational (`Fraction`); bitwise-lossless
Color-ordered relaxation sweep	$O(\|V\| + \|E\|)$ per sweep, six parallel classes	floating-point (diffusion kernel only)

Note. The "per vertex" costs are the work for a single vertex. When applied across the graph they are $O(|V|)$ but embarrassingly parallel and free of floating-point error. The separator size $6k$ depends only on the admissible radius r (via the quadratic-form representations of r^2), not on the truncation radius R. Floating-point only enters in the relaxation sweep, where the CPU and GPU backends are verified to agree to $\sim 10^{-13}$.

Overall, these experiments and their empirical results validate TQF's potential for scalable, reversible computations, bridging theoretical symmetries with practical advancements in symmetry-aware algorithms, as we further discuss in the conclusion.

7 Conclusion

In this paper, we introduce the TQF, extended to discrete radial dual triangular lattice graphs Λ_r, with intent to forge a strong mathematical and computational foundation for symmetry-aware algorithms. By unifying complex-Cartesian-polar coordinates with phase-pair assignments and topological zones, we enable exact bijective mappings without approximations—rooted in the norm trichotomy—that harness combinatorial duality for radial separation, Escher reflective duality for zone swapping, and bijective self-duality for reversible transformations. These key properties endure under Λ_r's order-6 rotational symmetry, which power modular decompositions via angular sectors S_t and equivariant encodings, including the trihexagonal six-coloring for conflict-free parallel processing. Underpinning these advances is the Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group $\mathbb{T}_{24}$, which fuses dihedral actions with inversive bijections to bypass recomputation in scalable, symmetry-preserving designs.

Our contributions include:

- the radial dual triangular lattice graph Λ_r, which embeds origin-centered radial symmetries and exact zone bijections into the lattice by construction;
- formal proofs of the three dualities on Λ_r—combinatorial, Escher reflective, and bijective self-duality (Propositions 4.2, 4.15, and 4.38);
- the constant-size balanced separator theorem (Theorem 3.28), which shows that the boundary set $V_{T,r}$ is an R-independent, perfectly balanced vertex separator of size $6k$ splitting Λ_r^R into exactly its two zones, together with the truncation-stability guarantee (Lemma 3.45) that makes every finitely-supported result exact on each sufficiently large finite truncation Λ_r^R;
- the concrete realization of the *Tri-Quarter Inversive Hexagonal Dihedral Symmetry Group* $\mathbb{T}_{24} = D_6 \times \mathbb{Z}_2$—which unifies the framework's rotational, reflective, and inversive symmetries with ι_r as a central involution—contributed not as a new abstract group (it is the classical centrosymmetric hexagonal point group D_{6h}) but as a circle inversion action on Λ_r;
- the dual metrics (discrete and continuous) that remain invariant under inversion, and the equivariant encodings (sector indexing and the trihexagonal six-coloring) that drive symmetry-reduced parallel algorithms;
- a face trichotomy and a $\mathbb{T}_{24}$-equivariant $\mathbb{Z}_2$-chain complex on vertices, edges, and faces, which lets cycle space and cut space algorithms inherit the same symmetry-reduction benefits that are already established for vertex-based computations, and which exposes a homological obstruction (the H_1 puncture class) that detects whether a configuration wraps the origin; and
- exactness throughout the discrete layer, which takes three forms. Integer-only norm comparisons remove floating-point error from the admissibility, zone-membership, and boundary tests. An integer angular-sector index—a lattice cross product sign test on (a, b)—makes the sector partition exactly uniform and the trihexagonal six-coloring exactly $\mathbb{T}_{24}$-equivariant (Corollary 5.13, Theorem 5.15). And exact rational computation of the clustering coefficient makes the symmetry reduction bitwise-lossless.

Our simulations demonstrate the practical payoff across three regimes:

- *Inversion-based path mirroring* achieves 1.6–1.8x speedups, where we verified that the mirrored result is bitwise-identical to full recomputation.
- *Symmetry-reduced clustering* achieves a steady ∼3.5–4.0x speedup (below the idealized ∼6x $\mathbb{Z}_6$ ceiling), which we computed in exact rational arithmetic so that the orbit-reduced coefficient (bitwise) reproduces the full graph result at every R.
- *Conflict-free relaxation* scheduled by the trihexagonal six-coloring reaches a median ∼8x speedup (ranging 6.6–8.6x over five sessions) over a single threaded CPU baseline on a consumer NVIDIA GeForce RTX 4060 laptop GPU, where the advantage widens as the graph grows.

Together these underscore the TQF's efficiency in graph traversals, network optimizations, and data parallel computation.

This work advances scalable, exact computations on symmetric structures, with implications for tiling, robotics path planning, multi-agent coordination, computational geometry, signal processing, image processing, and more. While the focus remains on mathematical and computational foundations, it will be worthy to investigate potential future integrations with models that harness intricate, superposition-like symmetries in complex emergent systems with entangled behaviors across both classical and non-classical computing paradigms—targeting comprehensive symmetry-aware algorithms and data structures on diverse architectures.

The TQF has two main limitations. First, perfect symmetry relies on admissible radii. Second, finite truncations only approximate the infinite lattice, which leaves unresolved-area gaps near the origin that scale as $O(1/R^2)$. This second limitation is softened by Lemma 3.45: although the truncation leaves a geometric gap, every individual finite-support computation is still captured exactly at sufficiently large R. Next, we may explore applications that maintain the exact structure exploited here, namely weighted dual metrics for geometric optimization and equivariant radial dual lattice graph neural network layers that use the trihexagonal six-coloring as a conflict-free update schedule. Broader extensions—to higher dimensions, lattice-based cryptography, and physical lattice models—are plausible but remain speculative pending dedicated study.

Ultimately, the TQF establishes a rigorous mathematical and computational foundation for exploiting radial dualities and symmetries in discrete lattice graphs to drive the design and development of scalable, reversible algorithms that advance symmetry-aware computing paradigms with applications ranging from graph traversals to emergent complex system modeling.

Author's Note

I earned a B.S. in computer science and a minor in mathematics at Eastern Oregon University, and then a dual M.S. in computer science and mathematics at Boise State University. From 2007 to 2017, I engaged in research work in various capacities to apply computer science and mathematics to various fields such as machine learning, robotics, bioinformatics, high energy physics, quantum gravity, sustainable energy, and cryptography. I see much interdisciplinary overlap across these great fields. I'm currently working full-time as a software development engineer and doing some unpaid after-hours research as a hobby. This is my third math or science paper after taking an eight-year pause from research.

I originally came up with ideas for the TQF back in 2012 during quark confinement work [77], and additional ideas for discretization and applications up through 2017, but I never had the opportunity and realization to fully test, refine, and finalize them into a formalized framework until recently. It's been exciting to discretize the TQF for computer science with such potential for practical real-world applications.

Acknowledgement

I wish to thank the AI tools Grok (created by xAI) and Claude (created by Anthropic) for assistance with testing and refining some ideas for this paper. The original ideas and core contributions remain the author's own.

References

[1] Rubén J. Sánchez-García. Exploiting symmetry in network analysis. *Communications Physics*, 3(1):87, 2020. doi: 10.1038/s42005-020-0345-z.

[2] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing (PODC '85)*, pages 13–22. ACM, 1985.

[3] Sayyad Nayyaroddeen, Mahak Gambhir, and Kishore Kothapalli. A study of graph decomposition algorithms for parallel symmetry breaking. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 598–607. IEEE Computer Society, 2017. doi: 10.1109/IPDPSW.2017.120.

[4] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16 (2):187–260, 1984. ISSN 0360-0300. doi: 10.1145/356924.356930.

[5] Franz Aurenhammer. Voronoi Diagrams—A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys*, 23(3):345–405, 1991. ISSN 0360-0300. doi: 10.1145/116873.116880.

[6] Ulrike Bücking. Conformally Symmetric Triangular Lattices and Discrete ϑ-Conformal Maps. *International Mathematics Research Notices*, 2020(21):7314–7335, December 2019. doi: 10.1093/imrn/rnz308. URL `https://arxiv.org/abs/1808.08064`. Published online December 2019; arXiv preprint arXiv:1808.08064 (last updated February 2020).

[7] David Glickenstein and Lee Sidbury. Convergence of discrete conformal mappings on surfaces, 2025.

[8] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997. doi: 10.1007/PL00009321.

[9] Andreas Fabri and Sylvain Pion. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '09)*, pages 538–539. ACM, 2009. doi: 10.1145/1653771.1653865. Library available at `https://www.cgal.org`.

[10] Taco S. Cohen and Max Welling. Group equivariant convolutional networks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pages 2990–2999. PMLR, 2016.

[11] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021.

[12] Nathan O. Schmidt. The Tri-Quarter Framework: Unifying Complex Coordinates with Topological and Reflective Duality Across Circles of Any Radius. HAL Open Archive, hal-05010951 (preprint), 2025. URL `https://hal.science/hal-05010951`. Preprint available on TechRxiv: `https://www.techrxiv.org/users/906377/articles/1281679`.

[13] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory*, volume 84 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1990. ISBN 0-387-97329-X. doi: 10.1007/978-1-4757-2103-4.

[14] Mois I. Aroyo, editor. *International Tables for Crystallography, Volume A: Space-Group Symmetry*. Wiley, 6th edition, 2016. doi: 10.1107/97809553602060000114.

[15] H. S. M. Coxeter. *Introduction to Geometry*. Wiley, 1961.

[16] Hans Schwerdtfeger. *Geometry of Complex Numbers: Circle Geometry, Moebius Transformation, Non-Euclidean Geometry*. Dover Publications, 1979.

[17] John H. Conway, Heidi Burgiel, and Chaim Goodman-Strauss. *The Symmetries of Things*. A K Peters, 2008.

[18] Reinhard Diestel. *Graph Theory*. Springer, 5th edition, 2017. Section on lattice graphs and chromatic numbers, including triangular lattices as 3-colorable examples.

[19] David A. Brannan, Matthew F. Esplen, and Jeremy J. Gray. *Geometry*. Cambridge University Press, Cambridge, UK, April 1999. ISBN 9780521597876. First edition.

[20] Tobias Friedrich, Maximilian Katzmann, and Leon Schiller. Computing Voronoi Diagrams in the Polar-Coordinate Model of the Hyperbolic Plane. *CoRR*, abs/2112.02553, 2021. URL `https://arxiv.org/abs/2112.02553`.

[21] Hassler Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34(2):339–362, 1932. doi: 10.1090/S0002-9947-1932-1501641-2.

[22] Michael P. Hitchman. *Geometry with an Introduction to Cosmic Topology*. Jones & Bartlett Learning, 2009.

[23] Lars V. Ahlfors. *Complex Analysis*. McGraw-Hill, 3rd edition, 1979.

[24] Bruno Ernst. *The Magic Mirror of M.C. Escher*. Ballantine Books, New York, 1976.

[25] László Lovász. *Discrete Mathematics: Elementary and Beyond*. Springer, 2003. doi: 10.1007/b97469.

[26] Yuriy Kozhubaev and Ruide Yang. Simulation of dynamic path planning of symmetrical trajectory of mobile robots based on improved a* and artificial potential field fusion for natural resource exploration. *Symmetry*, 16(7):801, 2024. doi: 10.3390/sym16070801.

[27] Nikolaos Bousias, Stefanos Pertigkiozoglou, Kostas Daniilidis, and George J. Pappas. Symmetries-enhanced multi-agent reinforcement learning, 2025.

[28] Daniele Micciancio. Duality in lattice cryptography, 2010. Invited talk at PKC 2010.

[29] Gareth Loy and Alexander Zelinsky. Fast Radial Symmetry for Detecting Points of Interest. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(8):959–973, 2003. doi: 10.1109/TPAMI.2003.1217601.

[30] Jean-Marc Girault. Symmetry in signals: A new insight. *Entropy*, 26(11):941, 2024. doi: 10.3390/e26110941.

[31] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006. doi: 10.1016/B978-0-12-369446-1.X5000-7.

[32] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, 2014. doi: 10.1109/IPDPS.2014.64. URL `https://doi.org/10.1109/IPDPS.2014.64`.

[33] Edward F. Moore. The Shortest Path Through a Maze. *Proceedings of an International Symposium on the Theory of Switching*, pages 285–292, 1959.

[34] Robert Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010.

[35] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1 (1):269–271, 1959. doi: 10.1007/BF01386390.

[36] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, UK, 2006. ISBN 9780521862059. doi: 10.1017/CBO9780511546877.

[37] Liliya Kharevych, Boris Springborn, and Peter Schröder. Discrete Conformal Mappings via Circle Patterns. *ACM Transactions on Graphics*, 25(2):412–438, 2006. doi: 10.1145/1138450.1138461.

[38] Ulrike Bücking. Approximation of Conformal Mappings Using Conformally Equivalent Triangular Lattices. In Alexander I. Bobenko, editor, *Advances in Discrete Differential Geometry*, pages 133–149. Springer, 2016. doi: 10.1007/978-3-662-50447-5_3.

[39] I. A. Dynnikov and S. P. Novikov. Geometry of the Triangle Equation on Two-Manifolds. *Moscow Mathematical Journal*, 3(2):419–438, 2003.

[40] S. P. Novikov. New discretization of complex analysis: The euclidean and hyperbolic planes. *Proceedings of the Steklov Institute of Mathematics*, 273:238–251, 2011. doi: 10.1134/S0081543811040122.

[41] Adam Doliwa, Maciej Nieszporski, and Paolo Maria Santini. Integrable lattices and their sublattices ii. from the b-quadrilateral lattice to the self-adjoint schemes on the triangular and the honeycomb lattices. *J. Math. Phys.*, 48:113506, 2007. doi: 10.1063/1.2803504.

[42] Kenneth Stephenson. *Introduction to Circle Packing: The Theory of Discrete Analytic Functions*. Cambridge University Press, 2005.

[43] William P. Thurston. The Geometry and Topology of Three-Manifolds. *Princeton University lecture notes*, 1980. Chapter 13 introduces circle packings on triangulated surfaces.

[44] Alexander I. Bobenko and Boris A. Springborn. Variational Principles for Circle Patterns and Koebe's Theorem. *Transactions of the American Mathematical Society*, 356(2):659–689, 2004. doi: 10.1090/S0002-9947-03-03239-2.

[45] Philip L. Bowers and Kenneth Stephenson. Uniformizing Dessins and Belyĭ Maps via Circle Packing. *Memoirs of the American Mathematical Society*, 170(805), 2004. doi: 10.1090/memo/0805.

[46] Jack E. Graver. Catalog of Self-Dual Plane Graphs. *Combinatorica*, 17(2):183–207, 1997. doi: 10.1007/BF01200894.

[47] Lester R. Ford and Delbert R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi: 10.4153/CJM-1956-045-5.

[48] Wendelin Boehmer, Vitaly Kurin, and Shimon Whiteson. Deep Coordination Graphs. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 980–991. PMLR, July 2020. URL `https://proceedings.mlr.press/v119/boehmer20a.html`.

[49] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001.

[50] John A. Bondy and U.S.R. Murty. *Graph Theory*. Springer, 2008. doi: 10.1007/978-1-84628-970-5.

[51] Wolfram Research. Triangular Grid Graph. `https://mathworld.wolfram.com/TriangularGridGraph.html`, 2024. Accessed: 2025-07-23.

[52] Eric W. Weisstein. Hexagonal lattice. MathWorld–A Wolfram Web Resource, 2024. URL `https://mathworld.wolfram.com/HexagonalLattice.html`. Accessed: 2025-08-20.

[53] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. doi: 10.1137/0136016.

[54] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, 5th edition, 1991. ISBN 9780471625469.

[55] Michael Blum. Circle Inversion of the Lines of the Triangular Lattice. `http://www1.lasalle.edu/~blum/c152wks/Triangular_Circle_Inversion.pdf`, n.d.

[56] Nathan O. Schmidt. Tri-Quarter Toolbox GitHub Repository. `https://github.com/nathanoschmidt/tri-quarter-toolbox/`, 2025. Accessed: 2026-06-10.

[57] Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali. Parallel graph partitioning on multicore architectures. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 246–260. Springer, Berlin, Heidelberg, 2011. ISBN 978-3-642-19594-5. doi: 10.1007/978-3-642-19595-2_17.

[58] IEEE Standard for Floating-Point Arithmetic, July 2019.

[59] Martin Hasenbusch. Finite size scaling study of lattice models in the three-dimensional ising universality class. *Phys. Rev. B*, 82:174433, 2010. doi: 10.1103/PhysRevB.82.174433.

[60] Florian Huc, Ignasi Sau, and Janez Žerovnik. (l, k)-routing on plane grids. *Journal of Interconnection Networks*, 10(01n02):27–57, 2009. doi: 10.1142/S0219265909002431.

[61] Yanbin Liu and Yuanyuan Jiang. Robotic path planning based on a triangular mesh map. *International Journal of Control, Automation and Systems*, 18(10):2658–2666, 2020. doi: 10.1007/s12555-019-0396-z.

[62] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing*, 8(1):1–39, 2021. doi: 10.1145/3434393.

[63] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. Efficiently Indexing Shortest Paths by Exploiting Symmetry in Graphs. In *EDBT 2009*, pages 493–504, Saint Petersburg, Russia, March 2009. ACM. doi: 10.1145/1516360.1516460.

[64] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel Triangle Counting and Enumeration using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015. doi: 10.1109/IPDPSW.2015.75.

[65] Tadamasa Sawada and Qasim Zaidi. Rotational-symmetry in a 3D Scene and its 2D Image. *Journal of Mathematical Psychology*, 87:108–125, 2018. doi: 10.1016/j.jmp.2018.10.001.

[66] David R. Karger and Clifford Stein. An $O(n^2)$ Algorithm for Minimum Cuts. *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC '93)*, pages 757–765, 1993. doi: 10.1145/167088.167281.

[67] Gary William Flake, Robert E. Tarjan, and Kostas Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 1(4):385–408, 2004. doi: 10.1080/15427951.2004.10129093.

[68] Robert Görke, Tanja Hartmann, and Dorothea Wagner. Dynamic Graph Clustering Using Minimum-Cut Trees. *Journal of Graph Algorithms and Applications*, 16(2):411–446, 2012. doi: 10.7155/jgaa.00269.

[69] Alan F. Beardon. *The Geometry of Discrete Groups*. Springer, 1983.

[70] Christopher Hammond. *The Basics of Crystallography and Diffraction*. International Union of Crystallography Texts on Crystallography. Oxford University Press, 4th edition, 2015.

[71] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi: 10.1137/0202019. URL `https://doi.org/10.1137/0202019`.

[72] Robert I Jedrzejewski. Ccd surface photometry of elliptical galaxies. i. observations, reduction and results. *Monthly Notices of the Royal Astronomical Society*, 226(4):747–768, 1987. doi: 10.1093/mnras/226.4.747.

[73] Steven B. Howell. *Handbook of CCD Astronomy*. Cambridge Observing Handbooks for Research Astronomers. Cambridge University Press, 2nd edition, 2006. Standard reference for logarithmic/geometric radial binning in profile photometry.

[74] James G. Oxley. *Matroid Theory*. Oxford University Press, 2006.

[75] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[76] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *Journal of the ACM*, 56(6):1–40, 2009. doi: 10.1145/1568318.1568324.

[77] A. E. Inopin and N. O. Schmidt. Proof of quark confinement and baryon-antibaryon duality: I: Gauge symmetry breaking in dual 4D fractional quantum Hall superfluidic space-time. *Hadronic Journal*, 35 (5):469, 2012.

Appendix A Simulation 1: Visualizing Random Connections in the Lattice Graph

This Python script dynamically visualizes random adjacent paths in the outer zone of the radial dual triangular lattice graph, with their inversions mirrored in the inner zone. The script animates the connections, updating every 5 seconds, to demonstrate the reflective duality in action.

This script requires Python 3.x and the Pygame library (install via `pip install pygame`). It runs on systems with a graphical interface (e.g., Windows, macOS, or Linux with X11) and does not require additional hardware. The visualization references Figure 4 for a static example. Run the script via:

```
python simulation_01_visualize_random_connections.py
```

PYTHON SOURCE CODE 1. `simulation_01_visualize_random_connections.py`

```python
import pygame
import sys
import math
import random

# Initialize Pygame library
pygame.init()

# Set screen dimensions and window title
WIDTH, HEIGHT = 1200, 900
screen = pygame.display.set_mode((WIDTH, HEIGHT))  # Create the display window
pygame.display.set_caption(
    "Tri-Quarter Framework Simulation: "
    "Visualizing Random Connections in the "
    "Radial Dual Triangular Lattice Graph"
)

# Define colors used in the figure
WHITE = (255, 255, 255)  # Background
BLACK = (0, 0, 0)        # Axes and text
GRAY = (128, 128, 128)   # Dotted lines
RED = (255, 0, 0)        # Outer zone vertices and paths
GREEN = (0, 180, 0)      # Boundary zone vertices and circle
BLUE = (0, 0, 255)       # Inner zone vertices and paths

# Sector colors with opacity for wedges
color0 = (255, 0, 0)
color1 = (255, 255, 0)
color2 = (0, 255, 0)
color3 = (2, 192, 230)
color4 = (2, 67, 230)
color5 = (188, 2, 230)
sector_colors = [color0, color1, color2, color3, color4, color5]

# Fonts for labels
font_str = "Arial"
font_large = pygame.font.SysFont(font_str, 48)
font_largedium = pygame.font.SysFont(font_str, 36)
font_medium = pygame.font.SysFont(font_str, 28)
font_small = pygame.font.SysFont(font_str, 24)
font_tiny = pygame.font.SysFont(font_str, 22)

# Center of the screen for origin
center_x, center_y = WIDTH // 2, HEIGHT // 2

# Scaling factor reduced to make drawing smaller and fit better: 120 -> 100
scale = 100  # Adjust this to zoom in/out

# Define inversion radius r = sqrt(1), r_sq = 1 for boundary
```

```
# (aligned with sector boundaries)
r = math.sqrt(1)
r_sq = 1

# Truncation radius R = 4 for generating finite vertices
R = 4

# Basis vectors for triangular lattice
omega1 = (1, 0)
omega2 = (0.5, math.sqrt(3) / 2)

# Deltas for finding nearest neighbors in lattice coordinates
deltas = [(1, 0), (0, 1), (-1, 0), (0, -1), (1, -1), (-1, 1)]

# Function to compute Cartesian position from lattice coordinates (m, n)
def compute_pos(m, n):
    x = (m * omega1[0]) + (n * omega2[0])
    y = (m * omega1[1]) + (n * omega2[1])
    return x, y

# Function to compute squared norm (integer) for exact comparisons
def compute_norm_sq(m, n):
    return (m * m) + (m * n) + (n * n)

# Generate lattice vertices for outer zone and boundary zone
outer_vertices = []  # List of (m, n) for outer vertices
boundary_vertices = []  # List of positions for boundary vertices
max_intnorm = 0  # Track max squared norm for scaling inner radii
min_intnorm_outer = float('inf')  # Track min squared norm for outer
for m in range(-20, 21):  # Range large enough to cover R=4
    for n in range(-20, 21):
        if m == 0 and n == 0:
            continue  # Exclude origin
        intnorm = compute_norm_sq(m, n)
        norm = math.sqrt(intnorm)
        if norm > R:
            continue  # Truncate beyond R
        if intnorm == r_sq:
            pos = compute_pos(m, n)
            boundary_vertices.append(
                {'pos': pos, 'angle': math.atan2(pos[1], pos[0])}
            )
        elif intnorm > r_sq:
            outer_vertices.append((m, n))  # Store outer lattice coords
            max_intnorm = max(max_intnorm, intnorm)
            min_intnorm_outer = min(min_intnorm_outer, intnorm)

# Sort boundary vertices by angle for hexagon drawing
boundary_vertices.sort(key = lambda p: p['angle'])

# Generate inner vertices by inverting outer vertices
inner_vertices = []  # List of inner vertices with positions and radii
# Min distance for inner scaling (adjusted for r_sq=1)
min_dist_prime = r_sq / math.sqrt(max_intnorm)
# Max distance for inner scaling (adjusted for r_sq=1)
max_dist_prime = r_sq / math.sqrt(min_intnorm_outer)

for m, n in outer_vertices:
    pos = compute_pos(m, n)
    intnorm = compute_norm_sq(m, n)
    xprime = r_sq * pos[0] / intnorm  # Invert x (r_sq=1)
    yprime = r_sq * pos[1] / intnorm  # Invert y (r_sq=1)
    dist_prime = math.sqrt(xprime**2 + yprime**2)
    # Scale radius based on distance (smaller near origin, adjusted for new norms)
    rad_blue = 1 + (dist_prime - min_dist_prime) / (
```

```
        max_dist_prime - min_dist_prime
    ) if max_dist_prime > min_dist_prime else 1
    rad_blue *= 0.75
    inner_vertices.append(
        {'pos': (xprime, yprime), 'rad': rad_blue, 'orig_mn': (m,n)}
    )

# Build adjacency list for outer graph
outer_set = set(outer_vertices)  # Set for quick lookup
neighbors = {}  # Dict of neighbors for each outer vertex
for v in outer_vertices:
    neigh = []
    m, n = v
    for dm, dn in deltas:
        mm, nn = m + dm, n + dn
        if (mm, nn) in outer_set:
            neigh.append((mm, nn))  # Add adjacent if in outer
    neighbors[v] = neigh

# Convert Cartesian to screen coordinates (invert y for Pygame)
def to_screen(x, y):
    return int(center_x + x * scale), int(center_y - y * scale)  # y inverted

# Dashed line
def draw_dashed_line(start, end, color, dash_length = 10, thickness = 2):
    sx, sy = start
    ex, ey = end
    dx = ex - sx
    dy = ey - sy
    dist = math.sqrt(dx**2 + dy**2)
    if dist == 0:
        return
    ux = dx / dist
    uy = dy / dist
    current_x, current_y = sx, sy
    while dist > 0:
        step = min(dash_length, dist)
        next_x = current_x + (ux * step)
        next_y = current_y + (uy * step)
        pygame.draw.line(screen, color, (current_x, current_y),
                         (next_x, next_y), thickness)
        current_x = next_x + (ux * dash_length)
        current_y = next_y + (uy * dash_length)
        dist -= 2 * dash_length

# Function to draw dashed circle (approximated with segments)
def draw_dashed_circle(center, radius, color, dash_length = 10):
    num_segments = 100  # Number of segments for smooth circle
    angle_step = 2 * math.pi / num_segments
    for i in range(num_segments):
        if i % 2 == 0:  # Draw every other segment for dash effect
            theta1 = i * angle_step
            theta2 = (i + 1) * angle_step
            x1 = center[0] + radius * math.cos(theta1)
            y1 = center[1] + radius * math.sin(theta1)
            x2 = center[0] + radius * math.cos(theta2)
            y2 = center[1] + radius * math.sin(theta2)
            pygame.draw.line(screen, color, (x1, y1), (x2, y2), 2)

# Draw the background elements: sectors, rays, axes, labels
def draw_background():
    screen.fill(WHITE)  # Clear screen with white

    # Draw transparent sector wedges
    for k in range(6):
```

```
        points = [(center_x, center_y)]  # Start from center
        wedge_radius = 10 * scale
         # Incremental points for polygon
        for angle in range(k * 60, ((k + 1) * 60) + 1, 5):
            rad = math.radians(angle)
            px = center_x + (wedge_radius * math.cos(rad))
            py = center_y - (wedge_radius * math.sin(rad))  # Inverted y
            points.append((px, py))
        col = sector_colors[k]
        col_alpha = (*col, 51)  # 0.2 opacity (255*0.2=51)
        s = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)  # Alpha surface
        pygame.draw.polygon(s, col_alpha, points)
        screen.blit(s, (0, 0))

    # Draw dashed radial rays
    for k in range(6):
        angle = math.radians(k * 60)
        rad_len = 4 if k in [0, 3] else 4.6  # Vary length as in figure
        end_x = rad_len * math.cos(angle)
        end_y = rad_len * math.sin(angle)
        start_screen = to_screen(0, 0)
        end_screen = to_screen(end_x, end_y)
        draw_dashed_line(start_screen, end_screen, BLACK, dash_length = 10,
                         thickness = 2)

    # Draw real and imaginary axes
    pygame.draw.line(screen, BLACK, to_screen(-4.2, 0), to_screen(4.2, 0), 3)
    pygame.draw.line(screen, BLACK, to_screen(0,-4), to_screen(0, 4), 3)

    # Add little black arrows at endpoints
    arrow_size = 10
    # Real axis right
    rx, ry = to_screen(4.2, 0)
    arrow_points = [(rx, ry - arrow_size//2), (rx + arrow_size, ry),
                    (rx, ry + arrow_size//2)]
    pygame.draw.polygon(screen, BLACK, arrow_points)
    # Real axis left
    lx, ly = to_screen(-4.2, 0)
    arrow_points = [(lx, ly - arrow_size//2), (lx - arrow_size, ly),
                    (lx, ly + arrow_size//2)]
    pygame.draw.polygon(screen, BLACK, arrow_points)
    # Imag axis up (positive imag)
    ux, uy = to_screen(0, 3.9)
    arrow_points = [(ux - arrow_size//2, uy), (ux, uy - arrow_size),
                    (ux + arrow_size//2, uy)]
    pygame.draw.polygon(screen, BLACK, arrow_points)
    # Imag axis down (negative imag)
    dx, dy = to_screen(0, -3.9)
    arrow_points = [(dx - arrow_size//2, dy), (dx, dy + arrow_size),
                    (dx + arrow_size//2, dy)]
    pygame.draw.polygon(screen, BLACK, arrow_points)

    # Draw white box with black border in top right for parameters
    box_x, box_y = to_screen(2.2, 4.57)
    box_width, box_height = 390, 100
    pygame.draw.rect(screen, WHITE, (box_x, box_y, box_width, box_height))
    pygame.draw.rect(screen, BLACK, (box_x, box_y, box_width, box_height), 1)

    # Draw truncation radius parameter value
    trunc_radius_x, trunc_radius_y = 5.0, 4.4
    text = font_medium.render('R = 4', True, BLACK)
    screen.blit(text, to_screen(trunc_radius_x, trunc_radius_y))

    # Draw inversion radius parameter value
    inver_radius_x, inver_radius_y = 5.08, 4.1
```

```
    text = font_medium.render('r = \u221A' + str(r_sq), True, BLACK)
    screen.blit(text, to_screen(inver_radius_x, inver_radius_y))

    # Outer zone vertex
    red_zone_vertex_x, red_zone_vertex_y = 2.3, 4.5
    text = font_medium.render('\u25CF Outer Zone', True, RED)
    screen.blit(text, to_screen(red_zone_vertex_x, red_zone_vertex_y))

    # Boundary zone vertex
    green_zone_vertex_x, green_zone_vertex_y = 2.3, 4.2
    text = font_medium.render('\u25CF Boundary Zone', True, GREEN)
    screen.blit(text, to_screen(green_zone_vertex_x, green_zone_vertex_y))

    # Inner zone vertex
    blue_zone_vertex_x, blue_zone_vertex_y = 2.3, 3.9
    text = font_medium.render('\u25CF Inner Zone', True, BLUE)
    screen.blit(text, to_screen(blue_zone_vertex_x, blue_zone_vertex_y))

    # Draw dashed boundary circle
    circle_center = to_screen(0, 0)
    circle_radius = int(r * scale)
    draw_dashed_circle(circle_center, circle_radius, GREEN)

    # Dashed green hexagon for boundary
    if boundary_vertices:
        for i in range(len(boundary_vertices)):
            pos1 = boundary_vertices[i]['pos']
            pos2 = boundary_vertices[(i + 1) % len(boundary_vertices)]['pos']
            p1 = to_screen(*pos1)
            p2 = to_screen(*pos2)
            draw_dashed_line(p1, p2, GREEN, dash_length = 5, thickness = 2)

    # Render zone labels with Unicode and subscripts
    # Inner zone Lambda_{-,r}
    pos_x, pos_y = to_screen(1.1, 0.5)
    text_main = font_largedium.render('\u039B', True, BLUE)
    screen.blit(text_main, (pos_x, pos_y))
    text_sub = font_tiny.render('-,\u221A' + str(r_sq), True, BLUE)
    screen.blit(text_sub, (pos_x + text_main.get_width(), pos_y + 20))
    text_sup = font_tiny.render('4', True, BLUE)
    screen.blit(text_sup, (pos_x + text_main.get_width(), pos_y + 1))
    # Draw inner zone pointer line segment
    # (because Lambda_{-,r} is too big to fit inside the circle)
    ptr_line_start = (pos_x, pos_y + 20)
    ptr_line_end = (pos_x - 50, pos_y + 20)
    draw_dashed_line(ptr_line_start, ptr_line_end, BLUE, dash_length = 5,
                     thickness = 2)

    # Boundary zone Lambda_{T,r}
    pos_x, pos_y = to_screen(0.9, 0.9)
    text_main = font_largedium.render('\u039B', True, GREEN)
    screen.blit(text_main, (pos_x, pos_y))
    text_sub = font_tiny.render('T,\u221A' + str(r_sq), True, GREEN)
    screen.blit(text_sub, (pos_x + text_main.get_width(), pos_y + 20))
    text_sup = font_tiny.render('4', True, GREEN)
    screen.blit(text_sup, (pos_x + text_main.get_width(), pos_y + 1))

    # Outer zone Lambda_{+,r}
    pos_x, pos_y = to_screen(1.5, 1.5)
    text_main = font_largedium.render('\u039B', True, RED)
    screen.blit(text_main, (pos_x, pos_y))
    text_sub = font_tiny.render('+,\u221A' + str(r_sq), True, RED)
    screen.blit(text_sub, (pos_x + text_main.get_width(), pos_y + 20))
    text_sup = font_tiny.render('4', True, RED)
    screen.blit(text_sup, (pos_x + text_main.get_width(), pos_y + 1))
```

```
    # Render angular sector labels
    ang_sect_radius = 4.3
    for k in [0, 2, 3, 5]:
        angle = (k * 60) + 30
        px, py = to_screen(
            ang_sect_radius * math.cos(math.radians(angle)),
            ang_sect_radius * math.sin(math.radians(angle))
        )
        text_main = font_large.render('S', True, BLACK)
        screen.blit(text_main, (px - 20, py - 20))
        text_sub = font_tiny.render(str(k), True, BLACK)
        screen.blit(text_sub, (px + 5, py + 17))

    # Special position for angular sector S1 to avoid overlap
    ang_sect_radius -= 0.4
    px, py = to_screen(
        ang_sect_radius * math.cos(math.radians(83)),
        ang_sect_radius * math.sin(math.radians(83))
    )
    text_main = font_large.render('S', True, BLACK)
    screen.blit(text_main, (px - 20, py - 20))
    text_sub = font_tiny.render('1', True, BLACK)
    screen.blit(text_sub, (px + 5, py + 17))

    # Special position for angular sector S4 to avoid overlap
    px, py = to_screen(
        ang_sect_radius * math.cos(math.radians(277)),
        ang_sect_radius * math.sin(math.radians(277))
    )
    py -= 20
    text_main = font_large.render('S', True, BLACK)
    screen.blit(text_main, (px - 20, py - 20))
    text_sub = font_tiny.render('4', True, BLACK)
    screen.blit(text_sub, (px + 5, py + 17))

    # Render quadrant phase pair labels
    q1_x, q1_y = 2.6, 3.5
    text = font_medium.render('Quadrant I: (0, \u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(q1_x, q1_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(q1_x, q1_y)[0] + 270, to_screen(q1_x, q1_y)[1] + 10)
    )

    q2_x, q2_y = -5.4, 3.5
    text = font_medium.render('Quadrant II: (\u03C0, \u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(q2_x, q2_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(q2_x, q2_y)[0] + 276, to_screen(q2_x, q2_y)[1] + 10)
    )

    q3_x, q3_y = -5.4, -3.1
    text = font_medium.render('Quadrant III: (\u03C0, 3\u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(q3_x, q3_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(q3_x, q3_y)[0] + 302, to_screen(q3_x, q3_y)[1] + 10)
    )

    q4_x, q4_y = 2.2, -3.1
```

```
    text = font_medium.render('Quadrant IV: (0, 3\u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(q4_x, q4_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(q4_x, q4_y)[0] + 307, to_screen(q4_x, q4_y)[1] + 10)
    )

    # Render axis phase pair labels
    east_x, east_y = 4.45, 0.15
    text = font_small.render('East: (0, 0)', True, BLACK)
    screen.blit(text, to_screen(east_x, east_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(east_x, east_y)[0] + 132, to_screen(east_x, east_y)[1] + 10)
    )

    north_x, north_y = -1, 4.4
    text = font_small.render('North: (\u03C0/2, \u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(north_x, north_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(north_x, north_y)[0] + 190, to_screen(north_x, north_y)[1] + 10)
    )

    west_x, west_y = -5.9, 0.15
    text = font_small.render('West: (\u03C0, 0)', True, BLACK)
    screen.blit(text, to_screen(west_x, west_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(west_x, west_y)[0] + 138, to_screen(west_x, west_y)[1] + 10)
    )

    south_x, south_y = -1.1, -4.05
    text = font_small.render('South: (\u03C0/2, 3\u03C0/2)', True, BLACK)
    screen.blit(text, to_screen(south_x, south_y))
    text_phi = font_tiny.render('\u03C6', True, BLACK)
    screen.blit(
        text_phi,
        (to_screen(south_x, south_y)[0] + 208, to_screen(south_x, south_y)[1] + 10)
    )

    # Render axis labels
    text = font_large.render('\u211D', True, BLACK)
    screen.blit(text, to_screen(3.5, 0.6))
    text = font_large.render('\U0001D540', True, BLACK)
    screen.blit(text, to_screen(-0.3, 4))

def draw_vertices():
    for p in boundary_vertices:
        px, py = to_screen(*p['pos'])
        pygame.draw.circle(screen, GREEN, (px, py), 8)

    for m, n in outer_vertices:
        pos = compute_pos(m, n)
        px, py = to_screen(*pos)
        pygame.draw.circle(screen, RED, (px, py), 8)

    for p in inner_vertices:
        px, py = to_screen(*p['pos'])
        rad = int(p['rad'] * 4)
        pygame.draw.circle(screen, BLUE, (px, py), rad)
```

```
def select_random_path():
    num_verts = random.randint(3, 5)
    start = random.choice(outer_vertices)
    path = [start]
    visited = set([start])
    while len(path) < num_verts:
        curr = path[-1]
        avail = [n for n in neighbors[curr] if n not in visited]
        if not avail:
            break
        next_v = random.choice(avail)
        path.append(next_v)
        visited.add(next_v)
    return path

def get_pos(mn):
    return compute_pos(*mn)

def get_inv_pos(mn):
    m, n = mn
    intnorm = compute_norm_sq(m, n)
    pos = get_pos(mn)
    xprime = r_sq * pos[0] / intnorm
    yprime = r_sq * pos[1] / intnorm
    return xprime, yprime

def draw_selected_path(path):
    for i in range(len(path) - 1):
        pos1 = get_pos(path[i])
        pos2 = get_pos(path[i + 1])
        p1 = to_screen(*pos1)
        p2 = to_screen(*pos2)
        pygame.draw.line(screen, RED, p1, p2, 4)

    for i in range(len(path) - 1):
        pos1 = get_inv_pos(path[i])
        pos2 = get_inv_pos(path[i + 1])
        p1 = to_screen(*pos1)
        p2 = to_screen(*pos2)
        pygame.draw.line(screen, BLUE, p1, p2, 3)

    for v in path:
        pos = get_pos(v)
        inv = get_inv_pos(v)
        p1 = to_screen(*pos)
        p2 = to_screen(*inv)
        draw_dashed_line(p1, p2, GRAY, dash_length = 5)

clock = pygame.time.Clock()
current_path = select_random_path()
timer = 0

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    dt = clock.tick(60) / 1000.0
    timer += dt

    if timer >= 5:
        current_path = select_random_path()
        timer = 0
```

```
    draw_background()
    draw_vertices()
    draw_selected_path(current_path)
    pygame.display.flip()
```

Appendix B Utility Helper Tools

This appendix provides utility scripts for generating and analyzing the radial dual triangular lattice graphs Λ_r^R, including functions to build zone subgraphs and complete graphs with twin edges, compute vertex counts across zones and angular sectors, calculate truncation error percentages, and enumerate boundary vertices for admissible inversion radii. These modular tools facilitate validation of the framework's symmetries (e.g., order-6 rotational invariance under D_6) and approximation accuracies, and are used to produce data for tables like 8, 7, and 9.

B.1 Generating the Lattice Graphs

This utility script generates the truncated radial dual triangular lattice graph and requires Python 3.x with NetworkX (install via `pip install networkx`). It runs on standard systems without special requirements. As a helper script, it is imported by other scripts (e.g., for vertex counting or benchmarking) and not intended to be run standalone.

Example usage: In dependent scripts that require the generation of only the zone lattice subgraphs, import as

```
from radial_dual_triangular_lattice_graph import build_zone_subgraphs
```

then call

```
G_outer, G_inner, inversion_map = build_zone_subgraphs(10, 1)
```

with truncation radius $R = 10$ and inversion radius $r = 1$. Additionally, in dependent scripts that require the generation of the complete lattice graph, import as

```
from radial_dual_triangular_lattice_graph import build_complete_lattice_graph
```

then call

```
G_outer, G_inner, inversion_map = build_complete_lattice_graph(10, 1)
```

with truncation radius $R = 10$ and inversion radius $r = 1$.

PYTHON SOURCE CODE 2. `radial_dual_triangular_lattice_graph.py`

```
import networkx as nx
import math
import cmath  # For complex rotations (unused in core, but kept for potential extensions)

def build_zone_subgraphs(R, r_sq=1):
    """
    Build separate outer and inner zone subgraphs for the truncated radial dual
    triangular lattice graph Lambda_r^R.
    This supports isolated zone operations like inversion-based
    path mirroring without boundary or cross-zone edges.

    Args:
        R (float): Truncation radius (R >> r for balanced approximation).
        r_sq (int, optional): Squared inversion radius (admissible N = r^2
                              representable as m^2 + m n + n^2 for m,n in Z,
                              not both zero; default 1 for unit hexagon boundary).

    Returns:
```

```
        tuple: (G_outer, G_inner, inversion_map)
            - G_outer: Outer zone subgraph Lambda_{+,r}^R (nx.Graph).
            - G_inner: Inner zone subgraph Lambda_{-,r}^R (nx.Graph, induced via iota_r).
            - inversion_map: Dict mapping outer nodes to inner twins (bijection).
    """
    G_outer = nx.Graph()
    G_inner = nx.Graph()
    inversion_map = {}

    # Neighbor deltas for degree-6 triangular lattice connectivity
    # (aligned with order-6 rotational symmetry of D_6)
    deltas = [(1,0), (0,1), (-1,0), (0,-1), (1,-1), (-1,1)]

    outer_nodes = []
    # Expand range to ensure coverage within truncation radius R
    # (symmetric around origin, excluding punctured origin per X = C || {0})
    max_range = int(math.ceil(2 * R))
    for m in range(-max_range, max_range + 1):
        for n in range(-max_range, max_range + 1):
            if m == 0 and n == 0: continue  # Exclude origin
            # Integer squared Euclidean norm (exact for Eisenstein integers)
            norm_sq = m*m + m*n + n*n
            if norm_sq <= r_sq or math.sqrt(norm_sq) > R: continue
            # Unified complex-Cartesian coordinates (Equation 2.1)
            x, y = m + n*0.5, n*(math.sqrt(3)/2)
            node = (m, n, 'outer')  # 3-tuple for consistency across zones
            G_outer.add_node(
                node, pos=(x,y), norm_sq=norm_sq
            )
            outer_nodes.append(node)

    # Connect outer edges: nearest-neighbor at Euclidean distance 1
    # (standard triangular lattice spacing, preserving combinatorial duality)
    for u in outer_nodes:
        m, n, _ = u  # Unpack lattice coordinates and type
        for dm, dn in deltas:
            v = (m+dm, n+dn, 'outer')
            if v in G_outer:
                G_outer.add_edge(u, v)

    # Invert outer to inner via circle inversion iota_r
    # (preserves directional consistency via the exact lattice rotation)
    for node in outer_nodes:
        m, n, _ = node
        pos = G_outer.nodes[node]['pos']
        norm_sq = G_outer.nodes[node]['norm_sq']
        x_inv = r_sq * pos[0] / norm_sq
        y_inv = r_sq * pos[1] / norm_sq
        inv_node = (m, n, 'inner')  # Twin node in inner zone
        G_inner.add_node(
            inv_node, pos=(x_inv, y_inv),
            norm_sq=r_sq**2 / norm_sq
        )
        inversion_map[node] = inv_node

    # Connect inner edges by mirroring outer (induce isomorphism via reflective duality)
    # (preserves adjacency topologically)
    for u, v in list(G_outer.edges()):
        inv_u = (u[0], u[1], 'inner')
        inv_v = (v[0], v[1], 'inner')
        G_inner.add_edge(inv_u, inv_v)

    return G_outer, G_inner, inversion_map

def build_complete_lattice_graph(R, r_sq=1):
```

```
    """
    Build the complete truncated radial dual triangular lattice graph Lambda_r^R,
    composing zone subgraphs with boundary vertices and twin edges.
    This supports global computations like clustering coefficients over the full
    structure, including the combinatorial dual boundary separator.

    Args:
        R (float): Truncation radius.
        r_sq (int, optional): Squared inversion radius (default 1).

    Returns:
        tuple: (G, inversion_map)
            - G: Full nx.Graph with inner/outer/boundary zones and twins.
            - inversion_map: Bijection outer <-> inner (boundary fixed).
    """
    G_outer, G_inner, inversion_map = build_zone_subgraphs(R, r_sq)

    # Compose outer and inner subgraphs (mutually disjoint, no cross edges yet)
    G = nx.compose(G_outer, G_inner)

    # Neighbor deltas (reused for boundary and twins)
    deltas = [(1,0), (0,1), (-1,0), (0,-1), (1,-1), (-1,1)]

    boundary_nodes = []
    # Expand range to capture boundary within R
    max_range = int(math.ceil(R)) + 10
    for m in range(-max_range, max_range + 1):
        for n in range(-max_range, max_range + 1):
            if m == 0 and n == 0: continue
            norm_sq = m*m + m*n + n*n
            if norm_sq == r_sq:  # Exact boundary zone V_{T,r}
                x = m + n*0.5
                y = n*(math.sqrt(3)/2)
                node = (m, n, 'boundary')
                G.add_node(
                    node, pos=(x,y), norm_sq=norm_sq
                )
                boundary_nodes.append(node)

    # Connect boundary cycle: nearest neighbors on V_{T,r} (e.g., hexagon for r=1)
    for i, u in enumerate(boundary_nodes):
        m_u, n_u, _ = u  # Unpack
        # Candidate neighbors on boundary
        v_cand = [
            (m_u+1, n_u, 'boundary'), (m_u, n_u+1, 'boundary'),
            (m_u-1, n_u, 'boundary'), (m_u, n_u-1, 'boundary'),
            (m_u+1, n_u-1, 'boundary'), (m_u-1, n_u+1, 'boundary')
        ]
        for v in v_cand:
            if v in G.nodes and math.isclose(
                G.nodes[v].get('norm_sq', 0), r_sq
            ):
                G.add_edge(u, v)

    # Add twin edges: for each outer-boundary edge {outer, b}, add {iota_r(outer), b}
    # (implements Escher reflective duality across boundary separator)
    for b in boundary_nodes:
        m_b, n_b, _ = b
        for dm, dn in deltas:
            m_o, n_o = m_b + dm, n_b + dn
            outer_node = (m_o, n_o, 'outer')
            if outer_node in G.nodes:
                pos_b = G.nodes[b]['pos']
                pos_o = G.nodes[outer_node]['pos']
                # Check standard lattice spacing ~1
```

```
                if math.isclose(
                    math.hypot(pos_o[0]-pos_b[0], pos_o[1]-pos_b[1]),
                    1.0, abs_tol=1e-6
                ):
                    G.add_edge(outer_node, b)
                    # Twin: inner counterpart to boundary
                    twin_outer = (m_o, n_o, 'inner')
                    if twin_outer in G.nodes:
                        G.add_edge(twin_outer, b)

    return G, inversion_map

def lattice_rotate(m, n, k):
    """
    Apply Z_6 rotation (order-6 cyclic group action) to Eisenstein integer
    coordinates (m, n) by k steps of 60 degrees, as per the rotational symmetry
    of the base triangular lattice L. This supports orbit computation
    for symmetry-reduced algorithms like clustering.

    Args:
        m, n (int): Lattice coordinates.
        k (int): Rotation steps (mod 6).

    Returns:
        tuple: Rotated (m', n').
    """
    k = k % 6  # Normalize to [0,5]
    for _ in range(k):
        temp = m
        m = -n
        n = temp + n

    return m, n
```

B.2 Counting Vertices in Radial Zones and Angular Sectors

This utility script computes vertex counts in zones and sectors for given inversion radius r (rounded to yield integer r^2 with lattice points) and truncation radius R. Best practices: Use values of r where r^2 is an integer with positive representations in the triangular lattice (e.g., 1, 7); validate output for valid r^2; handle large R with care to avoid memory issues due to $O(R^2)$ vertex growth; extend for custom analyses by modifying sector computation or adding visualizations.

This script requires Python 3.x and imports the helper `radial_dual_triangular_lattice_graph.py`. It runs on standard systems without special requirements. The output directly generates data for Table 8.

The results in Table 8 were generated using the following commands (we approximate $\sqrt{7} \approx 2.64575$ because the script rounds r^2 to 7):

```
python get_vertex_counts.py 1 4
python get_vertex_counts.py 1 10
python get_vertex_counts.py 1 20
python get_vertex_counts.py 1 50
python get_vertex_counts.py 2.64575 4
python get_vertex_counts.py 2.64575 10
python get_vertex_counts.py 2.64575 20
python get_vertex_counts.py 2.64575 50
```

Python Source Code 3. `get_vertex_counts.py`

```
import math
import argparse
from radial_dual_triangular_lattice_graph import build_zone_subgraphs
```

```
def compute_eisenstein_representations(max_nsq):
    """
    Compute the number of Eisenstein integer representations of integers as
    sums of the form m^2 + m*n + n^2 in the triangular lattice L, up to the
    maximum squared norm max_nsq. This supports validation of admissible
    inversion radii r where r^2 = N has positive representations.
    """
    representations = {}  # Dictionary to store the count of representations
                          # for each squared norm nsq
    max_m = int(math.ceil(math.sqrt(max_nsq))) + 10  # Safe range for m, n to
                                                     # cover all possible
                                                     # nsq <= max_nsq
    for m in range(-max_m, max_m + 1):  # Loop over possible m values
        for n in range(-max_m, max_m + 1):  # Loop over possible n values
            if m == 0 and n == 0:
                continue  # Skip the origin to align with punctured complex
                          # plane X
            nsq = m * m + m * n + n * n  # Compute the squared norm in
                                         # triangular lattice L
            if nsq > max_nsq:
                continue  # Skip if beyond the maximum
            if nsq not in representations:
                representations[nsq] = 0  # Initialize count if not present
            representations[nsq] += 1  # Increment the representation count
    return representations


# Primary lattice ray directions d_t (t in Z_6) as Eisenstein coordinate
# pairs: d_t is the image of (1, 0) under t steps of the order-6 lattice
# rotation, i.e. the six nearest-neighbor directions at angles t * 60 degrees.
SECTOR_RAYS = ((1, 0), (0, 1), (-1, 1), (-1, 0), (0, -1), (1, -1))


def angular_sector_index(m, n):
    """
    Determine the angular sector index t in Z_6 (0 to 5) of an Eisenstein
    lattice vertex (m, n), the origin excluded, using exact integer arithmetic.

    The six sectors are the 60-degree wedges between consecutive primary
    lattice rays SECTOR_RAYS. Wedge membership is decided from the integer
    coordinate pair (m, n) via the sign of the lattice cross product
    cross((a, b), (c, d)) = a * d - b * c, which is the orientation of the two
    directions up to the positive constant sqrt(3) / 2 and is therefore an
    exact integer (no floating-point phase is used). A vertex on a primary ray
    is assigned to the sector counterclockwise of that ray, so the six sectors
    tile Z_6 without overlap and the rule is exactly equivariant under the
    order-6 rotational symmetry of D_6.
    """
    for t in range(6):
        dm, dn = SECTOR_RAYS[t]
        em, en = SECTOR_RAYS[(t + 1) % 6]
        if dm * n - dn * m >= 0 and m * en - n * em > 0:
            return t
    raise ValueError(
        f"angular_sector_index: undefined for origin/invalid vertex ({m}, {n})"
    )


def on_primary_ray(m, n):
    """
    Return the index t in Z_6 of the primary lattice ray that vertex (m, n)
    lies on, or None if (m, n) is strictly interior to a sector. A vertex lies
    on ray d_t iff it is a positive integer multiple of d_t, tested exactly via
    a zero lattice cross product with d_t together with a positive dot product.
    """
```

```
    for t in range(6):
        dm, dn = SECTOR_RAYS[t]
        if dm * n - dn * m == 0 and (m * dm + n * dn) > 0:
            return t
    return None


def generate_boundary_zone_vertices(truncation_radius, r_sq):
    """
    Generate vertices in the boundary zone V_{T,r} at exact norm sqrt(r_sq)
    within the truncation radius R, using the base triangular lattice L.
    Stores (m, n) coordinate pairs for phase pair assignments and exact
    integer angular sector partitioning.
    """
    boundary_zone_vertices = []  # List to store boundary vertices (m, n)
    max_m = int(math.ceil(truncation_radius)) + 10  # Safe range for m, n
    for m in range(-max_m, max_m + 1):  # Loop over possible m values
        for n in range(-max_m, max_m + 1):  # Loop over possible n values
            if m == 0 and n == 0:
                continue  # Skip the origin to align with punctured complex
                          # plane X
            nsq = m * m + m * n + n * n  # Compute squared norm
            norm = math.sqrt(nsq)  # Compute actual norm
            if norm > truncation_radius:
                continue  # Skip if outside truncation radius
            if nsq == r_sq:  # Check if on the boundary zone V_{T,r}
                boundary_zone_vertices.append((m, n))  # Add coordinate pair
    return boundary_zone_vertices


def main():
    # Parse command-line arguments for inversion radius r and truncation radius R
    parser = argparse.ArgumentParser(
        description="Calculate vertex counts in Tri-Quarter radial dual "
                    "triangular lattice graph Lambda_r across zones and "
                    "angular sectors."
    )
    parser.add_argument(
        "r", type=float,
        help="Inversion radius r (must yield integer r_sq with lattice points)"
    )
    parser.add_argument(
        "R", type=float, help="Truncation radius R"
    )
    args = parser.parse_args()

    # Propose r_sq and compute max_nsq for Eisenstein representations
    r_sq_proposed = round(args.r * args.r)
    max_nsq = int(args.R * args.R) + 10
    representations = compute_eisenstein_representations(max_nsq)

    # Validate proposed r_sq as admissible (positive representations)
    if r_sq_proposed in representations and representations[r_sq_proposed] > 0:
        r_sq = r_sq_proposed
        effective_r = math.sqrt(r_sq)
        print(f"Valid r_sq = {r_sq}, effective r = {effective_r:.6f}")
    else:
        print(f"Invalid r = {args.r:.6f} (r_sq = {r_sq_proposed}, no lattice "
              f"points at this exact norm).")
        # Find next lower valid N
        lower_n = r_sq_proposed - 1
        while lower_n >= 1 and (
            lower_n not in representations or representations[lower_n] == 0
        ):
            lower_n -= 1
```

```
        # Find next higher valid N
        higher_n = r_sq_proposed + 1
        while higher_n <= max_nsq and (
            higher_n not in representations or representations[higher_n] == 0
        ):
            higher_n += 1
        if lower_n >= 1:
            print(
                f"Next lower valid r = sqrt({lower_n}) approx "
                f"{math.sqrt(lower_n):.6f}"
            )
        if higher_n <= max_nsq:
            print(
                f"Next higher valid r = sqrt({higher_n}) approx "
                f"{math.sqrt(higher_n):.6f}"
            )
        return

    # Generate truncated radial dual triangular lattice graph using imported function
    G_outer, G_inner, inversion_map = build_zone_subgraphs(args.R, r_sq)

    # Count vertices in outer and inner zones (excluding boundary)
    count_outer = len(G_outer.nodes)
    count_inner = len(G_inner.nodes)

    # Generate and count boundary zone vertices V_{T,r}
    boundary_zone_vertices = generate_boundary_zone_vertices(args.R, r_sq)
    count_boundary = len(boundary_zone_vertices)

    # Calculate total vertices in truncated Lambda_r^R
    total = count_outer + count_inner + count_boundary

    # Initialize sector counts for outer, inner, and boundary zones
    sector_counts_outer = [0] * 6
    for node, _ in G_outer.nodes(data=True):
        sector = angular_sector_index(node[0], node[1])
        sector_counts_outer[sector] += 1

    sector_counts_inner = [0] * 6
    for node, _ in G_inner.nodes(data=True):
        sector = angular_sector_index(node[0], node[1])
        sector_counts_inner[sector] += 1

    sector_counts_boundary = [0] * 6
    for m_v, n_v in boundary_zone_vertices:
        sector = angular_sector_index(m_v, n_v)
        sector_counts_boundary[sector] += 1

    # Print zone counts
    print(f"Outer zone vertices: {count_outer}")
    print(f"Inner zone vertices: {count_inner}")
    print(f"Boundary zone vertices: {count_boundary}")
    print(f"Total vertices: {total}")
    print(
        "Vertices per angular sector (outer + boundary + inner = total):"
    )
    for k in range(6):
        total_sector = (
            sector_counts_outer[k]
            + sector_counts_inner[k]
            + sector_counts_boundary[k]
        )
        print(
            f"S_{k}: {sector_counts_outer[k]} + "
            f"{sector_counts_boundary[k]} + "
```

```
            f"{sector_counts_inner[k]} = {total_sector}"
        )

    # Calculate and print average vertex count per angular sector
    print("\nAverage vertex count per angular sector:")
    print(f"Outer: {count_outer / 6:.2f}")
    print(f"Inner: {count_inner / 6:.2f}")
    print(f"Boundary: {count_boundary / 6:.2f}")
    print(f"Total: {total / 6:.2f}")

    # Count vertices on angular sector borders (primary rays at t * 60 degrees),
    # detected exactly from the integer coordinate pair (m, n) via on_primary_ray.
    ray_labels = [
        "East (0 deg)", "North-East (60 deg)", "North-West (120 deg)",
        "West (180 deg)", "South-West (240 deg)", "South-East (300 deg)"
    ]
    ray_counts_outer = [0] * 6
    for node, _ in G_outer.nodes(data=True):
        k = on_primary_ray(node[0], node[1])
        if k is not None:
            ray_counts_outer[k] += 1

    ray_counts_inner = [0] * 6
    for node, _ in G_inner.nodes(data=True):
        k = on_primary_ray(node[0], node[1])
        if k is not None:
            ray_counts_inner[k] += 1

    ray_counts_boundary = [0] * 6
    for m_v, n_v in boundary_zone_vertices:
        k = on_primary_ray(m_v, n_v)
        if k is not None:
            ray_counts_boundary[k] += 1

    print("\nVertices on angular sector borders (primary rays):")
    for k in range(6):
        print(
            f"{ray_labels[k]}: outer {ray_counts_outer[k]}, "
            f"boundary {ray_counts_boundary[k]}, "
            f"inner {ray_counts_inner[k]}"
        )


if __name__ == "__main__":
    main()
```

B.3 Computing Truncation Errors

This Python script computes truncation error percentages for various truncation radii R with inversion radius $r = 1$ for Λ_r to quantify the unresolved area near the origin in $\Lambda_{-,1}$ as a fraction of the total viewed area. The unresolved area is $\pi(r^2/R)^2 = \pi/R^2$ (for $r = 1$), and the total viewed area is approximated as πR^2. Results are output as a LaTeX table for direct integration.

This script requires Python 3.x with the math module (standard library; no external dependencies). It runs on standard systems without special requirements.

Run the script with:

```
python compute_truncation_errors.py
```

PYTHON SOURCE CODE 4. `compute_truncation_errors.py`

```
import math
```

```
# Parameters for the radial dual triangular lattice graph Lambda_r
r = 1.0  # Admissible inversion radius (r=1 yields symmetric boundary
         # set V_{T,1} with |V_{T,1}|=6 vertices forming a unit hexagon,
         # aligned with primary rays at phases t pi / 3 for t in Z_6)
Rs = [4, 10, 20, 50]  # List of truncation radii R to compute errors for
                      # (ensures R >> r for balanced finite approximations
                      # of the infinite Lambda_r with gaps scaling as O(1/R^2))

# Compute truncation errors for each R
# (unresolved area near punctured origin in inner zone Lambda_{-,r} as fraction
# of total viewed area; aligns with "looking scope" approximation)
print("Truncation Error Percentages for Various R (with r=1):")
print("R | Unresolved Area (pi r^4 / R^2) | Total Viewed Area (~ pi R^2) | "
      "Percentage (%)")
for R in Rs:
    # Unresolved area: pi (r^2 / R)^2 near origin in Lambda_{-,r}
    # (vanishing as R -> infinity, enabling scalable finite simulations)
    unresolved_area = math.pi * (r**4 / R**2)
    # Total viewed area approximation: pi R^2 (disk area up to truncation R)
    total_area_approx = math.pi * R**2
    # Error percentage: (unresolved / total) * 100
    percentage = (unresolved_area / total_area_approx) * 100
    print(f"{R} | {unresolved_area:.4f} | {total_area_approx:.2f} | "
          f"{percentage:.4f}%")
```

B.4 Computing Boundary Vertices for Admissible Inversion Radii

This Python script verifies the explicit boundary vertices for a given $N = r^2$ (e.g., $N = 7$ for $r = \sqrt{7}$) by finding integer solutions to $a^2 + ab + b^2 = N$, computing phases, and grouping by angular sector. It reproduces the data in Table 9 and can be extended for other N.

This script requires Python 3.x (uses standard libraries: math, argparse; no external dependencies like NetworkX or Pygame). It runs on standard systems without special requirements.

Run the script for $N = 7$ with:

```
python compute_boundary_vertices.py 7
```

to obtain the output:

```
Boundary points for N=7 (r=sqrt(7)):
Sector 0: (2,1) at 0.333 rad
Sector 0: (1,2) at 0.714 rad
Sector 1: (-1,3) at 1.381 rad
Sector 1: (-2,3) at 1.761 rad
Sector 2: (-3,2) at 2.428 rad
Sector 2: (-3,1) at 2.808 rad
Sector 3: (-2,-1) at 3.475 rad
Sector 3: (-1,-2) at 3.855 rad
Sector 4: (1,-3) at 4.522 rad
Sector 4: (2,-3) at 4.903 rad
Sector 5: (3,-2) at 5.569 rad
Sector 5: (3,-1) at 5.950 rad
```

PYTHON SOURCE CODE 5. `compute_boundary_vertices.py`

```
import math
import argparse

# Primary lattice ray directions d_t (t in Z_6) as Eisenstein coordinate
# pairs: d_t is the image of (1, 0) under t steps of the order-6 lattice
# rotation, i.e. the six nearest-neighbor directions at angles t * 60 degrees.
SECTOR_RAYS = ((1, 0), (0, 1), (-1, 1), (-1, 0), (0, -1), (1, -1))
```

```
def angular_sector_index(m, n):
    """
    Determine the angular sector index t in Z_6 (0 to 5) of an Eisenstein
    lattice vertex (m, n), the origin excluded, using exact integer arithmetic.

    The six sectors are the 60-degree wedges between consecutive primary
    lattice rays SECTOR_RAYS. Wedge membership is decided from the integer
    coordinate pair (m, n) via the sign of the lattice cross product
    cross((a, b), (c, d)) = a * d - b * c, which is the orientation of the two
    directions up to the positive constant sqrt(3) / 2 and is therefore an
    exact integer (no floating-point phase is used). A vertex on a primary ray
    is assigned to the sector counterclockwise of that ray, so the six sectors
    tile Z_6 without overlap and the rule is exactly equivariant under the
    order-6 rotational symmetry of D_6.
    """
    for t in range(6):
        dm, dn = SECTOR_RAYS[t]
        em, en = SECTOR_RAYS[(t + 1) % 6]
        if dm * n - dn * m >= 0 and m * en - n * em > 0:
            return t
    raise ValueError(
        f"angular_sector_index: undefined for origin/invalid vertex ({m}, {n})"
    )


def find_representations(N):
    """
    Find integer solutions (m, n) to m^2 + m*n + n^2 = N in the base triangular
    lattice L and assign each to its angular sector index t in Z_6 using exact
    integer arithmetic (no floating-point phase). This supports verification of
    boundary zone vertices V_{T,r} for admissible inversion radii r where
    r^2 = N has positive representations, ensuring symmetric distribution across
    angular sectors S_t under D_6 rotational symmetry.
    """
    reps = []  # List of (sector, (m, n))
    # Safe range for m, n to cover all possible solutions without overflow for small N
    max_m = int(math.ceil(math.sqrt(N))) + 10
    for m in range(-max_m, max_m + 1):
        for n in range(-max_m, max_m + 1):
            # Skip origin to align with punctured complex plane X = C \ {0}
            if m == 0 and n == 0: continue
            # Compute squared Euclidean norm (integer, exact for Eisenstein integers)
            norm_sq = m*m + m*n + n*n
            if norm_sq == N:  # Check if on the boundary circle of radius r
                # Exact integer angular sector index t in Z_6 for mod 6
                # partitioning and order-6 rotational invariance
                sector = angular_sector_index(m, n)
                reps.append((sector, (m, n)))
    # Sort by sector index then coordinates for ordered output that highlights
    # equidistribution across angular sectors S_t
    reps.sort(key=lambda item: (item[0], item[1]))
    return reps

def main():
    # Parse command-line arguments for flexibility in specifying N = r^2
    parser = argparse.ArgumentParser(
        description="Compute boundary vertices for admissible r^2 = N in the "
                    "base triangular lattice L, grouped by angular sector to "
                    "demonstrate symmetric distribution under D_6."
    )
    parser.add_argument(
        "N", type=int, nargs="?", default=7,
        help="Integer N = r^2 (default: 7 for r = sqrt(7) example with 12 "
```

```
            "boundary vertices)"
    )
    args = parser.parse_args()

    # Find all representations for the given N
    representations = find_representations(args.N)

    # Print results grouped by sector, showing uniform equidistribution
    # (e.g., exactly k vertices per sector for |V_{T,r}| = 6k)
    print(f"Boundary vertices for N={args.N} (r=sqrt({args.N})):")
    for rep in representations:
        sector, (m, n) = rep
        print(f"Sector {sector}: (m,n)=({m},{n})")

if __name__ == "__main__":
    main()
```

Appendix C Inversion-Based Path Mirroring Simulation Experiment

This appendix provides the Python scripts used for the inversion-based path mirroring benchmarks in Section 6. The first script builds the graph structure of the truncated radial dual triangular lattice graph Λ_r^R, while the subsequent ones benchmark standard and TQF approaches to demonstrate performance gains from duality, which includes the 1.6–1.8x speedups achieved via exact bijective mappings, verified bitwise-identical to full recomputation.

C.1 Benchmarking the Standard Approach

This script benchmarks the standard (recompute) approach to path mirroring and requires Python 3.x with NetworkX, time, random, argparse, and statistics modules (NetworkX install via `pip install networkx`; others are standard). It runs on standard systems without special requirements and imports the helper `radial_dual_triangular_lattice_graph.py`. The benchmarks contribute to Table 12.

Example commands to execute the benchmarks in Table 12:

```
python simulation_02_benchmark_standard_path_mirroring.py 5 --runs 20 --timing_repeats 100
python simulation_02_benchmark_standard_path_mirroring.py 10 --runs 20 --timing_repeats 100
python simulation_02_benchmark_standard_path_mirroring.py 15 --runs 20 --timing_repeats 100
python simulation_02_benchmark_standard_path_mirroring.py 20 --runs 20 --timing_repeats 100
python simulation_02_benchmark_standard_path_mirroring.py 25 --runs 20 --timing_repeats 100
```

Python Source Code 6. `simulation_02_benchmark_standard_path_mirroring.py`

```
import time
import random
import argparse
import statistics

import networkx as nx

from radial_dual_triangular_lattice_graph import build_zone_subgraphs


def standard_dual_zone_paths(G_outer, G_inner, start_outer, inversion_map):
    """Solve the dual-zone shortest-path problem by full recomputation.

    Computes single-source shortest-path hop distances (the discrete dual
    metric) in the outer zone from start_outer, and independently in the inner
    zone from the corresponding inversion twin start_inner. Returns both
    distance dictionaries so the result can be verified against the
    inversion-based approach of Simulation 03.
```

```
    Args:
        G_outer: Outer zone subgraph Lambda_{+,r}^R.
        G_inner: Inner zone subgraph Lambda_{-,r}^R.
        start_outer: Source vertex in the outer zone.
        inversion_map: Bijection mapping outer vertices to inner twins.

    Returns:
        tuple (outer_dist, inner_dist) of {vertex: hop_distance} dictionaries.
    """
    outer_dist = nx.single_source_shortest_path_length(G_outer, start_outer)
    start_inner = inversion_map.get(start_outer)
    inner_dist = (
        nx.single_source_shortest_path_length(G_inner, start_inner)
        if start_inner is not None
        else {}
    )
    return outer_dist, inner_dist


def benchmark_standard_path_mirroring(G_outer, G_inner, start_outer,
                                      inversion_map, runs, timing_repeats):
    """Time the standard recompute approach over multiple runs.

    Each run times timing_repeats solutions of the dual-zone problem and
    records the mean per-solution wall-clock time in milliseconds. Returns the
    mean and standard deviation across runs.
    """
    times = []
    for _ in range(runs):
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            standard_dual_zone_paths(
                G_outer, G_inner, start_outer, inversion_map
            )
        times.append((time.perf_counter() - t0) * 1000 / timing_repeats)
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return statistics.mean(times), std


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description=(
            "Benchmark the standard (recompute) baseline for the dual-zone "
            "shortest-path problem on a truncated radial dual triangular "
            "lattice graph Lambda_r^R. Times are in milliseconds (ms)."
        )
    )
    parser.add_argument("R", type=int, nargs="?", default=10,
                        help="Truncation radius R (default: 10)")
    parser.add_argument("--runs", type=int, default=20,
                        help="Number of benchmark runs (default: 20)")
    parser.add_argument("--timing_repeats", type=int, default=100,
                        help="Repeats per run for accuracy (default: 100)")
    parser.add_argument("--seed", type=int, default=42,
                        help="Random seed for source selection (default: 42)")
    args = parser.parse_args()

    print(f"Building radial dual triangular lattice graph with "
          f"truncation radius R={args.R}...")
    G_outer, G_inner, inversion_map = build_zone_subgraphs(args.R)
    num_outer = len(G_outer.nodes())
    num_inner = len(G_inner.nodes())
    print(f"Graphs built: Outer {num_outer} vertices, "
          f"Inner {num_inner} vertices.")
```

```
    # Fixed seed so the standard and Tri-Quarter benchmarks select the same
    # source vertex and are therefore directly comparable and verifiable.
    random.seed(args.seed)
    start_outer = (
        random.choice(sorted(G_outer.nodes())) if num_outer > 0 else None
    )

    print(f"Running {args.runs} benchmarks, each with {args.timing_repeats} "
          f"timing repeats.")
    avg, std = benchmark_standard_path_mirroring(
        G_outer, G_inner, start_outer, inversion_map,
        args.runs, args.timing_repeats
    )
    print(f"Standard Path Mirroring (Recompute): {avg:.3f} ms (+/-{std:.3f})")
```

C.2 Benchmarking the Tri-Quarter Approach

This script benchmarks the TQF duality approach to path mirroring (using bijection for efficiency) and requires Python 3.x with NetworkX, time, random, argparse, and statistics modules (NetworkX install via `pip install networkx`; others are standard). It runs on standard systems without special requirements and imports the helper `radial_dual_triangular_lattice_graph.py`. The benchmarks contribute to Table 12.

Example commands to execute the benchmarks in Table 12:

```
python simulation_03_benchmark_triquarter_path_mirroring.py 5 --runs 20 --timing_repeats 100
python simulation_03_benchmark_triquarter_path_mirroring.py 10 --runs 20 --timing_repeats 100
python simulation_03_benchmark_triquarter_path_mirroring.py 15 --runs 20 --timing_repeats 100
python simulation_03_benchmark_triquarter_path_mirroring.py 20 --runs 20 --timing_repeats 100
python simulation_03_benchmark_triquarter_path_mirroring.py 25 --runs 20 --timing_repeats 100
```

Python Source Code 7. `simulation_03_benchmark_triquarter_path_mirroring.py`

```
import time
import random
import argparse
import statistics

import networkx as nx

from radial_dual_triangular_lattice_graph import build_zone_subgraphs


def mirror_paths(outer_dist, inversion_map):
    """Map outer-zone hop distances into the inner zone via iota_r.

    For each outer vertex with a known hop distance, the inversion bijection
    iota_r yields the inner-zone twin. The Escher reflective duality makes
    iota_r a distance-preserving graph isomorphism between the zones, so the
    outer hop distance transfers unchanged to the twin. Runs in O(|outer_dist|)
    time and requires no shortest-path computation in the inner zone.

    Args:
        outer_dist: {outer_vertex: hop_distance} from the outer-zone solve.
        inversion_map: Bijection mapping outer vertices to inner twins.

    Returns:
        {inner_vertex: hop_distance} for the inner zone.
    """
    mirrored = {}
    for vertex, distance in outer_dist.items():
        twin = inversion_map.get(vertex)
        if twin is not None:
```

```
            mirrored[twin] = distance
    return mirrored


def triquarter_dual_zone_paths(G_outer, start_outer, inversion_map):
    """Solve the dual-zone shortest-path problem by inversion-based mirroring.

    Computes single-source shortest-path hop distances in the outer zone once,
    then obtains the inner-zone distances by mirroring through iota_r instead
    of recomputing. Returns both distance dictionaries.
    """
    outer_dist = nx.single_source_shortest_path_length(G_outer, start_outer)
    inner_dist = mirror_paths(outer_dist, inversion_map)
    return outer_dist, inner_dist


def verify_mirror_exactness(G_inner, start_outer, inversion_map, mirrored):
    """Confirm that mirrored inner-zone distances match an independent solve.

    Independently recomputes single-source shortest-path hop distances in the
    inner zone and checks that every mirrored value is exactly equal. Returns
    True only if the mirrored result is bitwise-identical to the recomputed
    result, demonstrating that the inversion-based shortcut preserves the
    discrete dual metric exactly.
    """
    start_inner = inversion_map.get(start_outer)
    if start_inner is None:
        return len(mirrored) == 0
    recomputed = nx.single_source_shortest_path_length(G_inner, start_inner)
    if set(recomputed.keys()) != set(mirrored.keys()):
        return False
    return all(recomputed[v] == mirrored[v] for v in recomputed)


def benchmark_triquarter_path_mirroring(G_outer, start_outer, inversion_map,
                                        runs, timing_repeats):
    """Time the Tri-Quarter inversion-based approach over multiple runs.

    Each run times timing_repeats solutions of the dual-zone problem and
    records the mean per-solution wall-clock time in milliseconds. Returns the
    mean and standard deviation across runs.
    """
    times = []
    for _ in range(runs):
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            triquarter_dual_zone_paths(G_outer, start_outer, inversion_map)
        times.append((time.perf_counter() - t0) * 1000 / timing_repeats)
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return statistics.mean(times), std


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description=(
            "Benchmark the Tri-Quarter inversion-based approach to the "
            "dual-zone shortest-path problem on a truncated radial dual "
            "triangular lattice graph Lambda_r^R. Solves the outer zone once "
            "and mirrors into the inner zone via the circle inversion "
            "bijection iota_r, with an exactness check against independent "
            "recomputation. Times are in milliseconds (ms)."
        )
    )
    parser.add_argument("R", type=int, nargs="?", default=10,
                        help="Truncation radius R (default: 10)")
```

```
    parser.add_argument("--runs", type=int, default=20,
                        help="Number of benchmark runs (default: 20)")
    parser.add_argument("--timing_repeats", type=int, default=100,
                        help="Repeats per run for accuracy (default: 100)")
    parser.add_argument("--seed", type=int, default=42,
                        help="Random seed for source selection (default: 42)")
    args = parser.parse_args()

    print(f"Building radial dual triangular lattice graph with "
          f"truncation radius R={args.R}...")
    G_outer, G_inner, inversion_map = build_zone_subgraphs(args.R)
    num_outer = len(G_outer.nodes())
    num_inner = len(G_inner.nodes())
    print(f"Graphs built: Outer {num_outer} vertices, "
          f"Inner {num_inner} vertices.")

    # Fixed seed so this benchmark and the standard baseline (Simulation 02)
    # select the same source vertex, making the comparison and the exactness
    # verification directly meaningful.
    random.seed(args.seed)
    start_outer = (
        random.choice(sorted(G_outer.nodes())) if num_outer > 0 else None
    )

    # Verify exactness once before timing: confirm the inversion-based shortcut
    # reproduces the independently recomputed inner-zone result exactly.
    _, mirrored = triquarter_dual_zone_paths(
        G_outer, start_outer, inversion_map
    )
    exact = verify_mirror_exactness(
        G_inner, start_outer, inversion_map, mirrored
    )
    print(f"Exactness check (mirrored == recomputed): {'PASS' if exact else 'FAIL'}")

    print(f"Running {args.runs} benchmarks, each with {args.timing_repeats} "
          f"timing repeats.")
    avg, std = benchmark_triquarter_path_mirroring(
        G_outer, start_outer, inversion_map, args.runs, args.timing_repeats
    )
    print(f"Tri-Quarter Path Mirroring (Inversion): {avg:.3f} ms (+/-{std:.3f})")
```

Appendix D Symmetry-Reduced Clustering Simulation Experiment

This appendix contains scripts for benchmarking the symmetry-reduced computation of the average local clustering coefficient on the full truncated radial dual triangular lattice graph Λ_r^R (including inner, boundary, and outer zones with twin edges), which exploits the order-6 rotational symmetry via $\mathbb{Z}_6$ orbits to compute on representatives and replicate via group actions (as per Lemma 3.10 and Corollary 4.3). The scripts demonstrate practical speedups from equivariance, which contribute to the results in Table 13 of Section 6.

D.1 Benchmarking the Standard Approach

This script benchmarks the standard (full recompute) approach to computing the average local clustering coefficient on the full truncated radial dual triangular lattice graph Λ_r^R (including inner, boundary, and outer zones with twin edges). It iterates over all vertices to compute triangle densities in neighborhoods, providing a baseline for symmetry-reduced methods. The benchmarks contribute to Table 13.

This script requires Python 3.x with NetworkX, time, random, argparse, and statistics modules (NetworkX install via `pip install networkx`; others are standard). It runs on standard systems without special requirements and imports the helper `radial_dual_triangular_lattice_graph.py`.

Example commands to execute the benchmarks in Table 13:

```
python simulation_04_benchmark_standard_clustering.py 10 --runs 30 --timing_repeats 100
python simulation_04_benchmark_standard_clustering.py 20 --runs 30 --timing_repeats 100
python simulation_04_benchmark_standard_clustering.py 50 --runs 30 --timing_repeats 30
python simulation_04_benchmark_standard_clustering.py 100 --runs 30 --timing_repeats 10
python simulation_04_benchmark_standard_clustering.py 150 --runs 30 --timing_repeats 5
python simulation_04_benchmark_standard_clustering.py 200 --runs 30 --timing_repeats 5
```

Python Source Code 8. simulation_04_benchmark_standard_clustering.py

```python
import time
import argparse
import statistics
from fractions import Fraction

from radial_dual_triangular_lattice_graph import build_complete_lattice_graph


def build_adjacency_sets(G):
    """Return a {vertex: frozenset(neighbors)} adjacency-set view of G.

    Precomputing this view once lets the clustering routine perform triangle
    counting through O(1) average-case set membership tests, keeping the
    baseline's per-vertex constant factor small.
    """
    return {v: frozenset(G.neighbors(v)) for v in G.nodes()}


def local_clustering_exact(adjacency, v):
    """Exact local clustering coefficient of vertex v as a Fraction.

    The value is 2 * (edges among neighbors) / (deg * (deg - 1)). Both the
    numerator and denominator are integers, so the coefficient is an exact
    rational with no floating-point round-off. Vertices of degree below 2
    contribute exactly Fraction(0).
    """
    neigh = adjacency[v]
    deg = len(neigh)
    if deg < 2:
        return Fraction(0)
    neigh_list = tuple(neigh)
    common = 0
    for i in range(deg):
        ni_adj = adjacency[neigh_list[i]]
        for j in range(i + 1, deg):
            if neigh_list[j] in ni_adj:
                common += 1
    return Fraction(2 * common, deg * (deg - 1))


def compute_average_clustering_standard_exact(adjacency):
    """Exact average local clustering coefficient over all vertices.

    Accumulates the per-vertex exact rational coefficients and divides by the
    vertex count, returning a single exact Fraction. Because the arithmetic is
    exact, the result is independent of summation order and is therefore
    bitwise-reproducible across implementations.

    Args:
        adjacency: {vertex: frozenset(neighbors)} adjacency-set view.

    Returns:
        The average local clustering coefficient as an exact Fraction
        (Fraction(0) if the graph is empty).
```

```
    """
    num_v = len(adjacency)
    if num_v == 0:
        return Fraction(0)
    total = Fraction(0)
    for v in adjacency:
        total += local_clustering_exact(adjacency, v)
    return total / num_v


def benchmark_standard_clustering(adjacency, runs, timing_repeats):
    """Time the exact standard clustering computation over multiple runs.

    Each run times timing_repeats full-graph exact clustering computations and
    records the mean per-computation wall-clock time in milliseconds. Returns
    the mean and standard deviation across runs.
    """
    times = []
    for _ in range(runs):
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            compute_average_clustering_standard_exact(adjacency)
        times.append((time.perf_counter() - t0) * 1000 / timing_repeats)
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return statistics.mean(times), std


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description=(
            "Benchmark the standard (full recompute) average local "
            "clustering coefficient on the complete truncated radial dual "
            "triangular lattice graph Lambda_r^R, in exact rational "
            "arithmetic. Times are in milliseconds."
        )
    )
    parser.add_argument("R", type=int, nargs="?", default=10,
                        help="Truncation radius R (default: 10)")
    parser.add_argument("--runs", type=int, default=20,
                        help="Number of benchmark runs (default: 20)")
    parser.add_argument("--timing_repeats", type=int, default=20,
                        help="Repeats per run for accuracy (default: 20)")
    args = parser.parse_args()

    G, _ = build_complete_lattice_graph(args.R)
    adjacency = build_adjacency_sets(G)
    num_v = len(adjacency)
    print(f"Graph: |V|={num_v}")

    coeff = compute_average_clustering_standard_exact(adjacency)
    print(f"Average clustering coefficient (exact): {coeff} = {float(coeff):.12f}")

    avg, std = benchmark_standard_clustering(
        adjacency, args.runs, args.timing_repeats
    )
    print(f"Standard (exact): {avg:.3f} ms +/- {std:.3f}")
```

D.2 Benchmarking the Tri-Quarter Approach

This script benchmarks the TQF symmetry-reduced approach to computing the average local clustering coefficient on the full truncated radial dual triangular lattice graph Λ_r^R, exploiting the order-6 rotational symmetry via a $\mathbb{Z}_6$-orbit transversal to compute on representatives and replicate via group actions (as per

Lemma 3.10 and the corollary on $\mathbb{Z}_6$ symmetry exploitation). This avoids recomputation on $\sim 5/6$ of the vertices, demonstrating speedups from equivariance. The benchmarks contribute to Table 13.

This script requires Python 3.x with NetworkX, time, random, argparse, and statistics modules (NetworkX install via `pip install networkx`; others are standard). It runs on standard systems without special requirements and imports the helper `radial_dual_triangular_lattice_graph.py`.

Example commands to execute the benchmarks in Table 13:

```
python simulation_05_benchmark_triquarter_clustering.py 10 --runs 30 --timing_repeats 100
python simulation_05_benchmark_triquarter_clustering.py 20 --runs 30 --timing_repeats 100
python simulation_05_benchmark_triquarter_clustering.py 50 --runs 30 --timing_repeats 30
python simulation_05_benchmark_triquarter_clustering.py 100 --runs 30 --timing_repeats 10
python simulation_05_benchmark_triquarter_clustering.py 150 --runs 30 --timing_repeats 5
python simulation_05_benchmark_triquarter_clustering.py 200 --runs 30 --timing_repeats 5
```

PYTHON SOURCE CODE 9. `simulation_05_benchmark_triquarter_clustering.py`

```python
import time
import argparse
import statistics
from fractions import Fraction

from radial_dual_triangular_lattice_graph import (
    build_complete_lattice_graph, lattice_rotate
)
from simulation_04_benchmark_standard_clustering import (
    build_adjacency_sets,
    local_clustering_exact,
    compute_average_clustering_standard_exact,
)

try:
    import numpy as np
    NUMPY_AVAILABLE = True
except ImportError:
    NUMPY_AVAILABLE = False


def get_symmetry_orbits_python(G, debug=False):
    """Partition the vertex set into Z_6 orbits via a visited-set traversal."""
    visited = set()
    orbits = []
    for node in G.nodes():
        if node in visited:
            continue
        m, n, node_type = node
        orbit = [node]
        for k in range(1, 6):
            rot_m, rot_n = lattice_rotate(m, n, k)
            rot_node = (rot_m, rot_n, node_type)
            if rot_node in G and rot_node not in visited:
                orbit.append(rot_node)
                visited.add(rot_node)
        visited.add(node)
        orbits.append(orbit)
    if debug:
        total = sum(len(set(o)) for o in orbits)
        print(f"Debug (python): {len(orbits)} orbits, "
              f"avg size {total / len(orbits):.2f}, "
              f"coverage {total == len(G)}")
    return orbits


def get_symmetry_orbits_numpy(G, debug=False):
```

```
    """Partition the vertex set into Z_6 orbits using NumPy batched rotations."""
    nodes = list(G.nodes())
    index_of = {node: i for i, node in enumerate(nodes)}
    coords = np.array([(m, n) for (m, n, _) in nodes], dtype=np.int64)
    base = np.array([[0, -1], [1, 1]], dtype=np.int64)
    rotations = [np.linalg.matrix_power(base, k) for k in range(6)]
    rotated = [coords @ rot.T for rot in rotations]
    visited = [False] * len(nodes)
    orbits = []
    for i in range(len(nodes)):
        if visited[i]:
            continue
        orbit = []
        for k in range(6):
            rm, rn = int(rotated[k][i, 0]), int(rotated[k][i, 1])
            cand = (rm, rn, nodes[i][2])
            j = index_of.get(cand)
            if j is not None and not visited[j]:
                visited[j] = True
                orbit.append(nodes[j])
        orbits.append(orbit)
    if debug:
        total = sum(len(set(o)) for o in orbits)
        print(f"Debug (numpy): {len(orbits)} orbits, "
              f"avg size {total / len(orbits):.2f}, "
              f"coverage {total == len(G)}")
    return orbits


def get_symmetry_orbits(G, method="auto", debug=False):
    """Dispatch to the requested orbit-transversal construction."""
    if method == "numpy" or (method == "auto" and NUMPY_AVAILABLE):
        if not NUMPY_AVAILABLE:
            raise RuntimeError("NumPy requested but not installed.")
        return get_symmetry_orbits_numpy(G, debug=debug)
    return get_symmetry_orbits_python(G, debug=debug)


def compute_average_clustering_triquarter_exact(adjacency, orbits):
    """Exact average local clustering coefficient via orbit replication.

    Computes the exact rational local coefficient on one representative per
    orbit and replicates it across the orbit, weighted by orbit size. Because
    the order-6 rotation is a graph automorphism, every member of an orbit has
    the identical coefficient, so this exactly reproduces the full-graph sum as
    a rational number. The denominator (total weight) equals the vertex count.

    Args:
        adjacency: {vertex: frozenset(neighbors)} adjacency-set view.
        orbits: Z_6-orbit transversal from get_symmetry_orbits.

    Returns:
        The average local clustering coefficient as an exact Fraction
        (Fraction(0) if the graph is empty).
    """
    total = Fraction(0)
    total_weight = 0
    for orbit in orbits:
        rep = orbit[0]
        clust_rep = local_clustering_exact(adjacency, rep)
        orbit_size = len(set(orbit))
        total += clust_rep * orbit_size
        total_weight += orbit_size
    return total / total_weight if total_weight > 0 else Fraction(0)
```

```
def verify_orbit_member_equality(adjacency, orbits):
    """Confirm every member of every orbit shares the representative's exact
    coefficient (i.e. the rotation action genuinely preserves local
    clustering). Returns True if the symmetry reduction is lossless."""
    for orbit in orbits:
        rep_val = local_clustering_exact(adjacency, orbit[0])
        for member in orbit:
            if local_clustering_exact(adjacency, member) != rep_val:
                return False
    return True


def benchmark_triquarter_clustering(adjacency, orbits, runs, timing_repeats):
    """Time the exact symmetry-reduced clustering computation over multiple
    runs. The orbit transversal is supplied precomputed (a one-time cost
    amortized over repeated queries)."""
    times = []
    for _ in range(runs):
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            compute_average_clustering_triquarter_exact(adjacency, orbits)
        times.append((time.perf_counter() - t0) * 1000 / timing_repeats)
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return statistics.mean(times), std


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description=(
            "Benchmark the Tri-Quarter symmetry-reduced average local "
            "clustering coefficient on the complete truncated radial dual "
            "triangular lattice graph Lambda_r^R in exact rational "
            "arithmetic, using a Z_6-orbit transversal. Times are in "
            "milliseconds."
        )
    )
    parser.add_argument("R", type=int, nargs="?", default=10,
                        help="Truncation radius R (default: 10)")
    parser.add_argument("--runs", type=int, default=20,
                        help="Number of benchmark runs (default: 20)")
    parser.add_argument("--timing_repeats", type=int, default=20,
                        help="Repeats per run for accuracy (default: 20)")
    parser.add_argument("--orbit-method", choices=["auto", "python", "numpy"],
                        default="auto",
                        help="Orbit-transversal construction (default: auto)")
    parser.add_argument("--debug", action="store_true",
                        help="Print orbit-transversal statistics")
    args = parser.parse_args()

    G, _ = build_complete_lattice_graph(args.R)
    adjacency = build_adjacency_sets(G)
    num_v = len(adjacency)
    print(f"Graph: |V|={num_v}")
    if args.orbit_method in ("auto", "numpy"):
        print(f"NumPy available: {NUMPY_AVAILABLE}")

    t0 = time.perf_counter()
    orbits = get_symmetry_orbits(G, method=args.orbit_method, debug=args.debug)
    orbit_build_ms = (time.perf_counter() - t0) * 1000
    print(f"Orbit transversal: {len(orbits)} orbits "
          f"built in {orbit_build_ms:.2f} ms "
          f"(method={args.orbit_method})")

    # Exactness verification BEFORE timing: the orbit-reduced exact rational
```

```
    # must equal the full-graph exact rational bitwise (==), not merely to a
    # floating-point tolerance.
    members_ok = verify_orbit_member_equality(adjacency, orbits)
    ct = compute_average_clustering_triquarter_exact(adjacency, orbits)
    cs = compute_average_clustering_standard_exact(adjacency)
    exact_match = (ct == cs)
    print(f"Orbit member-equality check: {'PASS' if members_ok else 'FAIL'}")
    print(f"Exact match (orbit == standard, as Fraction): "
          f"{'PASS' if exact_match else 'FAIL'}")
    print(f"Float images identical: {float(ct) == float(cs)}")
    print(f"Average clustering coefficient (exact): {ct} = {float(ct):.12f}")

    avg, std = benchmark_triquarter_clustering(
        adjacency, orbits, args.runs, args.timing_repeats
    )
    print(f"Tri-Quarter (exact): {avg:.3f} ms +/- {std:.3f}")
```

Appendix E Conflict-Free Parallel Processing Simulation Experiment

This appendix provides the Python script used for the trihexagonal six-coloring conflict-free parallel benchmark in Section 6. The script partitions the complete truncated radial dual triangular lattice graph Λ_1^R into the six independent color classes of the trihexagonal six-coloring e_6 (Definition 5.10; proper by Corollary 5.13) and performs a color-ordered relaxation sweep, benchmarking a CPU baseline (NumPy) against a GPU backend (PyTorch). It requires Python 3.x with NetworkX, NumPy, and PyTorch (NetworkX via `pip install networkx`, NumPy via `pip install numpy`, and a CUDA-enabled PyTorch build matched to the local NVIDIA driver via the instructions at `https://pytorch.org/get-started/locally/`). When no CUDA device is present the GPU backend falls back to the CPU PyTorch device. The script imports the helper `radial_dual_triangular_lattice_graph.py` and contributes to Table 14. Before timing, it verifies that e_6 is a proper coloring on the constructed graph and that the CPU and GPU results agree to within floating-point tolerance.

Example commands to execute the benchmarks in Table 14:

```
python simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py 50 --runs 30 --timing_repeats 20 --sweeps 10
python simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py 100 --runs 30 --timing_repeats 20 --sweeps 10
python simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py 150 --runs 30 --timing_repeats 20 --sweeps 10
python simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py 200 --runs 30 --timing_repeats 20 --sweeps 10
```

Python Source Code 10. `simulation_06_benchmark_trihexagonal_sixcoloring_gpu.py`

```
import math
import time
import argparse
import statistics

import numpy as np

from radial_dual_triangular_lattice_graph import build_complete_lattice_graph

try:
    import torch
    TORCH_AVAILABLE = True
except ImportError:
    TORCH_AVAILABLE = False


# Primary lattice ray directions d_t (t in Z_6) as Eisenstein coordinate
# pairs: d_t is the image of (1, 0) under t steps of the order-6 lattice
# rotation, i.e. the six nearest-neighbor directions at angles t * 60 degrees.
```

```
SECTOR_RAYS = ((1, 0), (0, 1), (-1, 1), (-1, 0), (0, -1), (1, -1))


def angular_sector_index(m, n):
    """Return the angular sector index s_6 in Z_6 of an Eisenstein lattice
    vertex (m, n) (the origin excluded).

    The six sectors are the 60-degree wedges between consecutive primary
    lattice rays SECTOR_RAYS. Wedge membership is decided exactly from the
    integer coordinate pair (m, n) using the sign of the lattice cross product
    cross((a, b), (c, d)) = a * d - b * c, which equals the orientation of the
    two directions up to the positive constant sqrt(3) / 2 and is therefore an
    exact integer (no floating-point phase is computed). A vertex lying on a
    primary ray is assigned to the sector counterclockwise of that ray, so the
    six sectors tile Z_6 without overlap. This rule is exactly equivariant
    under the order-6 lattice rotation---a vertex and its 60-degree image
    differ by exactly one sector---so the resulting sector partition, and the
    s_6 parity of the trihexagonal six-coloring built from it, are independent
    of any floating-point computation.
    """
    for t in range(6):
        dm, dn = SECTOR_RAYS[t]
        em, en = SECTOR_RAYS[(t + 1) % 6]
        # p is counterclockwise of (or on) ray t, and strictly clockwise of
        # ray t+1: this places on-ray vertices in the counterclockwise sector.
        if dm * n - dn * m >= 0 and m * en - n * em > 0:
            return t
    raise ValueError(
        f"angular_sector_index: undefined for origin/invalid vertex ({m}, {n})"
    )


def trihexagonal_six_coloring(node):
    """Return the trihexagonal six-coloring e_6 of a lattice vertex.

    Implements e_6 = 2 * c + (s_6 mod 2), where c = (m - n) mod 3 is the
    proper three-coloring of the triangular lattice in Eisenstein coordinates
    (m, n), and s_6 is the angular sector index. The result lies in {0,...,5}
    and is a proper coloring of the lattice graph: adjacent vertices receive
    distinct colors, so each color class is an independent set.
    """
    m, n, _ = node
    c = (m - n) % 3
    s6_parity = angular_sector_index(m, n) % 2
    return 2 * c + s6_parity


def build_relaxation_arrays(G):
    """Build the index arrays needed for batched neighbor relaxation.

    Produces a stable vertex ordering, a padded neighbor-index matrix, a
    neighbor-count vector, and the six independent color-class index lists
    induced by the trihexagonal six-coloring. The padded neighbor matrix has
    one row per vertex and a fixed number of columns equal to the maximum
    degree; unused entries are padded so that a single gather over the matrix
    yields every vertex's neighbor states at once.

    Args:
        G: The complete truncated radial dual triangular lattice graph.

    Returns:
        dict with keys:
          'num_vertices' : vertex count.
          'neighbor_idx' : int array, shape (num_vertices, max_degree).
          'neighbor_cnt' : int array, shape (num_vertices,).
```

```
          'color_classes': list of six int arrays of vertex indices.
          'proper'       : bool, True if the six-coloring is verified proper.
    """
    nodes = list(G.nodes())
    index_of = {node: i for i, node in enumerate(nodes)}
    num_vertices = len(nodes)

    neighbor_lists = [
        [index_of[v] for v in G.neighbors(node)] for node in nodes
    ]
    max_degree = max((len(nl) for nl in neighbor_lists), default=0)

    # Padded neighbor-index matrix. Padding entries point back at the vertex
    # itself; the neighbor-count vector masks them out of the mean reduction.
    neighbor_idx = np.zeros((num_vertices, max_degree), dtype=np.int64)
    neighbor_cnt = np.zeros(num_vertices, dtype=np.int64)
    for i, nl in enumerate(neighbor_lists):
        neighbor_cnt[i] = len(nl)
        for j, nbr in enumerate(nl):
            neighbor_idx[i, j] = nbr
        for j in range(len(nl), max_degree):
            neighbor_idx[i, j] = i

    # Partition vertices into the six independent color classes of e_6.
    color_of = np.array(
        [trihexagonal_six_coloring(node) for node in nodes], dtype=np.int64
    )
    color_classes = [
        np.where(color_of == color)[0] for color in range(6)
    ]

    # Verify that the six-coloring is proper: no edge joins same-colored
    # vertices. Independence of each color class is what licenses
    # conflict-free parallel updates within a class.
    proper = all(
        color_of[index_of[u]] != color_of[index_of[v]]
        for u, v in G.edges()
    )

    return {
        "num_vertices": num_vertices,
        "neighbor_idx": neighbor_idx,
        "neighbor_cnt": neighbor_cnt,
        "color_classes": color_classes,
        "proper": proper,
    }


def relaxation_sweep_cpu(state, arrays, alpha=0.5):
    """Perform one color-ordered relaxation sweep on the CPU with NumPy.

    Each vertex is updated to alpha * (own state) + (1 - alpha) * (mean of
    neighbor states). The six color classes are visited in turn; a class reads
    the updates already written by earlier classes in the same sweep
    (Gauss-Seidel ordering). Within a single class the vertices are mutually
    non-adjacent because the six-coloring is proper, so their updates are
    independent and applied as one batched array operation. The input array is
    not modified; the updated state array is returned.
    """
    neighbor_idx = arrays["neighbor_idx"]
    neighbor_cnt = arrays["neighbor_cnt"]
    state = state.copy()
    for cls in arrays["color_classes"]:
        if cls.size == 0:
            continue
```

```
        gathered = state[neighbor_idx[cls]]
        counts = np.maximum(neighbor_cnt[cls], 1).astype(state.dtype)
        neighbor_mean = gathered.sum(axis=1) / counts
        state[cls] = alpha * state[cls] + (1.0 - alpha) * neighbor_mean
    return state


def relaxation_sweep_gpu(state, neighbor_idx, neighbor_cnt,
                         color_classes, alpha=0.5):
    """Perform one color-ordered relaxation sweep with batched PyTorch tensors.

    Equivalent to relaxation_sweep_cpu: the six color classes are visited in
    turn with Gauss-Seidel ordering, and within each class the independent
    updates run as one batched tensor operation on the device holding the
    input tensors (a CUDA device when one is available). The input tensor is
    not modified; the updated state tensor is returned.
    """
    state = state.clone()
    for cls in color_classes:
        if cls.numel() == 0:
            continue
        gathered = state[neighbor_idx[cls]]
        counts = torch.clamp(neighbor_cnt[cls], min=1).to(state.dtype)
        neighbor_mean = gathered.sum(dim=1) / counts
        state[cls] = alpha * state[cls] + (1.0 - alpha) * neighbor_mean
    return state


def run_cpu(arrays, sweeps, runs, timing_repeats):
    """Benchmark the CPU backend; return (result_state, mean_ms, std_ms).

    The reported time is the mean wall-clock time per single relaxation sweep.
    """
    num_vertices = arrays["num_vertices"]
    initial = np.linspace(0.0, 1.0, num_vertices, dtype=np.float64)

    result_state = initial.copy()
    for _ in range(sweeps):
        result_state = relaxation_sweep_cpu(result_state, arrays)

    times = []
    for _ in range(runs):
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            state = initial.copy()
            for _ in range(sweeps):
                state = relaxation_sweep_cpu(state, arrays)
        elapsed = (time.perf_counter() - t0) * 1000
        times.append(elapsed / (timing_repeats * sweeps))
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return result_state, statistics.mean(times), std


def run_gpu(arrays, sweeps, runs, timing_repeats):
    """Benchmark the GPU backend; return (result_state, mean_ms, std_ms, device).

    Moves the index tensors and state to the CUDA device when available (CPU
    PyTorch device otherwise) and times batched relaxation sweeps there. The
    reported time is the mean wall-clock time per single relaxation sweep and
    includes CUDA synchronization so the measurement reflects completed device
    work. The result state is returned on the CPU as a NumPy array.
    """
    device = torch.device(
        "cuda" if torch.cuda.is_available() else "cpu"
    )
```

```
    num_vertices = arrays["num_vertices"]

    neighbor_idx = torch.from_numpy(arrays["neighbor_idx"]).to(device)
    neighbor_cnt = torch.from_numpy(arrays["neighbor_cnt"]).to(device)
    color_classes = [
        torch.from_numpy(cls).to(device) for cls in arrays["color_classes"]
    ]
    initial = torch.linspace(
        0.0, 1.0, num_vertices, dtype=torch.float64, device=device
    )

    def synchronize():
        if device.type == "cuda":
            torch.cuda.synchronize()

    state = initial.clone()
    for _ in range(sweeps):
        state = relaxation_sweep_gpu(
            state, neighbor_idx, neighbor_cnt, color_classes
        )
    synchronize()
    result_state = state.cpu().numpy()

    times = []
    for _ in range(runs):
        synchronize()
        t0 = time.perf_counter()
        for _ in range(timing_repeats):
            state = initial.clone()
            for _ in range(sweeps):
                state = relaxation_sweep_gpu(
                    state, neighbor_idx, neighbor_cnt, color_classes
                )
        synchronize()
        elapsed = (time.perf_counter() - t0) * 1000
        times.append(elapsed / (timing_repeats * sweeps))
    std = statistics.stdev(times) if len(times) > 1 else 0.0
    return result_state, statistics.mean(times), std, device.type


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description=(
            "Benchmark conflict-free parallel relaxation on the complete "
            "truncated radial dual triangular lattice graph Lambda_r^R, "
            "scheduled by the equivariant trihexagonal six-coloring, on a "
            "CPU baseline versus a GPU backend. Times are in milliseconds "
            "per sweep."
        )
    )
    parser.add_argument("R", type=int, nargs="?", default=100,
                        help="Truncation radius R (default: 100)")
    parser.add_argument("--runs", type=int, default=10,
                        help="Number of benchmark runs (default: 10)")
    parser.add_argument("--timing_repeats", type=int, default=20,
                        help="Repeats per run for accuracy (default: 20)")
    parser.add_argument("--sweeps", type=int, default=10,
                        help="Relaxation sweeps per timed iteration "
                             "(default: 10)")
    args = parser.parse_args()

    G, _ = build_complete_lattice_graph(args.R)
    arrays = build_relaxation_arrays(G)
    print(f"Graph: |V|={arrays['num_vertices']} |E|={len(G.edges())}")
    print(f"Trihexagonal six-coloring proper: "
```

```
          f"{'PASS' if arrays['proper'] else 'FAIL'}")
    class_sizes = [int(cls.size) for cls in arrays["color_classes"]]
    print(f"Six-coloring class sizes: {class_sizes}")

    cpu_state, cpu_avg, cpu_std = run_cpu(
        arrays, args.sweeps, args.runs, args.timing_repeats
    )
    print(f"CPU (NumPy, color-ordered): {cpu_avg:.4f} ms/sweep "
          f"+/- {cpu_std:.4f}")

    if not TORCH_AVAILABLE:
        print("PyTorch not installed; GPU backend skipped. "
              "Install via: pip install torch")
    else:
        gpu_state, gpu_avg, gpu_std, device = run_gpu(
            arrays, args.sweeps, args.runs, args.timing_repeats
        )
        print(f"GPU backend device: {device}")

        # Verify the GPU result matches the CPU result before reporting speed.
        max_abs_diff = float(np.max(np.abs(cpu_state - gpu_state)))
        agree = max_abs_diff < 1e-9
        print(f"CPU/GPU agreement: {'PASS' if agree else 'FAIL'} "
              f"(max abs diff {max_abs_diff:.2e})")
        print(f"GPU (PyTorch, color-ordered): {gpu_avg:.4f} ms/sweep "
              f"+/- {gpu_std:.4f}")
        if gpu_avg > 0:
            print(f"Speedup (CPU / GPU): {cpu_avg / gpu_avg:.2f}x")
```