Geometric VAE Framework for Superconductor Discovery

A Physics-Informed Approach to High-Tc Material Discovery

Technical Reference Document for Claude Code

Based on Entropy Saturation Theory

Table of Contents

---

1. Executive Summary

1.1 The Core Vision

This framework treats the latent space as an **explicit mathematical object** rather than a black-box neural network. The key insight is that superconductivity represents a thermodynamic state where materials have reached "entropy saturation" - all available entropy channels are closed.

The Inversion:

Traditional VAE	Our Framework
Origin = generic metal	Origin = superconducting ground state
Boundary = superconductor (asymptotic)	Outward = growing phase space
Cooling = moving toward boundary	Cooling = retreating toward origin
Tc encoded in decoder	Tc encoded in critical surface geometry

1.2 Key Principles

1. **Explicit over Implicit**: Geometry stored in data structures, not hidden in weights
2. **Universal over Restricted**: Any composition can be evaluated; most return Tc ≈ 0
3. **Discovery-Oriented**: We only need ONE high-Tc material

1.3 What We're Approximating

```
WHAT WE HAVE                    WHAT WE'RE APPROXIMATING
────────────────                ────────────────────────
Chemical formula        →       Many-body wavefunction
(discrete symbols)              (continuous, infinite-dimensional)

Magpie features         →       Electronic structure
(~150 numbers)                  (bands, Fermi surface, phonons)

Physics intuition       →       Full quantum statistical mechanics
(geometric priors)              (partition functions, phase transitions)
```

---

2. Theoretical Framework

2.1 Entropy Saturation Theory

Superconductivity emerges when a material reaches a thermodynamic state where resistance becomes impossible because it would require entropy production through unavailable channels.

Key Concepts:

- **Entropy Channels**: Mechanisms by which a system can increase entropy (phonon scattering, defect scattering, etc.)
- **Saturation**: State where available entropy channels are "full" or "closed"
- **Meissner Effect**: Interpreted as expulsion of magnetic monopoles to maintain topological triviality

2.2 The Growth Picture

Instead of superconductors approaching a boundary, we view them as **seeds that grow**:

```
Temperature = 0:  Point at origin. Perfect order. No phase space.
                  All entropy channels closed. Topologically trivial.

Temperature > 0:  Region around origin. Thermal fluctuations create
                  a "bubble" of accessible states. Bubble grows with T.

T = Tc:           Bubble reaches critical surface. Topology changes.
                  First defects become possible. Superconductivity breaks.

T > Tc:           Beyond critical surface. Full phase space accessible.
                  Monopoles, vortices, resistance all possible.
```

2.3 High-Tc Implications

High-Tc materials have **critical surfaces far from the origin** - their superconducting seeds can grow larger before topology breaks. The "quality" of a superconducting state is its **extensibility**.

Extensibility Factors: - Gap magnitude Δ_0 (depth of energy well) - Coherence length ξ (spatial extent of Cooper pairs) - Superfluid stiffness ρ_s (resistance to phase fluctuations) - Pairing symmetry - Defect tolerance

---

3. Geometric Structures

3.1 Overview

The geometry is stored in explicit, queryable data structures:

Component	What It Stores	Physical Meaning
MetricTensor	g_ij(x)	How to measure distances; stiffness of states
CriticalSurface	Φ(x) = 0	Where superconductivity breaks; encodes Tc
FiberStructure	Base × Fiber	Composition × thermal state

3.2 Metric Tensor

```
@dataclass
class MetricTensor:
    """
    Explicit representation of a Riemannian metric.

    The metric tensor g_ij(x) defines:
    - Distances: ds² = g_ij dx^i dx^j
    - Angles: cos(θ) = g(u,v) / (|u||v|)
    - Volume: dV = √det(g) dx¹...dxⁿ
    """

    dim: int
    metric_type: str = "diagonal"  # "diagonal", "full", "conformal", "learned"

    # For diagonal metric: g_ij = diag(g_11, g_22, ...)
    diagonal: torch.Tensor = None

    # For full metric: arbitrary symmetric positive definite
    full_matrix: torch.Tensor = None

    # For conformal metric: g_ij = f(x)² δ_ij
    conformal_factor: Callable = None

    # For learned metric: neural network
    metric_network: nn.Module = None

    def at_point(self, x: torch.Tensor) -> torch.Tensor:
        """
        Evaluate metric tensor at point x.
        Returns: [dim, dim] tensor (or [batch, dim, dim] if batched)
        """
        if self.metric_type == "diagonal":
            return torch.diag(self.diagonal)

        elif self.metric_type == "full":
            return self.full_matrix

        elif self.metric_type == "conformal":
            f = self.conformal_factor(x)
            return f**2 * torch.eye(self.dim, device=x.device)

        elif self.metric_type == "learned":
            raw = self.metric_network(x)
            # Ensure positive definite via Cholesky parameterization
            L = torch.zeros(self.dim, self.dim, device=x.device)
            idx = 0
            for i in range(self.dim):
                for j in range(i + 1):
                    L[i, j] = raw[..., idx]
                    idx += 1
                L[i, i] = torch.exp(L[i, i])  # Positive diagonal
            return L @ L.T

    def inner_product(self, u: torch.Tensor, v: torch.Tensor,
                      x: torch.Tensor) -> torch.Tensor:
        """Compute <u, v>_x = u^T g(x) v"""
        g = self.at_point(x)
        return torch.einsum('...i,...ij,...j->...', u, g, v)

    def norm(self, v: torch.Tensor, x: torch.Tensor) -> torch.Tensor:
```

```
        """Compute ||v||_x = sqrt(<v,v>_x)"""
        return torch.sqrt(self.inner_product(v, v, x))

    def determinant(self, x: torch.Tensor) -> torch.Tensor:
        """Compute det(g) - needed for volume element."""
        g = self.at_point(x)
        return torch.linalg.det(g)
```

3.3 Critical Surface

```
@dataclass
class CriticalSurface:
    """
    The boundary where superconductivity breaks.
    Defined as {x : Φ(x) = 0} for level set function Φ.

    Tc is encoded in the SHAPE of this surface.
    """

    dim: int
    surface_type: str = "radial"  # "radial", "ellipsoidal", "learned", "directional"

    # For radial surface: Φ(x) = R - ||x||
    critical_radius: torch.Tensor = None

    # For ellipsoidal: Φ(x) = 1 - x^T A x
    ellipsoid_matrix: torch.Tensor = None

    # For learned surface: neural network defining Φ
    level_set_network: nn.Module = None

    # For direction-dependent radius: R(θ)
    radius_by_direction: nn.Module = None

    def level_set(self, x: torch.Tensor) -> torch.Tensor:
        """
        Evaluate level set function Φ(x).

        Φ > 0: inside superconducting region
        Φ = 0: on critical surface
        Φ < 0: outside (normal state)
        """
        if self.surface_type == "radial":
            r = torch.norm(x, dim=-1)
            return self.critical_radius - r

        elif self.surface_type == "ellipsoidal":
            quad = torch.einsum('...i,ij,...j->...', x, self.ellipsoid_matrix, x)
            return 1 - quad

        elif self.surface_type == "learned":
            return self.level_set_network(x).squeeze(-1)

        elif self.surface_type == "directional":
            direction = x / (torch.norm(x, dim=-1, keepdim=True) + 1e-8)
            R = self.radius_by_direction(direction).squeeze(-1)
            r = torch.norm(x, dim=-1)
            return R - r

    def critical_radius_in_direction(self, direction: torch.Tensor) -> torch.Tensor:
        """
        Get critical radius along a direction.
        This is where Tc information is encoded!
        """
        direction = direction / (torch.norm(direction, dim=-1, keepdim=True) + 1e-8)

        if self.surface_type == "radial":
            return self.critical_radius.expand(direction.shape[0])

        elif self.surface_type == "ellipsoidal":
```

```
            quad = torch.einsum('...i,ij,...j->...', direction,
                                self.ellipsoid_matrix, direction)
            return 1 / torch.sqrt(quad)

        elif self.surface_type == "directional":
            return self.radius_by_direction(direction).squeeze(-1)
```

3.4 Fiber Bundle Structure

```
@dataclass
class FiberStructure:
    """
    Fiber bundle structure for superconductor space.

    Total space E = ⋃_b F_b (union of fibers over base points)

    Base space B: composition space (what material)
    Fiber F: growth manifold (thermal/phase state)
    Projection π: E → B
    """

    base_dim: int   # Dimension of base space (composition embedding)
    fiber_dim: int  # Dimension of fiber (growth space)

    # Fiber metric (can vary over base)
    fiber_metric: MetricTensor = None

    # Whether fiber metric depends on base point
    variable_fiber_metric: bool = False
    fiber_metric_function: Callable = None

    def decompose(self, total_point: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """Decompose total space point into (base, fiber) coordinates."""
        base = total_point[..., :self.base_dim]
        fiber = total_point[..., self.base_dim:]
        return base, fiber

    def compose(self, base: torch.Tensor, fiber: torch.Tensor) -> torch.Tensor:
        """Compose (base, fiber) into total space point."""
        return torch.cat([base, fiber], dim=-1)
```

3.5 Complete Geometry Object

```
@dataclass
class SuperconductorGeometry:
    """
    Complete geometric structure for superconductor space.
    This is the master object that holds ALL geometric data.
    Everything is explicit and queryable.
    """

    composition_dim: int
    fiber_dim: int

    metric: MetricTensor = None
    critical_surface: CriticalSurface = None
    fiber_structure: FiberStructure = None

    geometry_type: str = "fiber_bundle"

    @property
    def total_dim(self) -> int:
        return self.composition_dim + self.fiber_dim

    def distance(self, x1: torch.Tensor, x2: torch.Tensor) -> torch.Tensor:
        """Geodesic distance between points using the metric tensor."""
        n_steps = 20
        t = torch.linspace(0, 1, n_steps, device=x1.device)

        path = x1.unsqueeze(-2) + t.unsqueeze(-1) * (x2 - x1).unsqueeze(-2)
        tangent = (x2 - x1).unsqueeze(-2).expand_as(path)

        lengths = []
        for i in range(n_steps):
            g = self.metric.at_point(path[..., i, :])
            length = torch.sqrt(torch.einsum('...i,...ij,...j->...',
                                             tangent[..., i, :], g, tangent[..., i, :]))
            lengths.append(length)

        lengths = torch.stack(lengths, dim=-1)
        return torch.trapezoid(lengths, t) / n_steps

    def is_superconducting(self, x: torch.Tensor) -> torch.Tensor:
        """Check if point is in superconducting region."""
        return self.critical_surface.level_set(x) > 0

    def tc_at_direction(self, direction: torch.Tensor) -> torch.Tensor:
        """Get Tc for a composition (direction in base space)."""
        if direction.shape[-1] == self.composition_dim:
            full_direction = torch.cat([
                direction,
                torch.zeros(*direction.shape[:-1], self.fiber_dim, device=direction.device)
            ], dim=-1)
        else:
            full_direction = direction

        R_c = self.critical_surface.critical_radius_in_direction(full_direction)
        return R_c * 100  # Scale to Kelvin

    def volume_element(self, x: torch.Tensor) -> torch.Tensor:
        """Volume element √det(g) at point x."""
        return torch.sqrt(self.metric.determinant(x))
```

4. Compositional Encoding

4.1 The Problem with Simple Angular Encoding

A simple direction vector in $\mathbb{R}^n$ treats angular space as homogeneous and continuous, but chemical composition is neither:

- Elements are **discrete** (~100, not a continuum)
- Stoichiometry is **discrete** (integer ratios)
- Not all directions correspond to real materials
- "Nearby" in angle ≠ "nearby" in chemistry

4.2 Universal Composition Space

```
class UniversalCompositionSpace(nn.Module):
    """
    Universal embedding for ANY chemical composition.

    Key principle: Every composition gets a location.
    No restrictions, no boundaries, no "invalid" regions.
    """

    def __init__(self,
                 embedding_dim: int = 128,
                 element_embedding_dim: int = 64):
        super().__init__()

        self.embedding_dim = embedding_dim
        self.num_elements = 118

        # Element embeddings initialized with periodic table structure
        self.element_embeddings = nn.Parameter(
            self._init_periodic_embeddings(element_embedding_dim)
        )

        # Attention-based aggregation
        self.element_attention = nn.MultiheadAttention(
            embed_dim=element_embedding_dim,
            num_heads=4,
            batch_first=True
        )

        # Final projection
        self.composition_proj = nn.Sequential(
            nn.Linear(element_embedding_dim, embedding_dim),
            nn.LayerNorm(embedding_dim),
            nn.SiLU(),
            nn.Linear(embedding_dim, embedding_dim),
        )

    def _init_periodic_embeddings(self, dim: int) -> torch.Tensor:
        """Initialize with periodic table structure."""
        embeddings = torch.zeros(self.num_elements + 1, dim)

        for z in range(1, self.num_elements + 1):
            embed = torch.zeros(dim)

            # Atomic number (scaled)
            embed[0] = z / 118.0

            # Period (1-7)
            period = self._get_period(z)
            embed[1:8] = F.one_hot(torch.tensor(period - 1), 7).float()

            # Group (1-18)
            group = self._get_group(z)
            embed[8:26] = F.one_hot(torch.tensor(group - 1), 18).float()

            # Block (s=0, p=1, d=2, f=3)
            block = self._get_block(z)
            embed[26:30] = F.one_hot(torch.tensor(block), 4).float()

            # Electronegativity
            en = self._get_electronegativity(z)
```

```
            embed[30] = en / 4.0

            # Random fine-tuning component
            embed[31:] = torch.randn(dim - 31) * 0.1

            embeddings[z] = embed

        return embeddings

    def embed(self,
              elements: torch.Tensor,
              fractions: torch.Tensor) -> torch.Tensor:
        """
        Embed any composition into the universal space.

        Args:
            elements: [batch, max_elements] element atomic numbers (0 = padding)
            fractions: [batch, max_elements] atomic fractions (sum to 1)
        """
        # Get element embeddings
        elem_embeds = self.element_embeddings[elements]

        # Mask padding
        mask = (elements == 0)

        # Weight by fractions
        weighted_embeds = elem_embeds * fractions.unsqueeze(-1)

        # Self-attention for element interactions
        attended, _ = self.element_attention(
            weighted_embeds, weighted_embeds, weighted_embeds,
            key_padding_mask=mask
        )

        # Aggregate
        aggregated = (attended * fractions.unsqueeze(-1)).sum(dim=1)

        # Project
        return self.composition_proj(aggregated)

    def _get_period(self, z):
        if z <= 2: return 1
        if z <= 10: return 2
        if z <= 18: return 3
        if z <= 36: return 4
        if z <= 54: return 5
        if z <= 86: return 6
        return 7

    def _get_group(self, z):
        return ((z - 1) % 18) + 1

    def _get_block(self, z):
        if z in [1, 2] or (3 <= z <= 4) or (11 <= z <= 12) or (19 <= z <= 20):
            return 0  # s-block
        if z in range(57, 72) or z in range(89, 104):
            return 3  # f-block
        if z in range(21, 31) or z in range(39, 49) or z in range(72, 81):
            return 2  # d-block
        return 1  # p-block
```

```
    def _get_electronegativity(self, z):
        en_table = {1: 2.20, 6: 2.55, 7: 3.04, 8: 3.44, 26: 1.83, 29: 1.90}
        return en_table.get(z, 1.5)
```

---

5. The Universal Model

5.1 Architecture Overview

```
Input: Formula + Magpie data
         ↓
Encoder: → Composition embedding (direction in latent space)
         ↓
Geometry: → Critical radius in that direction (learned surface)
         ↓
Output: Tc = f(critical_radius)
```

5.2 Learnable Geometry Module

```
class LearnableGeometry(nn.Module):
    """
    Learnable geometry where structure is explicit.

    Neural networks PARAMETERIZE the geometry,
    but geometric STRUCTURE is always explicit and inspectable.
    """

    def __init__(self,
                 composition_dim: int = 64,
                 fiber_dim: int = 16):
        super().__init__()

        self.composition_dim = composition_dim
        self.fiber_dim = fiber_dim
        self.total_dim = composition_dim + fiber_dim

        # === LEARNABLE METRIC ===
        # Diagonal + low-rank: g = diag(d) + V V^T
        self.metric_diagonal = nn.Parameter(torch.ones(self.total_dim))
        self.metric_lowrank = nn.Parameter(torch.randn(self.total_dim, 4) * 0.1)

        # Position-dependent conformal factor
        self.conformal_network = nn.Sequential(
            nn.Linear(self.total_dim, 32),
            nn.Softplus(),
            nn.Linear(32, 1),
            nn.Softplus(),
        )

        # === LEARNABLE CRITICAL SURFACE ===
        # Direction-dependent critical radius R(θ) - THIS IS WHERE Tc LIVES
        self.critical_radius_network = nn.Sequential(
            nn.Linear(self.total_dim, 64),
            nn.SiLU(),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.Linear(32, 1),
            nn.Softplus(),
        )

        # Base critical radius
        self.log_base_radius = nn.Parameter(torch.tensor(0.0))

        # === LEARNABLE FIBER STRUCTURE ===
        self.fiber_metric_network = nn.Sequential(
            nn.Linear(composition_dim, 32),
            nn.SiLU(),
            nn.Linear(32, fiber_dim),
            nn.Softplus(),
        )

    def get_geometry(self) -> SuperconductorGeometry:
        """Extract explicit geometry object from learned parameters."""
        metric = MetricTensor(dim=self.total_dim, metric_type="learned")
        critical_surface = CriticalSurface(dim=self.total_dim, surface_type="directional")
        fiber_structure = FiberStructure(
            base_dim=self.composition_dim,
            fiber_dim=self.fiber_dim,
            variable_fiber_metric=True,
```

```
        )

        geom = SuperconductorGeometry(
            composition_dim=self.composition_dim,
            fiber_dim=self.fiber_dim,
            metric=metric,
            critical_surface=critical_surface,
            fiber_structure=fiber_structure,
        )

        # Attach learned functions
        geom.metric.metric_network = self._metric_at_point
        geom.critical_surface.radius_by_direction = self.critical_radius_network

        return geom

    def _metric_at_point(self, x: torch.Tensor) -> torch.Tensor:
        """Compute metric tensor at point."""
        base_metric = torch.diag(torch.exp(self.metric_diagonal))
        base_metric = base_metric + self.metric_lowrank @ self.metric_lowrank.T
        conformal = self.conformal_network(x)
        return conformal.unsqueeze(-1).unsqueeze(-1) * base_metric

    def critical_radius(self, direction: torch.Tensor) -> torch.Tensor:
        """Get critical radius in direction."""
        base_R = torch.exp(self.log_base_radius)
        learned_R = self.critical_radius_network(direction).squeeze(-1)
        return base_R + learned_R

    def level_set(self, x: torch.Tensor) -> torch.Tensor:
        """Level set function Φ(x)."""
        r = torch.norm(x, dim=-1)
        direction = x / (r.unsqueeze(-1) + 1e-8)
        R = self.critical_radius(direction)
        return R - r

    def predict_tc(self, composition_embedding: torch.Tensor) -> torch.Tensor:
        """Predict Tc from composition embedding."""
        if composition_embedding.shape[-1] == self.composition_dim:
            full = torch.cat([
                composition_embedding,
                torch.zeros(*composition_embedding.shape[:-1], self.fiber_dim,
                            device=composition_embedding.device)
            ], dim=-1)
        else:
            full = composition_embedding

        direction = full / (torch.norm(full, dim=-1, keepdim=True) + 1e-8)
        R_c = self.critical_radius(direction)
        return R_c * 100  # Scale to Kelvin

    # === INSPECTION METHODS ===

    def inspect_metric(self) -> dict:
        """Inspect learned metric structure."""
        return {
            'diagonal': torch.exp(self.metric_diagonal).detach(),
            'lowrank_contribution': (self.metric_lowrank @ self.metric_lowrank.T).detach(),
        }

    def inspect_critical_surface(self, n_directions: int = 100) -> dict:
```

```
        """Inspect critical surface shape."""
        directions = torch.randn(n_directions, self.total_dim)
        directions = directions / torch.norm(directions, dim=-1, keepdim=True)

        radii = self.critical_radius(directions).detach()

        return {
            'mean_radius': radii.mean().item(),
            'std_radius': radii.std().item(),
            'min_radius': radii.min().item(),
            'max_radius': radii.max().item(),
            'anisotropy': (radii.max() / radii.min()).item(),
        }
```

5.3 Full Model

```
class GeometricSuperconductorModel(nn.Module):
    """
    Full model where geometry is EXPLICIT.
    """

    def __init__(self,
                 composition_dim: int = 64,
                 fiber_dim: int = 16):
        super().__init__()

        self.composition_encoder = UniversalCompositionSpace(
            embedding_dim=composition_dim
        )

        self.geometry = LearnableGeometry(
            composition_dim=composition_dim,
            fiber_dim=fiber_dim
        )

    def encode_composition(self,
                           elements: torch.Tensor,
                           fractions: torch.Tensor) -> torch.Tensor:
        """Encode composition to geometric embedding."""
        return self.composition_encoder.embed(elements, fractions)

    def predict_tc(self,
                   elements: torch.Tensor,
                   fractions: torch.Tensor) -> torch.Tensor:
        """Predict Tc via geometry."""
        embedding = self.encode_composition(elements, fractions)
        return self.geometry.predict_tc(embedding)

    def get_geometry(self) -> SuperconductorGeometry:
        """Get explicit geometry object."""
        return self.geometry.get_geometry()

    def geometric_analysis(self,
                           elements: torch.Tensor,
                           fractions: torch.Tensor) -> dict:
        """Full geometric analysis of a composition."""
        embedding = self.encode_composition(elements, fractions)

        full = torch.cat([
            embedding,
            torch.zeros(*embedding.shape[:-1], self.geometry.fiber_dim,
                        device=embedding.device)
        ], dim=-1)

        direction = full / (torch.norm(full, dim=-1, keepdim=True) + 1e-8)

        return {
            'embedding': embedding.detach(),
            'direction': direction.detach(),
            'critical_radius': self.geometry.critical_radius(direction).detach(),
            'Tc': self.geometry.predict_tc(embedding).detach(),
        }

    def inspect(self) -> dict:
        """Full inspection of learned geometry."""
        return {
```

```
            'metric': self.geometry.inspect_metric(),
            'critical_surface': self.geometry.inspect_critical_surface(),
        }
```

---

6. Physics-Informed Search

6.1 Overview

The neural network provides fast evaluation, but exploration should be guided by structured domain knowledge:

```
PRIORS WE HAVE:
├── Geometric (entropy saturation theory)
├── Chemical (periodic table, bonding)
└── Empirical (known superconductor patterns)

ALGORITHMS:
├── Gaussian Process (uncertainty-aware)
├── Bayesian Optimization (acquisition functions)
├── Evolutionary (population-based)
├── MCTS (sequential composition building)
└── Hybrid controller
```

6.2 Chemistry-Informed Kernel

```
class ChemistryInformedKernel:
    """
    Kernel for Gaussian Process that encodes chemical knowledge.

    k(x, x') = similarity based on:
    - Element similarity (periodic table)
    - Stoichiometry patterns
    - Valence electron count
    """

    def __init__(self):
        self.element_similarity = self._build_element_similarity()
        self.length_scale = 1.0
        self.variance = 1.0
        self.element_weight = 0.4
        self.stoich_weight = 0.3
        self.valence_weight = 0.3

    def _build_element_similarity(self) -> np.ndarray:
        """Build element-element similarity matrix."""
        n_elements = 96
        similarity = np.zeros((n_elements, n_elements))

        for z1 in range(1, n_elements):
            for z2 in range(1, n_elements):
                sim = self._element_pair_similarity(z1, z2)
                similarity[z1, z2] = sim

        return similarity

    def _element_pair_similarity(self, z1: int, z2: int) -> float:
        """Similarity between two elements."""
        if z1 == z2:
            return 1.0

        g1, p1, b1 = self._get_element_properties(z1)
        g2, p2, b2 = self._get_element_properties(z2)

        group_sim = 1.0 if g1 == g2 else 0.3
        period_sim = 1.0 if p1 == p2 else 0.5
        block_sim = 1.0 if b1 == b2 else 0.2

        en1 = self._get_electronegativity(z1)
        en2 = self._get_electronegativity(z2)
        en_sim = np.exp(-abs(en1 - en2) / 0.5)

        return 0.3 * group_sim + 0.2 * period_sim + 0.2 * block_sim + 0.3 * en_sim

    def __call__(self, X1: List[dict], X2: List[dict] = None) -> np.ndarray:
        """Compute kernel matrix."""
        if X2 is None:
            X2 = X1

        n1, n2 = len(X1), len(X2)
        K = np.zeros((n1, n2))

        for i, x1 in enumerate(X1):
            for j, x2 in enumerate(X2):
                K[i, j] = self._kernel_single(x1, x2)
```

```
        return K

    def _kernel_single(self, x1: dict, x2: dict) -> float:
        """Kernel between two compositions."""
        elem_sim = self._element_set_similarity(x1['elements'], x2['elements'])
        stoich_sim = self._stoichiometry_similarity(x1, x2)
        valence_sim = self._valence_similarity(x1, x2)

        distance_sq = (
            self.element_weight * (1 - elem_sim) +
            self.stoich_weight * (1 - stoich_sim) +
            self.valence_weight * (1 - valence_sim)
        )

        return self.variance * np.exp(-distance_sq / (2 * self.length_scale**2))
```

6.3 Physics Prior

```
class PhysicsPrior:
    """
    Physics-informed prior probability for compositions.

    Returns P(composition is worth exploring) ∈ [0, 1]
    """

    def __init__(self):
        self.templates = self._build_templates()

    def __call__(self, x: dict) -> float:
        """Compute prior probability for composition."""
        scores = []

        scores.append(self._chemical_validity_score(x))
        scores.append(self._template_similarity_score(x))
        scores.append(self._entropy_saturation_score(x))
        scores.append(self._electron_count_score(x))

        return np.exp(np.mean(np.log(np.array(scores) + 1e-8)))

    def _chemical_validity_score(self, x: dict) -> float:
        """Score based on chemical plausibility."""
        elements = x['elements']
        fractions = x['fractions']

        ens = [self._get_electronegativity(e) for e in elements]
        en_spread = max(ens) - min(ens)
        en_score = np.exp(-((en_spread - 1.5) / 1.0)**2)

        return en_score

    def _entropy_saturation_score(self, x: dict) -> float:
        """Score based on entropy saturation geometry."""
        elements = x['elements']
        fractions = x['fractions']

        # Transition metal content
        d_block = {21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                   39, 40, 41, 42, 43, 44, 45, 46, 47, 48}
        d_fraction = sum(f for e, f in zip(elements, fractions) if e in d_block)
        d_score = 1.0 if 0.1 < d_fraction < 0.6 else 0.5

        # Chalcogen content
        chalcogens = {8, 16, 34, 52}
        chalc_fraction = sum(f for e, f in zip(elements, fractions) if e in chalcogens)
        chalc_score = 1.0 if 0.2 < chalc_fraction < 0.7 else 0.5

        # Complexity
        n_elements = len([e for e in elements if e > 0])
        complexity_score = 1.0 if 2 <= n_elements <= 5 else 0.5

        return (d_score * chalc_score * complexity_score) ** (1/3)

    def _build_templates(self) -> List[dict]:
        """Known superconductor family templates."""
        return [
            {'name': 'cuprate', 'required_elements': {29, 8}},
            {'name': 'iron_pnictide', 'required_elements': {26}},
            {'name': 'diboride', 'required_elements': {5}},
```

```
            {'name': 'A15', 'required_elements': set()},
        ]
```

6.4 Bayesian Optimization

```
class PhysicsInformedBayesianOptimization:
    """
    Bayesian Optimization for superconductor discovery.

    Acquisition = (Exploitation + Exploration) × Physics Prior
    """

    def __init__(self,
                 gp: 'ChemistryGP',
                 physics_prior: PhysicsPrior,
                 acquisition: str = 'EI'):

        self.gp = gp
        self.physics_prior = physics_prior
        self.acquisition_type = acquisition
        self.best_y = -np.inf

    def acquisition_function(self, X_candidates: List[dict]) -> np.ndarray:
        """Compute acquisition value for candidates."""
        mean, std = self.gp.predict(X_candidates)

        if self.acquisition_type == 'EI':
            acq = self._expected_improvement(mean, std)
        elif self.acquisition_type == 'UCB':
            acq = mean + 2.0 * std

        # Physics modulation
        physics_weight = np.array([self.physics_prior(x) for x in X_candidates])

        return acq * physics_weight

    def _expected_improvement(self, mean: np.ndarray, std: np.ndarray) -> np.ndarray:
        from scipy.stats import norm

        if self.best_y == -np.inf:
            return mean + std

        z = (mean - self.best_y) / (std + 1e-8)
        return (mean - self.best_y) * norm.cdf(z) + std * norm.pdf(z)
```

6.5 Discovery Engine

```
class SuperconductorDiscoveryEngine:
    """
    Engine for discovering high-Tc materials.

    Key principle: We can go ANYWHERE. Most places are Tc ≈ 0.
    We only need to find ONE high-Tc peak.
    """

    def __init__(self, model: GeometricSuperconductorModel):
        self.model = model
        self.discoveries = []

    def random_sample(self, num_samples: int, max_elements: int = 5) -> List[dict]:
        """Sample random compositions and evaluate Tc."""
        results = []

        for _ in range(num_samples):
            n_elements = torch.randint(1, max_elements + 1, (1,)).item()
            elements = torch.randint(1, 96, (n_elements,))
            fractions = torch.distributions.Dirichlet(torch.ones(n_elements)).sample()

            elements_padded = F.pad(elements, (0, max_elements - n_elements))
            fractions_padded = F.pad(fractions, (0, max_elements - n_elements))

            with torch.no_grad():
                tc = self.model.predict_tc(
                    elements_padded.unsqueeze(0),
                    fractions_padded.unsqueeze(0)
                ).item()

            results.append({
                'elements': elements.tolist(),
                'fractions': fractions.tolist(),
                'Tc': tc,
            })

        results.sort(key=lambda x: x['Tc'], reverse=True)
        return results

    def gradient_ascent(self,
                        start_elements: torch.Tensor,
                        start_fractions: torch.Tensor,
                        num_steps: int = 100,
                        lr: float = 0.01) -> dict:
        """Gradient ascent in fraction space to maximize Tc."""
        fractions = start_fractions.clone().requires_grad_(True)
        optimizer = torch.optim.Adam([fractions], lr=lr)

        trajectory = []

        for step in range(num_steps):
            optimizer.zero_grad()

            valid_fractions = F.softmax(fractions, dim=-1)
            tc = self.model.predict_tc(
                start_elements.unsqueeze(0),
                valid_fractions.unsqueeze(0)
            )

            loss = -tc
```

```
            loss.backward()
            optimizer.step()

            trajectory.append({'step': step, 'Tc': tc.item()})

        return {
            'final_fractions': F.softmax(fractions.detach(), dim=-1),
            'final_Tc': trajectory[-1]['Tc'],
            'trajectory': trajectory,
        }

    def evolutionary_search(self,
                            population_size: int = 100,
                            num_generations: int = 50,
                            mutation_rate: float = 0.1) -> dict:
        """Evolutionary search for high-Tc materials."""
        # Initialize population
        population = self._init_population(population_size)

        history = []

        for gen in range(num_generations):
            fitness = self._evaluate_population(population)

            best_idx = fitness.argmax()
            history.append({
                'generation': gen,
                'best_Tc': fitness[best_idx].item(),
                'mean_Tc': fitness.mean().item(),
            })

            # Selection
            top_k = population_size // 5
            top_indices = fitness.argsort(descending=True)[:top_k]
            parents = population[top_indices]

            # Create new population
            children = self._crossover(parents, population_size // 2)
            mutants = self._mutate(parents, population_size - top_k - len(children), mutation_rate)

            population = torch.cat([parents, children, mutants], dim=0)[:population_size]

        fitness = self._evaluate_population(population)
        best_idx = fitness.argmax()

        return {
            'best_Tc': fitness[best_idx].item(),
            'history': history,
        }
```

---

7. Implementation Guide

7.1 Quick Start

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from dataclasses import dataclass
from typing import List, Tuple, Optional, Callable

# 1. Create model
model = GeometricSuperconductorModel(
    composition_dim=64,
    fiber_dim=16
)

# 2. Prepare data
elements = torch.tensor([[29, 8, 56, 39, 0, 0, 0, 0]])  # YBCO-like
fractions = torch.tensor([[3/13, 7/13, 2/13, 1/13, 0, 0, 0, 0]])

# 3. Predict Tc
tc = model.predict_tc(elements, fractions)
print(f"Predicted Tc: {tc.item():.1f} K")

# 4. Inspect geometry
geom_info = model.inspect()
print(f"Critical surface anisotropy: {geom_info['critical_surface']['anisotropy']:.2f}")

# 5. Get explicit geometry object
geometry = model.get_geometry()
print(f"Is superconducting at origin: {geometry.is_superconducting(torch.zeros(80))}")
```

7.2 Training Loop

```
def train_model(model, train_loader, epochs=100, lr=1e-3):
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)

    for epoch in range(epochs):
        total_loss = 0

        for elements, fractions, tc_true in train_loader:
            optimizer.zero_grad()

            tc_pred = model.predict_tc(elements, fractions)
            loss = F.mse_loss(tc_pred, tc_true)

            # Smoothness regularization
            embedding = model.encode_composition(elements, fractions)
            noise = torch.randn_like(embedding) * 0.01
            tc_perturbed = model.geometry.predict_tc(embedding + noise)
            smoothness_loss = F.mse_loss(tc_pred, tc_perturbed)

            total_loss = loss + 0.1 * smoothness_loss
            total_loss.backward()
            optimizer.step()

        if epoch % 10 == 0:
            print(f"Epoch {epoch}: Loss = {total_loss.item():.4f}")
```

7.3 Discovery Workflow

```
# 1. Initialize
model = GeometricSuperconductorModel(composition_dim=64, fiber_dim=16)
model.load_state_dict(torch.load('trained_model.pt'))

physics_prior = PhysicsPrior()
engine = SuperconductorDiscoveryEngine(model)

# 2. Random exploration
random_results = engine.random_sample(num_samples=1000)
print(f"Best random Tc: {random_results[0]['Tc']:.1f} K")

# 3. Gradient-based refinement of best candidates
for candidate in random_results[:10]:
    elements = torch.tensor(candidate['elements'])
    fractions = torch.tensor(candidate['fractions'])

    result = engine.gradient_ascent(elements, fractions)
    print(f"Optimized Tc: {result['final_Tc']:.1f} K")

# 4. Evolutionary search
evo_result = engine.evolutionary_search(population_size=100, num_generations=50)
print(f"Best evolutionary Tc: {evo_result['best_Tc']:.1f} K")
```

---

8. Complete Code Reference

8.1 Required Imports

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from dataclasses import dataclass, field
from typing import Dict, List, Tuple, Optional, Callable
from enum import Enum
from scipy.spatial.distance import cdist
from scipy.linalg import cholesky, cho_solve
from scipy.stats import norm
```

8.2 Geometry Data Structures

All geometry structures are defined in Section 3.

8.3 Composition Encoding

Full UniversalCompositionSpace defined in Section 4.

8.4 Model Architecture

Full model defined in Section 5.

8.5 Search Algorithms

Full search algorithms defined in Section 6.

Summary

Where is the Geometry Stored?

Aspect	Data Type	Location	Inspectable?
Metric tensor	MetricTensor dataclass	LearnableGeometry parameters	Yes
Critical surface	CriticalSurface dataclass	critical_radius_network	Yes
Fiber structure	FiberStructure dataclass	fiber_metric_network	Yes
Full geometry	SuperconductorGeometry	model.get_geometry()	Yes

Key Insight

Neural networks **parameterize** the geometry, but the geometric **structure** is always explicit and inspectable. This is fundamentally different from a black-box neural network where geometry is hidden in opaque weights.

The Goal

We only need ONE high-Tc material. The system optimizes for peak-finding in a mostly-zero landscape, guided by physics and chemistry priors.

Notes for Future Development

1. **Negative Tc**: May indicate distance to nearest superconducting phase, or metastable states requiring activation. Worth exploring in future work.
2. **Fiber Bundle Topology**: Could compute winding numbers, Chern classes at critical surface.
3. **Variational Formulation**: Define free energy functional; ground state at origin, thermal states as distributions.
4. **Integration with DFT**: Use DFT calculations as oracle for Bayesian optimization.

Document prepared for Claude Code implementation Based on entropy saturation theory of superconductivity