""" DARKBOT™: Resonant Field Intelligence Architecture

Lattice operations module for the DARKBOT™ system.

This file contains implementations for E8 lattice construction and quaternion transformations used in the DARKBOT™ architecture.

import torch import numpy as np from typing import List, Dict, Optional, Any, Tuple, Union

def construct_e8_lattice(device: Optional[torch.device] = None) -> torch.Tensor: """ Construct the full E8 lattice using Conway-Sloane construction.

```
    Creates a 240-vector E8 lattice, which is used as the basis for the
    248-component of the DARKBOT™ architecture.

    Args:
        device: Computation device (CPU or CUDA)

    Returns:
        Tensor containing 240 unit vectors of the E8 lattice
    """
    # Set device if not provided
    if device is None:
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Use float32 for the real-valued lattice points
    dtype = torch.float32

    # === D8 ROOT SYSTEM CONSTRUCTION ===
    # D8 has 112 root vectors: permutations of (±1, ±1, 0, 0, 0, 0, 0, 0)
    d8_vectors = []

    # Generate all pairs of indices (i,j) where i<j
    indices = []
    for i in range(8):
        for j in range(i+1, 8):
            indices.append((i, j))

    # For each pair, create the 4 sign combinations
    for i, j in indices:
        for s1 in [-1, 1]:
            for s2 in [-1, 1]:
                v = torch.zeros(8, dtype=dtype, device=device)
                v[i] = s1
                v[j] = s2
                d8_vectors.append(v)

    # === HALF-INTEGER VECTORS CONSTRUCTION ===
    # E8 adds 128 vectors with half-integer coordinates with even parity
    # (even number of minus signs)

    # Using vectorized operations for efficiency
    # First, generate all 2^8 = 256 possibilities of ±1/2
    half = torch.tensor(0.5, dtype=dtype, device=device)

    # Pre-compute all possible 8-bit patterns
    bit_patterns = torch.zeros((256, 8), dtype=dtype, device=device)
    for i in range(256):
```

```
    # Convert i to binary and create pattern
    bits = format(i, '08b')
    for j in range(8):
        bit_patterns[i, j] = -1 if bits[j] == '1' else 1

# Scale by 0.5 to get half-integer coordinates
half_int_candidates = bit_patterns * half

# Keep only patterns with even parity (even number of -0.5)
# Count number of -0.5 entries in each row
neg_counts = (half_int_candidates < 0).sum(dim=1)

# Even parity means even number of negative entries
even_parity_mask = (neg_counts % 2 == 0)
half_int_vectors = half_int_candidates[even_parity_mask]

# Verify we have exactly 128 half-integer vectors
assert half_int_vectors.shape[0] == 128, f"Expected 128 half-integer vectors, got
{half_int_vectors.shape[0]}"

# === COMBINE D8 AND HALF-INTEGER VECTORS ===
all_vectors = torch.cat([
    torch.stack(d8_vectors),
    half_int_vectors
], dim=0)

# Verify we have exactly 240 vectors (112 from D8 + 128 half-integer)
assert all_vectors.shape[0] == 240, f"E8 construction error: {all_vectors.shape[0]} vectors"

# Normalize all vectors to unit length
norms = torch.norm(all_vectors, dim=1, keepdim=True)
normalized_vectors = all_vectors / norms

# Verify E8 properties:
# 1. All vectors have unit length
# 2. Dot products between vectors are in {-1, -1/2, 0, 1/2, 1}

# Check lengths
unit_lengths = torch.allclose(
    torch.norm(normalized_vectors, dim=1),
    torch.ones(240, device=device),
    rtol=1e-5, atol=1e-5
)
assert unit_lengths, "E8 vectors must have unit length"

# Optional verification of dot products (commented out for performance)
# This is expensive for all pairs, but useful for testing
```

```
    """
    # Calculate all pairwise dot products
    dot_products = torch.mm(normalized_vectors, normalized_vectors.t())

    # Mask out diagonal (self-products)
    mask = ~torch.eye(240, dtype=torch.bool, device=device)
    off_diag_prods = dot_products[mask]

    # Get unique values (with tolerance)
    unique_approx = torch.unique(torch.round(off_diag_prods * 2) / 2)

    # Verify they match expected values
    expected = torch.tensor([-1.0, -0.5, 0.0, 0.5, 1.0], device=device)
    for val in unique_approx:
        assert min(torch.abs(val - expected)) < 1e-4, f"Unexpected dot product value: {val}"
    """

    return normalized_vectors
```

def construct_e8_routing_matrix(device: Optional[torch.device] = None) -> torch.Tensor: """ Construct the E8 lattice routing matrix for the 248-component.

```
Args:
    device: Computation device (CPU or CUDA)

Returns:
    Routing matrix based on E8 lattice
"""
# Set device if not provided
if device is None:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Get E8 lattice vectors
e8_vectors = construct_e8_lattice(device=device)

# Create rotation matrices: R2, R4, R8
R2 = torch.tensor([[np.cos(np.pi/2), -np.sin(np.pi/2)],
                   [np.sin(np.pi/2), np.cos(np.pi/2)]], dtype=torch.float32, device=device)

R4 = torch.zeros((4, 4), dtype=torch.float32, device=device)
for i in range(4):
    angle = 2 * np.pi * i / 4
    # Fill 2x2 blocks with rotation matrices
    R4[i//2*2:(i//2+1)*2, i%2*2:(i%2+1)*2] = torch.tensor(
        [[np.cos(angle), -np.sin(angle)],
         [np.sin(angle), np.cos(angle)]], dtype=torch.float32, device=device)

# Create 8x8 octonion matrix using E8 structure
R8 = torch.zeros((8, 8), dtype=torch.float32, device=device)
for i in range(8):
    for j in range(8):
        # Use Cayley table for octonions
        R8[i, j] = 1.0 if i == j else 0.0

# The full routing matrix would be extremely large (D×D),
# so we return the components instead
return {
    'e8_vectors': e8_vectors,
    'R2': R2,
    'R4': R4,
    'R8': R8
}
```

def quaternion_transform(tensor: torch.Tensor, angle: float, scale: float = 1.0, axis: Optional[torch.Tensor] = None, device: Optional[torch.device] = None) -> torch.Tensor: """ Apply quaternion rotation to tensor.

```
Args:
    tensor: Tensor to transform
    angle: Rotation angle in radians
    scale: Scaling factor
    axis: Rotation axis (default: z-axis)
    device: Computation device (CPU or CUDA)

Returns:
    Transformed tensor
"""
# Set device if not provided
if device is None:
    device = tensor.device

# Ensure tensor is on the correct device
if tensor.device != device:
    tensor = tensor.to(device)

# Handle rotation axis
if axis is None:
    # Default to z-axis
    axis = torch.tensor([0.0, 0.0, 1.0], device=device)
elif isinstance(axis, list):
    axis = torch.tensor(axis, dtype=torch.float32, device=device)

# Check for zero-length axis
axis_norm = torch.norm(axis)
if axis_norm < 1e-8:
    # Default to z-axis if input axis is degenerate
    axis = torch.tensor([0, 0, 1], dtype=torch.float32, device=device)
else:
    # Normalize axis vector
    axis = axis / axis_norm

# Create quaternion from axis-angle
half_angle = angle / 2.0
q_real = torch.cos(torch.tensor(half_angle, device=device))
q_imag = axis * torch.sin(torch.tensor(half_angle, device=device))

# Full quaternion
q = torch.cat([q_real.unsqueeze(0), q_imag], dim=0)

# Construct quaternion rotation matrix using direct formula
# This ensures perfect mathematical consistency
x, y, z = q_imag
w = q_real
```

```
R = torch.zeros((4, 4), dtype=torch.float32, device=device)

# Standard quaternion rotation matrix
R[0,0] = 1 - 2*y*y - 2*z*z
R[0,1] = 2*x*y - 2*w*z
R[0,2] = 2*x*z + 2*w*y
R[0,3] = 0

R[1,0] = 2*x*y + 2*w*z
R[1,1] = 1 - 2*x*x - 2*z*z
R[1,2] = 2*y*z - 2*w*x
R[1,3] = 0

R[2,0] = 2*x*z - 2*w*y
R[2,1] = 2*y*z + 2*w*x
R[2,2] = 1 - 2*x*x - 2*y*y
R[2,3] = 0

R[3,0] = 0
R[3,1] = 0
R[3,2] = 0
R[3,3] = 1

# Reshape tensor for quaternion interpretation
orig_shape = tensor.shape

# Document the quaternion interpretation
"""
Tensor reshaping for quaternion interpretation:
Every 4 consecutive values in the tensor are interpreted as a quaternion:
- index % 4 = 0: real part (w)
- index % 4 = 1: first imaginary component (x)
- index % 4 = 2: second imaginary component (y)
- index % 4 = 3: third imaginary component (z)
"""
if tensor.size(-1) % 4 != 0:
    # If tensor size not divisible by 4, pad with zeros
    pad_size = 4 - (tensor.size(-1) % 4)
    padded = torch.cat([tensor, torch.zeros(pad_size, dtype=tensor.dtype, device=device)])
    reshaped = padded.reshape(-1, 4)
else:
    reshaped = tensor.reshape(-1, 4)

# Apply rotation
rotated = torch.matmul(reshaped, R)
```

```
# Scale if needed
if scale != 1.0:
    rotated = rotated * scale

# Restore original shape (trimming padding if added)
if tensor.size(-1) % 4 != 0:
    rotated = rotated.reshape(-1)[:tensor.size(-1)]
    return rotated.reshape(orig_shape)
else:
    return rotated.reshape(orig_shape)
```

def quaternion_multiply(q1: torch.Tensor, q2: torch.Tensor, device: Optional[torch.device] = None) -> torch.Tensor: """ Multiply two quaternions.

```
Args:
    q1: First quaternion [w, x, y, z]
    q2: Second quaternion [w, x, y, z]
    device: Computation device (CPU or CUDA)

Returns:
    Quaternion product
"""
# Set device if not provided
if device is None:
    device = q1.device if isinstance(q1, torch.Tensor) else torch.device("cpu")

# Convert to tensor if needed
if not isinstance(q1, torch.Tensor):
    q1 = torch.tensor(q1, dtype=torch.float32, device=device)
if not isinstance(q2, torch.Tensor):
    q2 = torch.tensor(q2, dtype=torch.float32, device=device)

# Ensure quaternions are on the correct device
if q1.device != device:
    q1 = q1.to(device)
if q2.device != device:
    q2 = q2.to(device)

# Extract components
w1, x1, y1, z1 = q1
w2, x2, y2, z2 = q2

# Quaternion multiplication formula
w = w1*w2 - x1*x2 - y1*y2 - z1*z2
x = w1*x2 + x1*w2 + y1*z2 - z1*y2
y = w1*y2 - x1*z2 + y1*w2 + z1*x2
z = w1*z2 + x1*y2 - y1*x2 + z1*w2

return torch.tensor([w, x, y, z], dtype=torch.float32, device=device)
```

def quaternion_conjugate(q: torch.Tensor, device: Optional[torch.device] = None) -> torch.Tensor: """
Calculate quaternion conjugate.

```
Args:
    q: Quaternion [w, x, y, z]
    device: Computation device (CPU or CUDA)

Returns:
    Conjugate quaternion [w, -x, -y, -z]
"""
# Set device if not provided
if device is None:
    device = q.device if isinstance(q, torch.Tensor) else torch.device("cpu")

# Convert to tensor if needed
if not isinstance(q, torch.Tensor):
    q = torch.tensor(q, dtype=torch.float32, device=device)

# Ensure quaternion is on the correct device
if q.device != device:
    q = q.to(device)

# Return conjugate
return torch.tensor([q[0], -q[1], -q[2], -q[3]], dtype=torch.float32, device=device)
```

def octonion_transform(tensor: torch.Tensor, index: int, device: Optional[torch.device] = None) -> torch.Tensor: """ Apply octonion transformation based on E8 structure.

```
    Args:
        tensor: Tensor to transform
        index: Octonion basis index (0-7)
        device: Computation device (CPU or CUDA)

    Returns:
        Transformed tensor
    """
    # Set device if not provided
    if device is None:
        device = tensor.device

    # Ensure tensor is on the correct device
    if tensor.device != device:
        tensor = tensor.to(device)

    # This is a simplified version using basic octonion multiplication
    result = tensor.clone()

    # Reshape tensor into octonion form
    oct_shape = (tensor.shape[0] // 8, 8)
    if tensor.shape[0] % 8 != 0:
        # Pad tensor to make divisible by 8
        padding = 8 - (tensor.shape[0] % 8)
        padded = torch.cat([tensor, torch.zeros(padding, dtype=tensor.dtype, device=device)])
        reshaped = padded.reshape(oct_shape)
    else:
        reshaped = tensor.reshape(oct_shape)

    # Create octonion unit based on index
    e = torch.zeros(8, dtype=torch.float32, device=device)
    e[index] = 1.0

    # Apply octonion transformation
    for i in range(reshaped.shape[0]):
        # Extract octonion components
        o = reshaped[i].real  # Use real part for simplicity

        # Apply octonion multiplication (simplified)
        # In a full implementation, would use complete octonion multiplication table
        result_o = torch.zeros(8, dtype=torch.float32, device=device)

        # Basic octonion rotation - in practice would use Cayley table
        result_o[(index + torch.arange(8, device=device)) % 8] = o

        # Convert back to complex form
```

```
        reshaped[i] = torch.complex(result_o, torch.zeros_like(result_o))

    # Reshape back to original dimensions (trim padding if added)
    if tensor.shape[0] % 8 != 0:
        return reshaped.reshape(-1)[:tensor.shape[0]]
    else:
        return reshaped.reshape(tensor.shape)
```

def map_to_e8(field: torch.Tensor, e8_vectors: Optional[torch.Tensor] = None, device: Optional[torch.device] = None) -> torch.Tensor: """ Map field to E8 lattice coordinates.

```
Args:
    field: Field to map
    e8_vectors: Pre-computed E8 lattice vectors (optional)
    device: Computation device (CPU or CUDA)

Returns:
    E8 lattice coordinates
"""
# Set device if not provided
if device is None:
    device = field.device

# Ensure field is on the correct device
if field.device != device:
    field = field.to(device)

# Get E8 vectors if not provided
if e8_vectors is None:
    e8_vectors = construct_e8_lattice(device=device)

# Reshape field into 8-dimensional chunks
chunks = field.reshape(-1, 8)

# Initialize E8 coordinates
e8_coords = torch.zeros((chunks.shape[0], 8), dtype=torch.float32, device=device)

# For each chunk, project onto E8 basis
for i in range(chunks.shape[0]):
    # Extract real components for mapping
    chunk_real = chunks[i].real

    # Project onto E8 basis vectors
    projections = torch.matmul(e8_vectors, chunk_real)

    # Find top 8 projections
    _, top_indices = torch.topk(torch.abs(projections), k=8)

    # Use these as coordinates in E8 space
    for j, idx in enumerate(top_indices):
        e8_coords[i, j] = projections[idx]

# Flatten for further processing if needed
return e8_coords.reshape(-1)
```