""" DARKBOT™: Resonant Field Intelligence Architecture

Tensor operations module for the DARKBOT™ system.

This file contains tensor operation functions for field resonance, coherence, and transformations used in the DARKBOT™ architecture.

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

def calculate_resonance(field1: torch.Tensor, field2: torch.Tensor, epsilon: float = 1e-8, device: Optional[torch.device] = None) -> torch.Tensor: """ Calculate resonance between two field states.

```
The resonance function ρ(x,y) measures harmonic alignment between field states:

ρ(x,y) = <Φ(x),Φ(y)> / (||Φ(x)||·||Φ(y)||)

Args:
    field1: First field state
    field2: Second field state
    epsilon: Small constant for numerical stability
    device: Computation device (CPU or CUDA)

Returns:
    Resonance value in range [-1, 1]
"""
# Set device if not provided
if device is None:
    device = field1.device

# Ensure fields are on the same device
if field1.device != device:
    field1 = field1.to(device)
if field2.device != device:
    field2 = field2.to(device)

# Ensure fields are same size
min_size = min(field1.size(0), field2.size(0))
field1 = field1[:min_size]
field2 = field2[:min_size]

# Calculate inner product
inner_product = torch.sum(field1 * torch.conj(field2))

# Calculate magnitudes
mag1 = torch.sqrt(torch.sum(torch.abs(field1)**2)) + epsilon
mag2 = torch.sqrt(torch.sum(torch.abs(field2)**2)) + epsilon

# Calculate resonance
resonance = inner_product / (mag1 * mag2)

# Take absolute value for positive resonance measure
# Note: Some applications may want to keep the sign - modify as needed
return torch.abs(resonance)
```

def calculate_batch_resonance(fields1: torch.Tensor, fields2: torch.Tensor, epsilon: float = 1e-8, device: Optional[torch.device] = None) -> torch.Tensor: """ Calculate resonance between batches of field states.

```
Args:
    fields1: First batch of field states [B1, D]
    fields2: Second batch of field states [B2, D]
    epsilon: Small constant for numerical stability
    device: Computation device (CPU or CUDA)

Returns:
    Batch of resonance values [B1, B2]
"""
# Set device if not provided
if device is None:
    device = fields1.device

# Ensure fields are on the same device
if fields1.device != device:
    fields1 = fields1.to(device)
if fields2.device != device:
    fields2 = fields2.to(device)

# Get batch sizes and dimension
B1 = fields1.size(0)
B2 = fields2.size(0)

# Ensure fields are same dimension
min_dim = min(fields1.size(-1), fields2.size(-1))
fields1 = fields1[..., :min_dim]
fields2 = fields2[..., :min_dim]

# Reshape for broadcasting
fields1_expanded = fields1.unsqueeze(1)  # [B1, 1, D]
fields2_expanded = fields2.unsqueeze(0)  # [1, B2, D]

# Calculate batch inner products
inner_products = torch.sum(fields1_expanded * torch.conj(fields2_expanded), dim=-1)  # [B1,
B2]

# Calculate magnitudes
mag1 = torch.sqrt(torch.sum(torch.abs(fields1)**2, dim=-1)).unsqueeze(1) + epsilon  # [B1,
1]
mag2 = torch.sqrt(torch.sum(torch.abs(fields2)**2, dim=-1)).unsqueeze(0) + epsilon  # [1,
B2]

# Calculate resonance
resonance = inner_products / (mag1 * mag2)
```

```
# Take absolute value for positive resonance measure
return torch.abs(resonance)
```

def calculate_coherence(field: torch.Tensor, reference_states: List[torch.Tensor], weights: Optional[List[float]] = None, device: Optional[torch.device] = None) -> float: """ Calculate coherence of a field with reference states.

```python
The field coherence function χ(x) measures alignment with reference states:

χ(x) = Σ ρ(Φ(x),Φᵢ)·wᵢ / Σ wᵢ

Args:
    field: Field state to measure
    reference_states: List of reference field states
    weights: Importance weights for each reference (optional)
    device: Computation device (CPU or CUDA)

Returns:
    Coherence value in range [0, 1]
"""
# Set device if not provided
if device is None:
    device = field.device

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

# Set default weights if not provided
if weights is None:
    weights = [1.0] * len(reference_states)
else:
    assert len(weights) == len(reference_states), "Weights and reference states must have same length"

# Calculate resonance with each reference state
resonances = []
for ref_state in reference_states:
    # Ensure reference state is on the device
    if ref_state.device != device:
        ref_state = ref_state.to(device)

    # Calculate resonance
    res = calculate_resonance(field, ref_state, device=device)
    resonances.append(res.item())

# Calculate weighted average
total_weight = sum(weights)
if total_weight < 1e-8:  # Avoid division by zero
    return 0.0

coherence = sum(r * w for r, w in zip(resonances, weights)) / total_weight
```

```
    return coherence
```

def resonant_product(field1: torch.Tensor, field2: torch.Tensor, device: Optional[torch.device] = None) -> torch.Tensor: """ Implement the resonant product operator (⊛).

```
    A ⊛ B := Σᵢⱼ aᵢ·bⱼ·ρ(aᵢ,bⱼ)

    Args:
        field1: First field state
        field2: Second field state
        device: Computation device (CPU or CUDA)

    Returns:
        Result of resonant product operation
    """
    # Set device if not provided
    if device is None:
        device = field1.device

    # Ensure fields are on the same device
    if field1.device != device:
        field1 = field1.to(device)
    if field2.device != device:
        field2 = field2.to(device)

    # Ensure fields are compatible sizes
    if field1.size(0) != field2.size(0):
        min_size = min(field1.size(0), field2.size(0))
        field1 = field1[:min_size]
        field2 = field2[:min_size]

    # Calculate resonance
    resonance = calculate_resonance(field1, field2, device=device)

    # Apply resonant product
    result = field1 * field2 * resonance

    return result
```

def fractal_transform(field: torch.Tensor, level: int, device: Optional[torch.device] = None) -> torch.Tensor: """ Apply fractal transformation to a field state.

```
Args:
    field: Field state to transform
    level: Recursion level
    device: Computation device (CPU or CUDA)

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

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

# Create Hadamard-like transformation matrix
dim = field.size(0)
h_matrix = torch.ones((dim, dim), dtype=torch.complex64, device=device) /
torch.sqrt(torch.tensor(dim, device=device))

# Apply transformation
transformed = torch.matmul(h_matrix, field.unsqueeze(1)).squeeze()

# Apply scale factor based on level
gamma = 0.7  # Fractal scaling factor
scaled = transformed * (gamma ** level)

return scaled
```

def temporal_projection(field: torch.Tensor, delta_t: float, temporal_memory: List[torch.Tensor], device: Optional[torch.device] = None) -> torch.Tensor: """ Project field state forward or backward in time.

```
Args:
    field: Current field state
    delta_t: Time offset (positive for future, negative for past)
    temporal_memory: List of previous field states
    device: Computation device (CPU or CUDA)

Returns:
    Projected field state
"""
# Set device if not provided
if device is None:
    device = field.device

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

# Simple projection for empty memory
if not temporal_memory:
    # Apply phase rotation based on delta_t
    phase = torch.tensor(2.0 * np.pi * delta_t / 7.0, device=device)  # 157 -> 7
    projection = field * torch.exp(1j * phase)
    return projection

# Initialize projection with current field
projected = field.clone()

# Apply temporal weights based on memory
phi = (1.0 + torch.sqrt(torch.tensor(5.0, device=device))) / 2.0  # Golden ratio

# Determine weights based on delta_t
if delta_t > 0:  # Future projection
    # Use exponential weighting for future projection
    alpha = 0.1
    for n, past_field in enumerate(reversed(temporal_memory), 1):
        if past_field.device != device:
            past_field = past_field.to(device)

        # Apply temporal phase and decay
        weight = torch.exp(-alpha * n)
        phase = torch.tensor(2.0 * np.pi * delta_t * n / 7.0, device=device)

        projected += past_field * weight * torch.exp(1j * phase)

    # Normalize
    projected = projected / torch.sqrt(torch.sum(torch.abs(projected)**2))
```

```
else:  # Past reconstruction
    # Use resonance with memory for past reconstruction
    for n, past_field in enumerate(temporal_memory, 1):
        if past_field.device != device:
            past_field = past_field.to(device)

        # Calculate resonance with current field
        res = calculate_resonance(field, past_field, device=device)

        # Apply temporal weighting
        weight = res * (phi ** -n)
        projected = (1.0 - weight) * projected + weight * past_field

return projected
```

def phase_shift(field: torch.Tensor, phase: float, device: Optional[torch.device] = None) -> torch.Tensor: """ Apply phase shift to field state.

```
Args:
    field: Field state to transform
    phase: Phase angle in radians
    device: Computation device (CPU or CUDA)

Returns:
    Phase-shifted field state
"""
# Set device if not provided
if device is None:
    device = field.device

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

# Apply phase shift
phase_factor = torch.exp(torch.tensor(1j * phase, device=device))
shifted = field * phase_factor

return shifted
```

def hierarchical_resonant_product(field1: torch.Tensor, field2: torch.Tensor, k: int = 1024, device: Optional[torch.device] = None) -> torch.Tensor: """ Implement hierarchical approximation of resonant product for large fields.

```
Uses Nyström method for approximating the full resonance matrix.

Args:
    field1: First field state
    field2: Second field state
    k: Reduced rank size
    device: Computation device (CPU or CUDA)

Returns:
    Result of resonant product operation
"""
# Set device if not provided
if device is None:
    device = field1.device

# Ensure fields are on the same device
if field1.device != device:
    field1 = field1.to(device)
if field2.device != device:
    field2 = field2.to(device)

# Get dimensions
dim1 = field1.size(0)
dim2 = field2.size(0)

# For small dimensions, use standard resonant product
if dim1 <= k and dim2 <= k:
    return resonant_product(field1, field2, device=device)

# 1. Select smaller subset of points for approximation
k = min(k, min(dim1, dim2) // 4)  # Reduced rank size

# 2. Random sampling of indices
indices1 = torch.randperm(dim1, device=device)[:k]
indices2 = torch.randperm(dim2, device=device)[:k]

# 3. Extract submatrices
field1_sample = field1[indices1]
field2_sample = field2[indices2]

# 4. Compute resonance matrix for the sample
sample_res = calculate_batch_resonance(field1_sample, field2_sample, device=device)

# 5. Compute weights for original field points
batch_size = 64  # Process in batches for memory efficiency
```

```
weights1 = torch.zeros((dim1, k), device=device)
for i in range(0, dim1, batch_size):
    end_i = min(i + batch_size, dim1)
    f1_batch = field1[i:end_i].unsqueeze(1)
    f1_sample_batch = field1_sample.unsqueeze(0)

    res = calculate_batch_resonance(f1_batch, f1_sample_batch, device=device)
    weights1[i:end_i, :] = res

weights2 = torch.zeros((dim2, k), device=device)
for i in range(0, dim2, batch_size):
    end_i = min(i + batch_size, dim2)
    f2_batch = field2[i:end_i].unsqueeze(1)
    f2_sample_batch = field2_sample.unsqueeze(0)

    res = calculate_batch_resonance(f2_batch, f2_sample_batch, device=device)
    weights2[i:end_i, :] = res

# 6. Compute approximation of full resonance matrix
# This is a low-rank approximation: R ≈ W1 * S * W2^T

# Apply to field2 first
weighted_field2 = torch.matmul(weights2, sample_res.T)  # [dim2, k] * [k, dim1] = [dim2,
dim1]

# Then apply to field1
result = torch.matmul(field1, weighted_field2.T)  # [dim1] * [dim1, dim2] = [dim2]

return result
```