""" DARKBOT™ Test Suite: Core functionality tests

Unit tests for the core functionality of the DARKBOT™ system.

© 2025 Cato Johansen // DARKBOT™ // Artifact №369.157.248 """

import pytest import torch import numpy as np from darkbot import DarkBot, DarkBotConfig from darkbot.tensor_ops import calculate_resonance, calculate_coherence from darkbot.lattice import construct_e8_lattice

Fixtures for test setup

@pytest.fixture def darkbot(): """Create DarkBot instance with deterministic settings""" # Use small dimensions for faster tests config = DarkBotConfig() config.field.dimensions = 64

```
db = DarkBot(config)
return db
```

@pytest.fixture def test_fields(darkbot): """Generate test field states""" # Set specific seed for test determinism torch.manual_seed(369)

```
# Generate test fields
field1 = torch.randn(64, dtype=torch.complex64, device=darkbot.device)
field2 = torch.randn(64, dtype=torch.complex64, device=darkbot.device)

# Also create edge case fields
zero_field = torch.zeros(64, dtype=torch.complex64, device=darkbot.device)

# Create near-identical fields
similar_field = field1 * 0.99 + torch.randn(64, dtype=torch.complex64,
device=darkbot.device) * 0.01

return {
    "field1": field1,
    "field2": field2,
    "zero_field": zero_field,
    "similar_field": similar_field
}
```

Core functionality tests

def test_e8_lattice_construction(darkbot): """Test E8 lattice construction produces 240 vectors""" e8 = construct_e8_lattice(device=darkbot.device)

```
    # Verify correct number of vectors
    assert e8.shape[0] == 240, "E8 lattice should have 240 vectors"

    # Verify all vectors have correct norm
    norms = torch.norm(e8, dim=1)
    assert torch.allclose(norms, torch.ones_like(norms), rtol=1e-5, atol=1e-5), \
        "E8 vectors should have unit norm"

    # Verify E8 properties (root system)
    # For any two distinct vectors v, w in E8, their dot product <v,w> ∈ {-1, -1/2, 0, 1/2, 1}
    dot_products = torch.matmul(e8, e8.T)

    # Get unique dot products, excluding diagonal
    mask = ~torch.eye(240, dtype=torch.bool, device=darkbot.device)
    unique_dots = torch.unique(torch.round(dot_products[mask] * 2) / 2)

    # Check dot products are in the expected set
    expected_dots = torch.tensor([-1.0, -0.5, 0.0, 0.5, 1.0], device=darkbot.device)

    # Allow for floating point error
    for dot in unique_dots:
        assert min(torch.abs(dot - expected_dots)) < 1e-4, \
            f"Found unexpected dot product {dot}"
```

def test_resonance_properties(darkbot, test_fields): """Test resonance function mathematical properties""" # Test self-resonance is 1.0 res_self = calculate_resonance(test_fields["field1"], test_fields["field1"], device=darkbot.device) assert torch.abs(res_self - 1.0) < 1e-5, "Self-resonance should be 1.0"

```
    # Test symmetry: ρ(x,y) = ρ(y,x)
    res_12 = calculate_resonance(test_fields["field1"], test_fields["field2"],
    device=darkbot.device)
    res_21 = calculate_resonance(test_fields["field2"], test_fields["field1"],
    device=darkbot.device)
    assert torch.abs(res_12 - res_21) < 1e-5, "Resonance should be symmetric"

    # Test bound: -1 ≤ ρ(x,y) ≤ 1
    assert -1.0 <= res_12 <= 1.0, "Resonance should be bounded [-1, 1]"

    # Test near-identical fields have high resonance
    res_similar = calculate_resonance(test_fields["field1"], test_fields["similar_field"],
    device=darkbot.device)
    assert res_similar > 0.95, "Similar fields should have high resonance"

    # Test edge case: zero field
    res_zero = calculate_resonance(test_fields["field1"], test_fields["zero_field"],
    device=darkbot.device)
    assert not torch.isnan(res_zero), "Resonance with zero field should not be NaN"
    assert not torch.isinf(res_zero), "Resonance with zero field should not be Inf"
```

def test_one_draw_search(darkbot): """Test One Draw search finds correct maximum resonance match""" # Create target space with known target target_space = [torch.randn(64, dtype=torch.complex64, device=darkbot.device) for _ in range(5)]

```
    # Make a query that's very similar to one target
    target_idx = 2
    query = target_space[target_idx] * 0.9 + torch.randn(64, dtype=torch.complex64,
    device=darkbot.device) * 0.1

    # Perform search
    result = darkbot.one_draw_search(query, target_space)

    # Verify correct target found
    assert result['match_index'] == target_idx, "One Draw should find most resonant match"

    # Test with exact match
    exact_query = target_space[1].clone()
    exact_result = darkbot.one_draw_search(exact_query, target_space)
    assert exact_result['match_index'] == 1, "One Draw should find exact match"
    assert exact_result['confidence'] > 0.99, "Confidence for exact match should be high"
```

def test_quantum_processing(darkbot): """Test quantum processing cycle""" # Create input data input_data = torch.randn(64, dtype=torch.complex64, device=darkbot.device)

```
    # Process through the quantum system
    result = darkbot._process_quantum_core(input_data)

    # Check result properties
    assert result.shape == input_data.shape, "Output shape should match input"
    assert torch.is_complex(result), "Output should be complex-valued"

    # Check field normalization
    norm = torch.sqrt(torch.sum(torch.abs(result)**2))
    assert torch.abs(norm - 1.0) < 1e-5, "Field should be normalized to unit magnitude"

    # Process again and check determinism
    result2 = darkbot._process_quantum_core(input_data)
    assert torch.allclose(result, result2, rtol=1e-5, atol=1e-5), \
        "Processing should be deterministic for same input"
```

def test_batch_processing(darkbot): """Test batch processing functionality""" batch_size = 4 dim = 64

```
    # Create batch input
    batch_input = torch.randn(batch_size, dim, dtype=torch.complex64, device=darkbot.device)

    # Process batch
    batch_output = darkbot._process_batch_core(batch_input)

    # Check output shape
    assert batch_output.shape[0] == batch_size, "Output batch size should match input"
    assert batch_output.shape[1] == dim, "Output dimensions should match input"

    # Test each item individually and compare
    individual_outputs = [
        darkbot._process_quantum_core(batch_input[i])
        for i in range(batch_size)
    ]

    # Stack for comparison
    individual_stacked = torch.stack(individual_outputs)

    # Verify deterministic behavior
    for i in range(batch_size):
        assert torch.allclose(batch_output[i], individual_stacked[i], rtol=1e-5, atol=1e-5), \
            f"Batch processing item {i} differs from individual processing"
```

def test_configuration(darkbot): """Test configuration system""" config = darkbot.config

```
    # Check default values
    assert config.field.dimensions == 64, "Field dimensions should be 64 for testing"
    assert config.resonance.gamma == 0.7, "Default gamma should be 0.7"
    assert config.resonance.chi_high == 0.85, "Default chi_high should be 0.85"
    assert config.resonance.chi_low == 0.65, "Default chi_low should be 0.65"

    # Check numerology
    assert config.numerology.NUM_369 == (3, 6, 9), "Numerology 369 should be (3, 6, 9)"
    assert config.numerology.NUM_157 == (1, 5, 7), "Numerology 157 should be (1, 5, 7)"
    assert config.numerology.NUM_248 == (2, 4, 8), "Numerology 248 should be (2, 4, 8)"

    # Check weights sum to 1.0
    assert abs(sum(config.numerology.WEIGHTS_369) - 1.0) < 1e-8, "Weights 369 should sum to 1.0"
    assert abs(sum(config.numerology.WEIGHTS_157) - 1.0) < 1e-8, "Weights 157 should sum to 1.0"
    assert abs(sum(config.numerology.WEIGHTS_248) - 1.0) < 1e-8, "Weights 248 should sum to 1.0"
```

def test_config_validation(): """Test configuration validation""" # Test invalid field dimensions with pytest.raises(ValueError): config = DarkBotConfig() config.field.dimensions = 32 # Below minimum

```
    with pytest.raises(ValueError):
        config = DarkBotConfig()
        config.field.dimensions = 8192  # Above maximum

    # Test invalid coherence thresholds
    with pytest.raises(ValueError):
        config = DarkBotConfig()
        config.resonance.chi_high = 0.6  # Below minimum

    with pytest.raises(ValueError):
        config = DarkBotConfig()
        config.resonance.chi_low = 0.4  # Below minimum

    # Test threshold inversion
    with pytest.raises(ValueError):
        config = DarkBotConfig()
        config.resonance.chi_high = 0.7
        config.resonance.chi_low = 0.7  # Equal to chi_high
```