2025-11-01

## section 0

[2em] **Part Overview · Duration:**

***Beginner-Friendly Visual Study Guide***

### subsection 0.0 What You'll Learn

- **Coverage Tiers**: 85% overall, 95% critical, 100% safety-critical
- **Test Organization**: Peer files (test_*.py for every *.py)
- **Validation Strategies**: Theoretical properties, edge cases, integration tests
- **Automation**: Pre-commit hooks, CI/CD integration, benchmark regression

### subsection 0.0 Why This Matters

**Problem**: Uncaught edge cases (e.g., singular matrices, division by zero) cause silent failures in production.

**Solution**: Enforce 95% coverage on critical paths (controllers, dynamics) with theoretical property validation (Lyapunov stability, boundedness).

**Impact**: Zero critical bugs reported in Phase 5 research (11 tasks, 46 hours) after achieving coverage standards.

## section 0 Coverage Tiers

### subsection 0.0 Three-Tier System

| Tier | Coverage Target | Examples |
|---|---|---|
| Safety-Critical | 100% | SMC control law, saturation |
| Critical Paths | $\geq 95\%$ | Controllers, dynamics, PSO |
| Overall Codebase | $\geq 85\%$ | Utils, visualization, CLI |

### subsection 0.0 Rationale

- **100% Safety-Critical**: Code that can cause physical damage (control saturation, state constraints)
- **95% Critical Paths**: Core algorithms (SMC variants, PSO optimizer, dynamics models)
- **85% Overall**: Acceptable trade-off (diminishing returns above 85% for utilities)

### subsection 0.0 Current Status (January 2025)

- **Overall Coverage**: 87% (target: 85%, status: PASS)
- **Critical Paths**: 96% controllers, 94% dynamics (target: 95%, status: PASS)
- **Safety-Critical**: 100% (saturation, state validators, status: PASS)
- **Test Count**: 200+ pytest tests, 15 benchmark tests

## section 0 Test Organization

### subsection 0.0 Peer File Structure

**Rule**: Every `*.py` file has a corresponding `test_*.py` peer.

```
lstnumbersrc/
lstnumber  controllers/
lstnumber    classical_smc.py
lstnumber    sta_smc.py
lstnumber    adaptive_smc.py
lstnumber  core/
lstnumber    simulation_runner.py
lstnumber  optimizer/
lstnumber    pso_optimizer.py
lstnumber
lstnumbertests/
lstnumber  test_controllers/
lstnumber    test_classical_smc.py        # Peer for classical_smc.py
lstnumber    test_sta_smc.py              # Peer for sta_smc.py
```

```
    test_adaptive_smc.py          # Peer for adaptive_smc.py
  test_core/
    test_simulation_runner.py     # Peer for simulation_runner.py
  test_optimizer/
    test_pso_optimizer.py         # Peer for pso_optimizer.py
```

### subsection 0.0 Test Discovery

```
# Run All Tests
python -m pytest tests/ -v

# Run Specific Module
python -m pytest tests/test_controllers/ -v

# Run Single File
python -m pytest tests/test_controllers/test_classical_smc.py -v

# Run with Coverage Report
python -m pytest tests/ --cov=src --cov-report=html
```

### subsection 0.0 Peer File Validation

```
# Check for Missing Peer Files
python scripts/architecture/find_untested.py

# Output Example
[WARNING] Missing test peers:
  src/utils/plotting/advanced_plots.py
  src/core/batch_simulator.py

[INFO] Action: Create tests/test_utils/test_plotting/test_advanced_plots.py
[INFO] Action: Create tests/test_core/test_batch_simulator.py
```

## section 0 Validation Strategies

### subsection 0.0 1. Unit Tests (Isolated Components)

**Goal**: Verify single function/class behavior in isolation.

```python
def test_classical_smc_zero_error():
    """Test SMC returns zero control when error is zero."""
    controller = ClassicalSMC(lambda1=10, lambda2=5, phi1=2, phi2=1)
    state = np.array([0, 0, 0, 0])  # Zero error
    control = controller.compute_control(state)
    assert np.allclose(control, 0.0, atol=1e-6)

def test_pso_converges():
    """Test PSO reduces cost over iterations."""
    optimizer = PSOTuner(swarm_size=10, iterations=20)
    initial_cost = optimizer.best_cost
    optimizer.optimize(objective_function)
    final_cost = optimizer.best_cost
    assert final_cost < initial_cost  # Cost must decrease
```

### subsection 0.0 2. Integration Tests (Multi-Component)

**Goal**: Verify components work together correctly.

```python
def test_simulation_end_to_end():
    """Test full simulation pipeline (controller + dynamics)."""
    controller = create_controller("classical_smc", config)
    dynamics = SimplifiedDynamics(config)
    runner = SimulationRunner(controller, dynamics, config)

    result = runner.run()

    # Verify convergence
    assert result.settling_time < 5.0  # Settles in <5s
    assert np.max(np.abs(result.states[-1])) < 0.1  # Final error <0.1

def test_pso_tunes_controller():
    """Test PSO optimizer improves controller performance."""
    baseline_iae = simulate_with_manual_gains()
    pso = PSOTuner(swarm_size=30, iterations=50)
    optimized_gains = pso.optimize(controller_objective)

    optimized_iae = simulate_with_gains(optimized_gains)
    assert optimized_iae < baseline_iae  # PSO improves performance
```

### subsection 0.0 3. Property-Based Tests (Theoretical Validation)

**Goal**: Verify mathematical properties hold for ALL inputs.

```
from hypothesis import given, strategies as st

@given(st.floats(min_value=-10, max_value=10, allow_nan=False))
def test_saturation_bounds(control_input):
    """Test saturation enforces bounds for ALL inputs."""
    saturated = saturate(control_input, u_max=5.0)
    assert -5.0 <= saturated <= 5.0  # Always bounded

@given(st.lists(st.floats(), min_size=4, max_size=4))
def test_lyapunov_decreasing(state):
    """Test Lyapunov function decreases for ALL states."""
    V_current = compute_lyapunov(state)
    state_next = simulate_step(state)
    V_next = compute_lyapunov(state_next)
    assert V_next < V_current  # Lyapunov always decreases
```

### subsection 0.0 4. Edge Case Tests

**Goal**: Verify behavior at boundaries and degenerate cases.

```
def test_controller_singular_matrix():
    """Test controller handles singular inertia matrix."""
    dynamics = SimplifiedDynamics(inertia=np.zeros((4,4)))  # Singular
    with pytest.raises(SingularMatrixError):
        simulate(dynamics)

def test_pso_empty_search_space():
    """Test PSO handles zero-width search bounds."""
    bounds = [(1.0, 1.0), (2.0, 2.0)]  # Zero width
    with pytest.raises(ValueError, match="Search␣space␣has␣zero␣volume"):
        PSOTuner(bounds=bounds)

def test_controller_inf_nan_inputs():
    """Test controller rejects inf/nan inputs gracefully."""
    controller = ClassicalSMC(lambda1=10, lambda2=5)
    state = np.array([np.inf, np.nan, 0, 0])
    with pytest.raises(ValueError, match="State␣contains␣inf/nan"):
        controller.compute_control(state)
```

## section 0 Benchmark Tests

### subsection 0.0 Purpose

- **Performance Regression Detection**: Alert if simulation slows by >5%
- **Comparative Analysis**: Benchmark 7 controllers head-to-head
- **Optimization Validation**: Verify PSO reduces IAE by 40%+

### subsection 0.0 Benchmark Organization

```
tests/test_benchmarks/
  test_controller_benchmarks.py   # Time per step, IAE, chattering
  test_pso_benchmarks.py           # Convergence speed, final cost
  test_simulation_benchmarks.py    # End-to-end runtime
```

### subsection 0.0 Example Benchmark Test

```python
import pytest

@pytest.mark.benchmark(group="controllers")
def test_classical_smc_benchmark(benchmark):
    """Benchmark classical SMC compute_control() speed."""
    controller = ClassicalSMC(lambda1=10, lambda2=5, phi1=2, phi2=1)
    state = np.array([0.1, 0.2, 0.0, 0.0])

    result = benchmark(controller.compute_control, state)

    # Verify performance target (10ms max)
    assert result.stats.mean < 0.01  # <10ms average

@pytest.mark.benchmark(group="optimization")
def test_pso_benchmark(benchmark):
    """Benchmark PSO convergence time."""
    pso = PSOTuner(swarm_size=30, iterations=50)

    result = benchmark(pso.optimize, rosenbrock_function)

    # Verify convergence time (<2 seconds)
    assert result.stats.mean < 2.0
```

### subsection 0.0 Running Benchmarks

```bash
# Run All Benchmarks
python -m pytest tests/test_benchmarks/ --benchmark-only

# Compare Against Baseline
python -m pytest tests/test_benchmarks/ --benchmark-compare=baseline.json

# Save New Baseline
python -m pytest tests/test_benchmarks/ --benchmark-save=new_baseline
```

## section 0 Pre-commit Hooks

### subsection 0.0 Automated Test Enforcement

**File**: `.pre-commit-config.yaml`

```yaml
repos:
  - repo: local
    hooks:
      # Run Pytest on Changed Files
      - id: pytest
        name: Run Tests
        entry: python -m pytest tests/ -v
        language: system
        pass_filenames: false
        stages: [commit]

      # Check Coverage Thresholds
      - id: coverage
        name: Check Coverage
        entry: python -m pytest --cov=src --cov-report=term --cov-fail-under=85
        language: system
        pass_filenames: false
        stages: [commit]

      # Find Untested Files
      - id: untested
        name: Check for Untested Files
        entry: python scripts/architecture/find_untested.py
        language: system
        stages: [commit]
```

### subsection 0.0 Commit Workflow

```
# Attempt Commit (triggers pre-commit hooks)
git commit -m "feat: Add adaptive SMC controller"

# Pre-commit Hook Output
[INFO] Running pytest tests/
==================== 200 passed in 12.3s ====================

[INFO] Checking coverage thresholds
Coverage: 87% (target: 85%) PASS

[INFO] Checking for untested files
All files have test peers. PASS

[OK] Pre-commit checks passed. Commit created.
```

## section 0 CI/CD Integration

### subsection 0.0 GitHub Actions Workflow

**File**: `.github/workflows/tests.yml`

```
name: Test Suite
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install Dependencies
        run: pip install -r requirements.txt

      - name: Run Tests
        run: python -m pytest tests/ -v --cov=src --cov-report=xml

      - name: Upload Coverage
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml
          fail_ci_if_error: true

      - name: Check Coverage Thresholds
        run: |
          python -m pytest --cov=src --cov-fail-under=85
          python scripts/quality/check_critical_coverage.py  # Verify 95% critical

      - name: Run Benchmarks
        run: python -m pytest tests/test_benchmarks/ --benchmark-only
```

### subsection 0.0 Pull Request Checks

- **Test Pass Rate**: 100% required (all 200+ tests must pass)
- **Coverage Thresholds**: Overall $\geq$ 85%, critical $\geq$ 95%
- **Benchmark Regression**: No slowdowns >5% vs. baseline
- **Linting**: Ruff + MyPy strict mode, zero errors

## section 0 Testing Anti-Patterns

### subsection 0.0 Anti-Pattern 1: Trivial Tests

**Problem**: Tests that only verify imports or type checks.

```
# BAD: Trivial test (no value)
def test_controller_imports():
    from src.controllers.classical_smc import ClassicalSMC
    assert ClassicalSMC is not None  # Useless assertion

# GOOD: Functional test
def test_controller_convergence():
    controller = ClassicalSMC(...)
    result = simulate(controller, initial_state=[1, 0, 0, 0])
    assert result.settling_time < 5.0  # Meaningful property
```

### subsection 0.0 Anti-Pattern 2: Non-Deterministic Tests

**Problem**: Tests that fail randomly due to uncontrolled randomness.

```
# BAD: Random seed not fixed
def test_pso_converges():
    pso = PSOTuner()  # No seed
    result = pso.optimize(objective)
    assert result.cost < 0.1  # Fails 10% of time

# GOOD: Fixed seed for reproducibility
def test_pso_converges():
    pso = PSOTuner(seed=42)  # Deterministic
    result = pso.optimize(objective)
    assert result.cost < 0.1  # Always passes
```

### subsection 0.0 Anti-Pattern 3: Monolithic Tests

**Problem**: Single test verifies 10+ behaviors (hard to debug).

```
# BAD: Test does too much
def test_entire_system():
    # 50 lines testing controller + dynamics + PSO + visualization
    assert everything_works  # Unclear what failed

# GOOD: Atomic tests
def test_controller_compute():
    # Test only controller.compute_control()

def test_dynamics_step():
    # Test only dynamics.step()

def test_pso_optimize():
    # Test only pso.optimize()
```

## section 0 Coverage Measurement

### subsection 0.0 Line Coverage

**Tool**: `pytest-cov`

```
# Generate HTML Report
python -m pytest --cov=src --cov-report=html

# View Report
open htmlcov/index.html  # (Mac/Linux)
start htmlcov/index.html # (Windows)
```

### subsection 0.0 Branch Coverage

**Goal**: Verify all conditional branches (if/else) tested.

```
lstnumber# Enable Branch Coverage
lstnumberpython -m pytest --cov=src --cov-branch --cov-report=term
lstnumber
lstnumber# Example Output
lstnumbersrc/controllers/classical_smc.py   96%   (4 branches, 3 covered)
lstnumbersrc/optimizer/pso_optimizer.py     94%   (12 branches, 11 covered)
```

### subsection 0.0 Critical Path Coverage

**Script**: `scripts/quality/check_critical_coverage.py`

```
lstnumberCRITICAL_PATHS = [
lstnumber    "src/controllers/",
lstnumber    "src/core/dynamics.py",
lstnumber    "src/optimizer/pso_optimizer.py",
lstnumber    "src/utils/control_primitives.py"
lstnumber]
lstnumber
lstnumberdef check_critical_coverage():
lstnumber    coverage_data = parse_coverage_report()
lstnumber    for path in CRITICAL_PATHS:
lstnumber        coverage = coverage_data[path]
lstnumber        if coverage < 0.95:  # 95% threshold
lstnumber            raise Exception(f"Critical␣path␣{path}␣only␣{coverage*100}%␣covered")
lstnumber    print("[OK]␣All␣critical␣paths␣meet␣95%␣threshold")
```

## section 0 Maintenance & Continuous Improvement

### subsection 0.0 Monthly Test Audit

```
lstnumber# 1. Find Untested Code
lstnumberpython scripts/architecture/find_untested.py
lstnumber
lstnumber# 2. Identify Low-Coverage Modules
lstnumberpython -m pytest --cov=src --cov-report=term | grep -E '[0-7][0-9]%'
lstnumber
lstnumber# 3. Review Flaky Tests (tests that fail intermittently)
lstnumberpython scripts/quality/detect_flaky_tests.py
lstnumber
lstnumber# 4. Update Benchmark Baselines
lstnumberpython -m pytest tests/test_benchmarks/ --benchmark-save=baseline_jan2025
```

### subsection 0.0 Regression Analysis

- **Track Test Count Growth**: Target 10+ new tests per major feature
- **Monitor Coverage Trends**: Alert if coverage drops below 85%
- **Benchmark Drift**: Flag if any benchmark slows by >5%
- **Flaky Test Detection**: Remove non-deterministic tests

### subsection 0.0 Test Documentation

**File**: `tests/README.md`

- Document test organization (peer files, benchmarks, integration)
- List coverage targets (85%/95%/100% tiers)
- Explain how to run specific test suites
- Provide examples of property-based tests
- Link to CI/CD workflow and pre-commit hooks

## section 0 Success Metrics

### subsection 0.0 Quantitative Indicators

| Metric | Target | Current |
|---|---|---|
| Overall coverage | $\geq 85\%$ | 87% |
| Critical coverage | $\geq 95\%$ | 96% (controllers) |
| Safety-critical coverage | 100% | 100% (saturation) |
| Test count | $> 200$ | 215 |
| Benchmark tests | $> 15$ | 18 |
| Flaky test rate | $< 2\%$ | 0.5% |

### subsection 0.0 Qualitative Indicators

- Zero critical bugs reported in Phase 5 research (11 tasks, 46 hours)
- All edge cases discovered via property-based tests (Hypothesis)
- Pre-commit hooks prevent untested code from merging
- CI/CD catches regressions before production deployment

## Checklist: Testing Philosophy

- ☐ **Organize**: Create peer test files (test_*.py for every *.py)
- ☐ **Coverage**: Achieve 85% overall, 95% critical, 100% safety-critical
- ☐ **Validate**: Write property-based tests for theoretical invariants
- ☐ **Benchmark**: Track performance regressions (<5% tolerance)
- ☐ **Pre-commit**: Enable hooks to block untested code
- ☐ **CI/CD**: Integrate GitHub Actions for PR checks
- ☐ **Edge Cases**: Test singular matrices, inf/nan, zero-width bounds
- ☐ **Monitor**: Monthly audits for flaky tests and coverage drift

## Next Steps

- **E019**: Production safety - memory management and thread safety
- **E020**: MCP integration - auto-trigger strategy and server orchestration
- **E021**: Maintenance mode and future vision for DIP-SMC-PSO