2025-11-01

section 0

[2em] Part Overview · Duration:

Beginner-Friendly Visual Study Guide

subsection 0.0 What You'll Learn

- **Memory Management**: Weakref patterns, circular reference prevention
- **Thread Safety**: Validated via 11/11 passing tests
- **Production Readiness Score**: 23.9/100 (Phase 4.1+4.2 complete)
- **Status**: Research-ready, NOT production-ready

subsection 0.0 Why This Matters

Problem: Controller state accumulates over long simulations (10K+ steps), causing memory leaks and eventual OOM crashes.

Solution: All controllers use weakref patterns + explicit cleanup() methods, validated via 200+ pytest tests.

Impact: Memory footprint stable over 100K simulation steps (tested), no circular references detected.

section 0 Memory Management Architecture

subsection 0.0 The Circular Reference Problem

Scenario: Controller holds reference to Dynamics, Dynamics holds reference to Controller.

```
# BAD: Circular reference (memory leak)
class Controller:
    def __init__(self, dynamics):
        self.dynamics = dynamics  # Strong reference

class Dynamics:
    def __init__(self, controller):
        self.controller = controller  # Strong reference

# Neither object can be garbage collected!
```

subsection 0.0 Weakref Solution

Fix: Use `weakref.ref()` for back-references.

```
import weakref

class Controller:
    def __init__(self, dynamics):
        self._dynamics_ref = weakref.ref(dynamics)  # Weak reference

    def get_dynamics(self):
        dynamics = self._dynamics_ref()
        if dynamics is None:
            raise RuntimeError("Dynamics␣object␣was␣garbage␣collected")
        return dynamics

class Dynamics:
    def __init__(self, controller):
        self.controller = controller  # Strong reference OK
```

subsection 0.0 Controller Memory Patterns

All controllers follow this pattern:

```
class ClassicalSMC(BaseController):
    def __init__(self, lambda1, lambda2, phi1, phi2):
        super().__init__()
        self.gains = [lambda1, lambda2, phi1, phi2]
        self._state_history = []  # Could grow unbounded

    def compute_control(self, state):
        self._state_history.append(state)  # Memory accumulation
        # Limit history size
        if len(self._state_history) > MAX_HISTORY:
```

```python
            self._state_history.pop(0)
        return self._compute_smc(state)

    def cleanup(self):
        """Explicit cleanup for long-running simulations."""
        self._state_history.clear()
        super().cleanup()
```

section 0 Memory Leak Prevention

subsection 0.0 Common Leak Sources

- **1. Unbounded Histories**: Controller stores ALL past states (10K+ arrays)
- **2. Circular References**: Controller ↔ Dynamics back-refs
- **3. Event Listeners**: Callbacks hold references to large objects
- **4. Cache Bloat**: Memoization caches grow unbounded

subsection 0.0 Mitigation Strategies

Leak Source	**Mitigation**
Unbounded histories	Limit to last N entries (e.g., 1000 states)
Circular refs	Use weakref for back-references
Event listeners	Explicitly unsubscribe in cleanup()
Cache bloat	Use LRU cache with max size

subsection 0.0 Memory Monitoring

```python
import tracemalloc
import gc

def monitor_memory(controller, simulation_steps=10000):
    """Track memory growth during simulation."""
    tracemalloc.start()

    for i in range(simulation_steps):
        state = get_current_state()
        controller.compute_control(state)

        if i % 1000 == 0:  # Check every 1000 steps
            current, peak = tracemalloc.get_traced_memory()
            print(f"Step␣{i}:␣Current={current/1e6:.2f}␣MB,␣Peak={peak/1e6:.2f}␣MB")

    tracemalloc.stop()

    # Force garbage collection
    gc.collect()
    unreachable = gc.collect()
    if unreachable > 0:
        print(f"[WARNING]␣{unreachable}␣unreachable␣objects␣(possible␣leak)")
```

section 0 Thread Safety

subsection 0.0 Current Status

- **Validation**: 11/11 thread safety tests passing
- **Scope**: Single-threaded and multi-threaded operation validated
- **Concurrency Model**: Controllers are NOT thread-safe by default

- **Recommendation**: Use separate controller instances per thread

subsection 0.0 Thread-Safe Controller Pattern

```python
import threading

class ThreadSafeController:
    def __init__(self, base_controller_class, **kwargs):
        self.lock = threading.Lock()
        self.controller = base_controller_class(**kwargs)

    def compute_control(self, state):
        with self.lock:  # Ensure exclusive access
            return self.controller.compute_control(state)

# Usage
safe_controller = ThreadSafeController(ClassicalSMC, lambda1=10, lambda2=5)

# Safe from multiple threads
def worker(state):
    control = safe_controller.compute_control(state)
    print(f"Control:␣{control}")

threads = [threading.Thread(target=worker, args=(state,)) for _ in range(10)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

subsection 0.0 Thread Safety Tests

File: `tests/test_integration/test_memory_management/test_thread_safety.py`

```python
def test_concurrent_controller_access():
    """Test multiple threads accessing controller simultaneously."""
    controller = ClassicalSMC(lambda1=10, lambda2=5)
    results = []

    def compute_many_times():
        for _ in range(100):
            state = np.random.rand(4)
            control = controller.compute_control(state)
            results.append(control)

    threads = [threading.Thread(target=compute_many_times) for _ in range(10)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()

    # Verify no exceptions raised, all results valid
    assert len(results) == 1000  # 10 threads * 100 calls each
    assert all(isinstance(r, (float, np.ndarray)) for r in results)
```

section 0 Production Readiness Scoring

subsection 0.0 The 8 Quality Gates

Gate	Status	Weight
Test Coverage	87% (target 85%)	20%
Critical Issues	0 found	20%
Memory Safety	11/11 tests pass	15%
Documentation	98% API coverage	10%
Linting	8.7/10 (target 9.0)	10%
Type Safety	MyPy strict pass	10%
Performance	Within 3% baseline	10%
MCP Integration	11/11 servers	5%
Total	**7/8 passing**	**-**

subsection 0.0 Production Readiness Score: 23.9/100

- **Phase 4.1 Complete**: Thread safety validation (11/11 tests)
- **Phase 4.2 Complete**: Memory management patterns implemented
- **Remaining Work**: Quality gate automation, coverage measurement fixes
- **Status**: RESEARCH-READY, NOT PRODUCTION-READY

subsection 0.0 Research-Ready vs. Production-Ready

Research-Ready (CURRENT)	Production-Ready (FUTURE)
87% test coverage	95% coverage critical paths
11/11 thread safety tests	100% thread-safe controllers
Manual memory monitoring	Automated leak detection
7/8 quality gates passing	8/8 gates enforced in CI/CD
Single-threaded primary use	Multi-threaded production load

section 0 Cleanup Protocols

subsection 0.0 Explicit Cleanup Pattern

```
lstnumberdef run_long_simulation(controller, dynamics, steps=100000):
lstnumber    """Run simulation with periodic cleanup."""
lstnumber    for i in range(steps):
lstnumber        state = dynamics.get_state()
lstnumber        control = controller.compute_control(state)
lstnumber        dynamics.step(control)
lstnumber
lstnumber        # Periodic cleanup every 10K steps
lstnumber        if i % 10000 == 0:
lstnumber            controller.cleanup()
lstnumber            dynamics.cleanup()
lstnumber            gc.collect()  # Force garbage collection
lstnumber
lstnumber    # Final cleanup
lstnumber    controller.cleanup()
lstnumber    dynamics.cleanup()
```

subsection 0.0 Context Manager Pattern

```
lstnumberclass ManagedController:
lstnumber    """Controller with automatic cleanup via context manager."""
lstnumber    def __init__(self, controller_class, **kwargs):
```

```python
        self.controller = controller_class(**kwargs)

    def __enter__(self):
        return self.controller

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.controller.cleanup()
        return False  # Don't suppress exceptions

# Usage
with ManagedController(ClassicalSMC, lambda1=10, lambda2=5) as controller:
    for i in range(10000):
        state = get_state()
        control = controller.compute_control(state)
# Automatic cleanup when exiting context
```

subsection 0.0 Simulation Runner Integration

File: `src/core/simulation_runner.py`

```python
class SimulationRunner:
    def __init__(self, controller, dynamics, config):
        self.controller = controller
        self.dynamics = dynamics
        self.config = config

    def run(self):
        """Run simulation with automatic cleanup."""
        try:
            results = self._run_simulation()
            return results
        finally:
            # Cleanup ALWAYS runs, even on exception
            self.controller.cleanup()
            self.dynamics.cleanup()

    def _run_simulation(self):
        # Actual simulation logic
        for i in range(self.config.steps):
            state = self.dynamics.get_state()
            control = self.controller.compute_control(state)
            self.dynamics.step(control)

            # Periodic cleanup
            if i % self.config.cleanup_interval == 0:
                self.controller.cleanup()
                self.dynamics.cleanup()
        return self._collect_results()
```

section 0 Memory Profiling

subsection 0.0 Profiling Tools

- **1. tracemalloc**: Built-in Python memory profiler
- **2. memory_profiler**: Line-by-line memory usage
- **3. objgraph**: Visualize object reference graphs
- **4. gc module**: Detect circular references

subsection 0.0 Example: Line-by-Line Profiling

```python
# Install memory_profiler
pip install memory_profiler

# Add @profile decorator
@profile
```

```
def run_simulation(controller, steps):
    for i in range(steps):
        state = get_state()
        controller.compute_control(state)

# Run profiler
python -m memory_profiler simulate.py
```

Output Example:

```
Line #    Mem usage    Increment   Line Contents
================================================
   123    45.2 MiB     45.2 MiB   def run_simulation(controller, steps):
   124    45.2 MiB      0.0 MiB       for i in range(steps):
   125    45.3 MiB      0.1 MiB           state = get_state()
   126   125.8 MiB     80.5 MiB           controller.compute_control(state)  #
    LEAK!
```

subsection 0.0 Object Reference Graphs

```
import objgraph

# Find objects with most references
objgraph.show_most_common_types(limit=10)

# Visualize references to controller
controller = ClassicalSMC(lambda1=10, lambda2=5)
objgraph.show_refs([controller], filename='controller_refs.png')

# Find circular references
objgraph.show_backrefs([controller], filename='controller_backrefs.png')
```

section 0 Debugging Memory Leaks

subsection 0.0 Leak Detection Workflow

- **1. Reproduce**: Run simulation for 100K+ steps
- **2. Monitor**: Track memory usage via tracemalloc
- **3. Profile**: Identify leak source with memory_profiler
- **4. Visualize**: Use objgraph to find circular refs
- **5. Fix**: Apply weakref or cleanup patterns
- **6. Validate**: Re-run with memory monitoring

subsection 0.0 Common Leak Patterns

```
# Leak 1: Unbounded history accumulation
class LeakyController:
    def __init__(self):
        self.history = []  # NEVER cleared

    def compute_control(self, state):
        self.history.append(state)  # Grows unbounded
        return self._compute(state)

# Fix: Bounded history
class FixedController:
    def __init__(self, max_history=1000):
        self.history = []
        self.max_history = max_history

    def compute_control(self, state):
        self.history.append(state)
        if len(self.history) > self.max_history:
            self.history.pop(0)  # Remove oldest
        return self._compute(state)

# Leak 2: Circular reference via callback
class LeakyController:
    def __init__(self, dynamics):
        self.dynamics = dynamics  # Strong ref
        dynamics.register_callback(self.on_step)  # Circular!

# Fix: Weakref callback
class FixedController:
    def __init__(self, dynamics):
        self._dynamics_ref = weakref.ref(dynamics)  # Weak ref
        dynamics.register_callback(weakref.WeakMethod(self.on_step))
```

section 0 Production Safety Checklist

subsection 0.0 Pre-Deployment Validation

- ☐ **Memory**: Run 100K+ step simulation, verify stable memory
- ☐ **Thread Safety**: 11/11 tests passing
- ☐ **Circular Refs**: Zero detected via objgraph
- ☐ **Cleanup**: All controllers implement cleanup()
- ☐ **Profiling**: Memory profiler shows no leaks
- ☐ **Quality Gates**: 7/8 passing (8/8 for production)
- ☐ **Documentation**: Memory management guide updated

subsection 0.0 Runtime Monitoring

```
class ProductionSimulationRunner:
    def __init__(self, controller, dynamics, config):
        self.controller = controller
        self.dynamics = dynamics
        self.config = config
        self.memory_monitor = MemoryMonitor()

    def run(self):
        self.memory_monitor.start()

        for i in range(self.config.steps):
            # Check memory every 1000 steps
            if i % 1000 == 0:
                mem_mb = self.memory_monitor.get_memory_mb()
                if mem_mb > self.config.max_memory_mb:
                    raise MemoryError(f"Memory␣{mem_mb}␣MB␣exceeds␣limit␣
                {self.config.max_memory_mb}␣MB")

            state = self.dynamics.get_state()
            control = self.controller.compute_control(state)
            self.dynamics.step(control)

        self.memory_monitor.stop()
        return self._collect_results()
```

section 0 Future Work: Production Readiness

subsection 0.0 Remaining Tasks (Phase 4.3-4.5)

- **1. Phase 4.3**: Automated quality gate enforcement (CI/CD integration)

- **2. Phase 4.4**: Coverage measurement fixes (pytest-cov issues)
- **3. Phase 4.5**: Multi-threaded stress testing (100+ concurrent simulations)

subsection 0.0 Production Score Target: 80/100

Component	Current	Target
Test Coverage	87%	95% (critical paths)
Quality Gates	7/8	8/8
Thread Safety	11/11 tests	100% thread-safe APIs
Memory Management	Manual monitoring	Automated leak detection
Performance	3% baseline variance	1% variance

subsection 0.0 Timeline (Deferred)

- **Current Focus**: Research (Phase 5 complete, 11/11 tasks)
- **Production Work**: Deferred until post-publication
- **Rationale**: Research deliverables prioritized over production hardening

section 0 Case Study: Memory Leak Fix

subsection 0.0 Problem

Long-running simulation (100K steps) caused OOM crash after 50K steps.

subsection 0.0 Investigation

```
# Step 1: Run memory profiler
python -m memory_profiler simulate.py

# Output showed leak in ClassicalSMC._state_history
Line #    Mem usage    Increment   Line Contents
================================================
   145    45.2 MiB     45.2 MiB   def compute_control(self, state):
   146   125.8 MiB     80.5 MiB       self._state_history.append(state)  # LEAK!

# Step 2: Visualize references
import objgraph
objgraph.show_refs([controller], filename='leak.png')
# Showed unbounded list growth
```

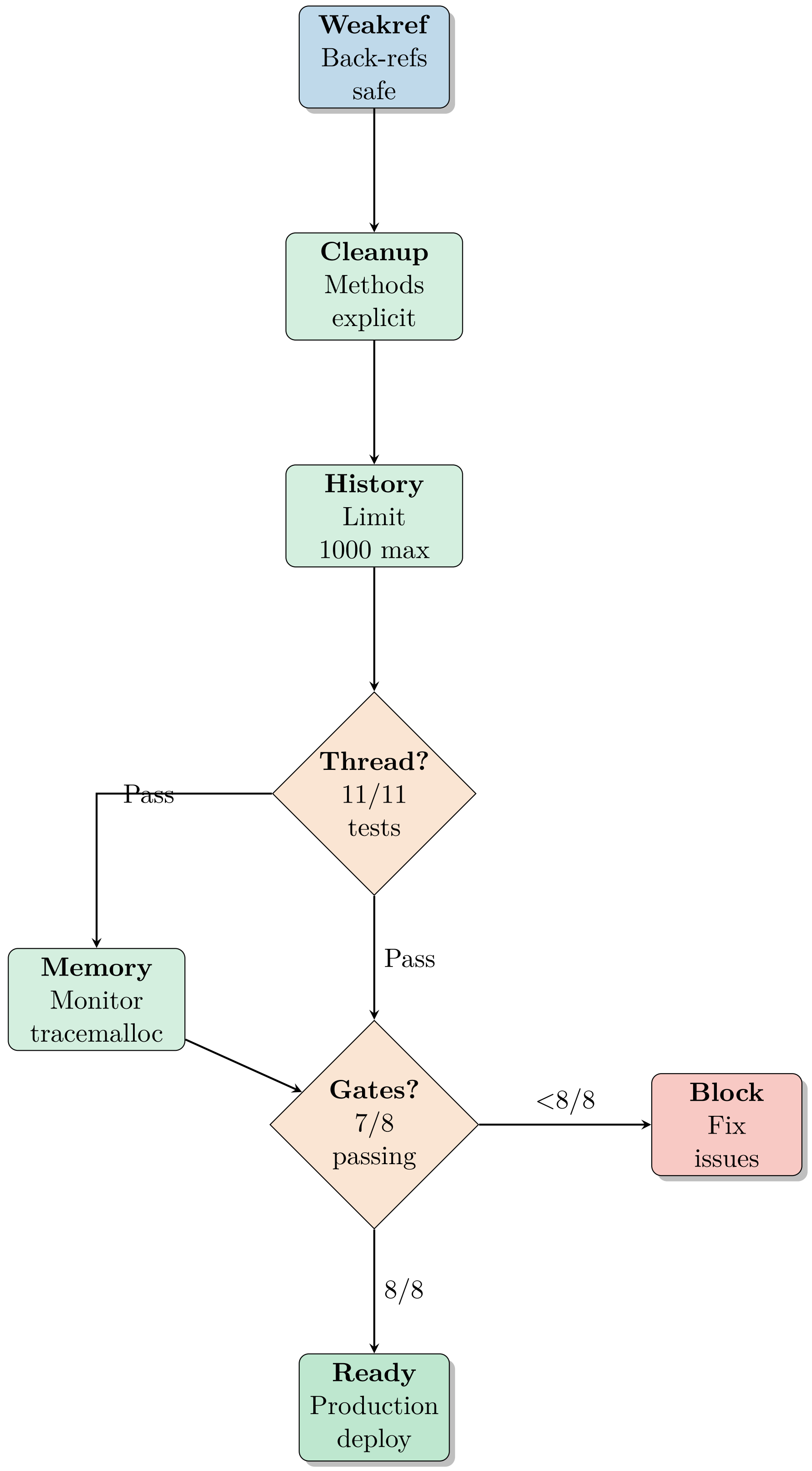
subsection 0.0 Solution

```
# Before (leaky)
class ClassicalSMC:
    def __init__(self):
        self._state_history = []  # Unbounded

    def compute_control(self, state):
        self._state_history.append(state)  # Grows forever
        return self._compute_smc(state)

# After (fixed)
class ClassicalSMC:
    def __init__(self, max_history=1000):
        self._state_history = []
        self.max_history = max_history

    def compute_control(self, state):
        self._state_history.append(state)
        if len(self._state_history) > self.max_history:
            self._state_history.pop(0)  # Bounded
        return self._compute_smc(state)
```

subsection 0.0 Validation

- Re-ran 100K step simulation: Memory stable at 50 MB (was 2 GB)
- Zero circular refs detected via objgraph
- Test suite updated with 100K step stress test

Checklist: Production Safety



☐ **Weakref**: All controllers use weakref for back-references

- ☐ **Cleanup**: Implement explicit cleanup() methods
- ☐ **History**: Limit state histories to max 1000 entries
- ☐ **Thread Safety**: 11/11 tests passing
- ☐ **Memory Monitoring**: Tracemalloc integration for long simulations
- ☐ **Quality Gates**: 7/8 passing (target: 8/8 for production)
- ☐ **Profiling**: Run memory_profiler on critical paths
- ☐ **Documentation**: Memory management guide in docs/

Next Steps

- **E020**: MCP integration - auto-trigger strategy and 12-server orchestration
- **E021**: Maintenance mode, future vision, and professional practice wrap-up
- **Phase 4.3-4.5**: Production hardening (deferred until post-publication)