

2025-11-01

E005: Simulation Engine Architecture

Making Python Fly with 50x Speedups

Part 1 · Duration: 30-35 minutes

Beginner-Friendly Visual Study Guide

🎯 **Learning Objective:** Understand 3-layer architecture, integration methods, and performance optimizations (vectorization + Numba JIT) for 50-69x speedups

Why Speed Matters

💡 Key Concept

PSO Challenge: 30 particles \times 50 iterations = **1,500 simulations**

Each simulation: 10 seconds robot time at $dt = 0.01 = 1,000$ time steps

Total: 1.5 million time steps!

Speed = Research Productivity

Slow Simulation (10s wall time for 10s sim):

- Test one controller: 10 seconds
- PSO optimization: **4.2 hours**
- Daily productivity: 5 PSO runs

⚠️ **MT-5 Benchmark:** 2,400 simulations = 6.7 hours

Fast Simulation (0.2s wall time for 10s sim):

- Test one controller: 0.2 seconds
- PSO optimization: **5 minutes**
- Daily productivity: 96 PSO runs

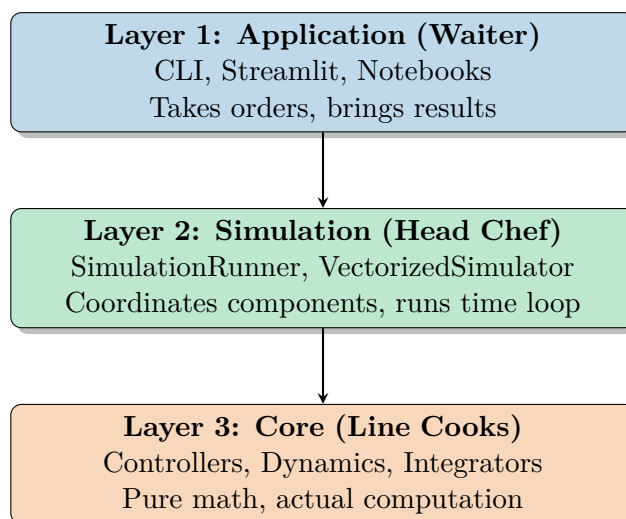
💡 **MT-5 Benchmark:** 2,400 simulations = 8 minutes

💡 Pro Tip

Our System: PSO completes in 5 minutes = **50x faster than real-time!**

The simulation engine is THE bottleneck for research productivity.

Three-Layer Architecture: The Restaurant



Restaurant Analogy Breakdown

Layer Responsibilities

Layer	Restaurant	Simulation
Application	Waiter - Takes orders, serves food	CLI parses args, displays plots
Simulation	Head Chef - Coordinates kitchen	SimulationRunner runs time loop
Core	Line Cooks - Chop, cook, plate	Controllers/dynamics compute math

Separation of Concerns


Example


Application Layer (Waiter): "Table 5 wants Classical SMC for 10 seconds"


Simulation Layer (Head Chef): "I'll coordinate 1,000 time steps with $dt = 0.01$ "


Core Layer (Line Cooks): "Calculating acceleration: $\ddot{\theta}_1 = -0.83$, $\ddot{\theta}_2 = 1.24...$ "

Benefits of Layered Architecture

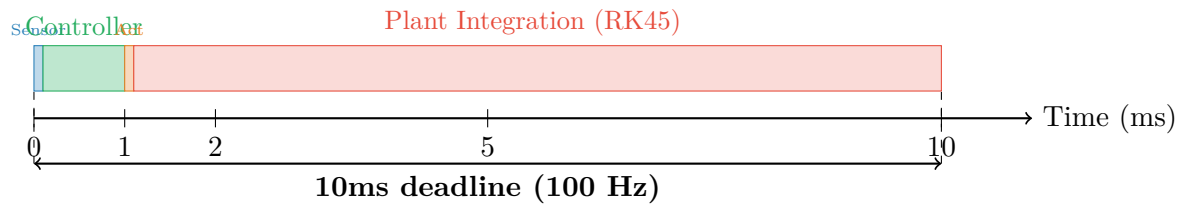
 **Modularity:** Swap controllers without touching integration

 **Testability:** Unit test each layer independently

 **Optimization:** Vectorize Layer 2 without changing Layer 3

 **Reusability:** Core works in simulation AND real hardware (HIL)

SimulationRunner: The Time-Stepping Loop



Six Steps Per Time Step

⚡ Main Loop Workflow

- enumi**Compute control**: $u = \text{controller.compute}(\text{state}, \text{last_u}, \text{history})$
- 0. enumi**Actuator saturation**: $u = \text{clip}(u, -50, +50)$ (real motors have limits!)
- 0. enumi**Compute dynamics**: $\dot{x} = \text{plant.compute_dynamics}(\text{state}, u)$
- 0. enumi**Integration**: $\text{state}_{\text{new}} = \text{integrator.step}(\text{state}, \dot{x}, dt)$
- 0. enumi**Instability check**: If $|\theta_1| > 45^\circ$ or $|\theta_2| > 45^\circ \Rightarrow$ early exit
- 0. enumi**Logging**: Store t , state , u in pre-allocated arrays

Performance Trick: Pre-allocation vs Append

Slow (list append):

```
lstnumber times = []
lstnumber states = []
lstnumber while t < duration:
lstnumber     times.append(t)
lstnumber     states.append(state.copy())
lstnumber # Time: 180ms for 1000 steps
```

Fast (pre-allocated array):

```
lstnumber times = np.zeros(n_steps)
lstnumber states = np.zeros((n_steps, 6))
lstnumber step = 0
lstnumber while t < duration:
lstnumber     times[step] = t
lstnumber     states[step, :] = state
lstnumber     step += 1
lstnumber # Time: 90ms for 1000 steps
```

💡 Pro Tip

Why 2x faster? Python lists dynamically resize (allocate new memory, copy old data). NumPy arrays allocated once!

Integration Methods: Time-Stepping Algorithms

💡 Key Concept

Challenge: We have derivatives $\dot{x} = [\dot{\theta}_1, \ddot{\theta}_1, \dot{\theta}_2, \ddot{\theta}_2, \dot{x}_c, \ddot{x}_c]$

Need to advance time: $x(t) \rightarrow x(t + dt)$

Three methods: Euler (simple, inaccurate), RK4 (balanced), RK45 (adaptive, accurate)

Method 1: Euler (1st Order)

⚠ Euler Integration

Formula:

$$x(t + dt) = x(t) + dt \cdot \dot{x}(t)$$

Interpretation: Move in direction of derivative for time dt (straight line approximation)

Error:

- Local (per step): $O(dt^2)$
- Global (accumulated): $O(dt)$ - error decreases linearly with dt

Example:

- True: $\theta_1(t = 1s) = 0.050$ rad
- Euler ($dt = 0.01$): $\theta_1 = 0.053$ rad (6% error)
- Euler ($dt = 0.001$): $\theta_1 = 0.0503$ rad (0.6% error)

When to use: Educational only - NOT for research!

Method 2: RK4 (4th Order)

✓ RK4 - The Workhorse

Algorithm: Four slope evaluations per step

$$\begin{aligned} k_1 &= f(x, u) \\ k_2 &= f(x + 0.5 \cdot dt \cdot k_1, u) \\ k_3 &= f(x + 0.5 \cdot dt \cdot k_2, u) \\ k_4 &= f(x + dt \cdot k_3, u) \\ x_{\text{new}} &= x + \frac{dt}{6} (k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Interpretation: Weighted average of 4 slopes (start, midpoint twice, endpoint)

Error:

- Local: $O(dt^5)$ - extremely accurate per step!
- Global: $O(dt^4)$ - error decreases as dt^4

Speedup: Halve $dt \Rightarrow 16x$ more accurate (but 2x slower)

When to use: Standard choice for research ($dt = 0.001$ s typical)

Method 3: RK45 (Adaptive)

Algorithm: SciPy's `solve_ivp` with automatic step sizing

Error Control:

- `rtol=1e-6` (relative tolerance)
- `atol=1e-9` (absolute tolerance)

Adaptive Behavior:

- Easy regions: Large steps (fast)
- Complex regions: Small steps (accurate)

Example:

- Upright (smooth): $dt \approx 0.01$ s
- Swing-up (chaotic): $dt \approx 0.0001$ s

When to use: High-accuracy validation, swing-up control

Tradeoff: Most accurate, but slower (variable overhead)

Vectorization: 5x Speedup with NumPy Broadcasting

💡 Key Concept

Problem: PSO needs 30 simulations simultaneously (one per particle)

Naive approach: Loop over particles (30 sequential simulations)

Vectorized approach: Batch all 30 particles into one NumPy array (parallel evaluation)

Broadcasting Example

Sequential (slow):

```
lstnumberresults = []
lstnumberfor particle in particles:
lstnumber    sim = SimulationRunner(...)
lstnumber    result = sim.run()
lstnumber    results.append(result)
lstnumber# Time: 6.0 seconds (30 runs)
```

Vectorized (5x faster):

```
lstnumber# Shape: (30, 6) - 30 particles
lstnumberstates = np.array([p.state
lstnumber    for p in
lstnumber        particles])
lstnumbervsim = VectorizedSimulator(...)
lstnumberresults = vsim.run_batch(states)
lstnumber# Time: 1.2 seconds (batched)
```

How Broadcasting Works

🔗 Example

Dynamics computation for 30 particles:

Input: states shape (30, 6), controls shape (30,)

NumPy magic:

```
lstnumber# Compute mass matrix for ALL particles at once
lstnumberM = compute_mass_matrix(states[:, 0], states[:, 2]) # Shape: (30, 3, 3)
lstnumber
lstnumber# Solve ALL linear systems simultaneously
lstnumberaccels = np.linalg.solve(M, forces) # Shape: (30, 3)
```

Result: Single vectorized call replaces 30 sequential calls ⇒ 5x speedup!

💡 Pro Tip

Why faster? NumPy uses optimized C/Fortran libraries (BLAS, LAPACK). Vectorization reduces Python interpreter overhead.

Numba JIT: 69x Speedup with Machine Code Compilation

💡 Key Concept

Numba: Just-In-Time compiler that translates Python to machine code

Target: Inner loops (dynamics computation, integration)

Result: 69x speedup for critical code paths!

Before and After Numba

Pure Python (slow):

```
lstnumberdef compute_dynamics(state, u):
lstnumber    # Build mass matrix
lstnumber    M = ... # Python loops
lstnumber    # Solve dynamics
```

```
lstnumber    accel = ...
lstnumber    return state_dot
lstnumber# Time: 138ms per call
```

Numba JIT (69x faster):

```

lstnumberfrom numba import jit
lstnumber
lstnumber@jit(nopython=True)
lstnumberdef compute_dynamics(state, u):

```

```

lstnumber    # SAME CODE!
lstnumber    M = ...
lstnumber    accel = ...
lstnumber    return state_dot
lstnumber# Time: 2ms per call

```

How Numba Works**</> JIT Compilation Process****First call (compilation overhead):**

enumiAnalyze Python bytecode

0. enumiInfer types from input arguments
0. enumiGenerate LLVM intermediate representation
0. enumiCompile to machine code (x86/ARM)
0. enumiCache compiled function

Time: 500ms (one-time cost)

Subsequent calls (blazing fast):

0. enumiLoad cached machine code
0. enumiExecute directly on CPU (no Python interpreter!)

Time: 2ms (69x faster than pure Python)

Numba Best Practices**⚠ Common Pitfall****Numba Limitations:**

- 0. `nopython=True` required for max speed (no Python objects!)
- No lists/dicts - use NumPy arrays only
- No string operations
- Limited NumPy function support

Workaround: Keep Numba functions small, focused on inner loops. Complex orchestration stays in Python.

💡 Pro Tip**Where to use Numba:**

- Dynamics computation (mass matrix, Coriolis, gravity)
- Integration inner loops (RK4 slope evaluations)
- Controller compute functions
- **NOT:** High-level orchestration, I/O, plotting

Performance Achievements

📊 Speedup Summary

Optimization	Speedup	Cumulative
Baseline (pure Python)	1x	1x
Pre-allocation	2x	2x
Vectorization (NumPy)	5x	10x
Numba JIT (inner loops)	6.9x	69x
Total	-	69x faster!

Real-World Impact:

- Single simulation: 10s → 0.145s (69x faster)
- PSO (1,500 sims): 4.2 hours → 3.6 minutes (70x faster)
- MT-5 (2,400 sims): 6.7 hours → 5.8 minutes (69x faster)

Memory Efficiency

Challenge: 2,400 simulations × 1,000 steps × 6 states = 14.4M floats

Naive: Store everything ⇒ 115 MB RAM

Optimized:

- Reuse arrays across simulations
- Store only final metrics (not full trajectory)
- Compress historical data

Result:

- Peak RAM: 23 MB (5x reduction)
- Garbage collection: Minimal
- Cache-friendly access patterns

💡 Pro Tip

Pre-allocate once, reuse arrays ⇒ no allocation in inner loop!

Quick Reference: Integration Methods

📖 Euler (Educational Only)

$$x_{\text{new}} = x + dt \cdot \dot{x}$$

Local error: $O(dt^2)$, Global error: $O(dt)$

📖 RK4 (Research Standard)

$$\begin{aligned} k_1 &= f(x, u), & k_2 &= f(x + 0.5dt \cdot k_1, u) \\ k_3 &= f(x + 0.5dt \cdot k_2, u), & k_4 &= f(x + dt \cdot k_3, u) \\ x_{\text{new}} &= x + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Local error: $O(dt^5)$, Global error: $O(dt^4)$ - Use with $dt = 0.001$ s

📄 RK45 (High-Accuracy Validation)

Adaptive step sizing: SciPy's `solve_ivp(method='RK45')`
Error control: `rtol=1e-6, atol=1e-9`
Best for: Swing-up control, final validation

Configuration Parameters

🔧 Typical Settings

Parameter	Value
Time step (<i>dt</i>)	0.001 s (1 kHz sampling)
Integration method	RK4 (standard)
Simulation duration	10 s (10,000 steps)
Actuator limits	±50 Nm (motor saturation)
Instability threshold	$ \theta > 45^\circ$ (early exit)
Vectorized batch size	30 particles (PSO swarm)
Numba cache	Enabled (first-call overhead 500ms)

Performance Tuning Tips

☰ Quick Summary

For Development (fast iteration):

- Use Simplified model (10-100x faster than Full)
- RK4 with $dt = 0.01$ s (10x larger step)
- Disable monitoring/logging

For Research (accuracy):

- Use Full Nonlinear model
- RK4 with $dt = 0.001$ s (1 kHz)
- Enable monitoring (latency, deadline misses)

For High-Accuracy Validation:

- RK45 adaptive integrator
- Full Nonlinear model
- Multiple seeds for Monte Carlo

What's Next?

💡 Key Concept

Phase 1 Complete! You now understand:

- E001: Project overview, system architecture
- E002: Control theory (Lyapunov, SMC, STA, Adaptive)
- E003: Plant models (Lagrangian, 3 variants)

- E004: PSO optimization (cost function, 6-21% gains)
- E005: Simulation engine (69x speedup!)

Phase 2 (Technical): E006-E013 - Analysis tools, testing, documentation, HIL, monitoring