2025-11-01

E010: Documentation System & Navigation

706 Files, 5 Learning Paths, Automated API Reference

Part 2 · Duration: 15-20 minutes

Beginner-Friendly Visual Study Guide

◎ **Learning Objective:** Understand the Sphinx documentation system (706 files), automated API reference generation, 5 learning paths, cross-reference validation, and freshness mechanisms

The Documentation Challenge

💡 **Key Concept**

Question: How do you document 105,000 lines of code across 358 files?
Answer: 706 documentation files in Sphinx + automated API generation + 5 learning paths
Key Insight: Assume future you will forget everything current you knows!

Three Core Principles

📄 Documentation Philosophy

Principle 1: Colocate Documentation with Code
Every controller in `src/controllers/` has corresponding page in `sphinx_docs/controllers/`
Principle 2: Automate What You Can
API reference generated from docstrings - NEVER write manually
Code changes → Docs update automatically (no synchronization problems!)
Principle 3: Multiple Navigation Strategies
5 learning paths for 5 audience types with different mental models

Sphinx: The Librarian for Scattered Notes

💡 **Key Concept**

What is Sphinx?
Gathers code comments, theory explanations, tutorial examples → Organizes into chapters → Creates TOC → Builds index → Adds cross-references → Outputs searchable website
Why Sphinx? Standard in Python ecosystem (NumPy, SciPy, Django, Flask all use it)
Alternative: Chaos - dozens of disconnected markdown files nobody can navigate

Location:

- `academic/paper/sphinx_docs/`
- `conf.py` - 19 KB of configuration

Source Subdirectories:

- `api/` - Auto-generated reference
- `controllers/` - Theory & implementation
- `benchmarks/` - Performance results
- `architecture/` - Design decisions
- `development/` - Contributor guides
- `deployment/` - Installation & production
- `examples/` - Runnable tutorials
- `for_reviewers/` - Academic paper materials

Build Process:

```
lstnumbersphinx-build -M html docs
    docs/_build
```

What It Does:

enumiReads `conf.py`

0. enumiParses markdown & reST files

0. enumiGenerates API from docstrings (autodoc)

0. enumiBuilds cross-reference links

0. enumiOutputs searchable HTML

Strict Mode:
`-W` flag: Warnings become errors
Catches broken links, missing docstrings, invalid markup EARLY

Directory Structure: 706 Files Organized

> **📂 Eight Main Categories**
>
> **1. api/** - Auto-generated API reference for 358 source files
> Every public class, function, module documented from docstrings
> **2. controllers/** - Theory & implementation (7 subdirectories, one per controller)
> Classical SMC, STA, Adaptive, Hybrid Adaptive STA, Swing-up, Terminal, Integral
> **3. benchmarks/** - Performance results from 11 research tasks (MT-5, MT-6, MT-7, MT-8, LT-4, LT-6, LT-7, etc.)
> **4. architecture/** - Design decisions and patterns
> Module design, controller factory, simulation engine, testing strategy
> **5. development/** - Contributor guides and coding standards
> **6. deployment/** - Installation, production setup, HIL integration
> **7. examples/** - Runnable tutorials (5 numbered scripts + markdown explanations)
> **8. for_reviewers/** - Materials for academic paper reviewers

API Reference: Automated Documentation

> **💡 Key Concept**
>
> **Key Insight:** Documentation should live NEXT TO the code it documents!
> Write docstring inside Python function → Sphinx auto-generates formatted reference pages
> Change function signature → Docs update automatically → NO manual copy-pasting!

Why Docstrings Prevent Staleness

> **⚠ Common Pitfall**
>
> **Separate Files Problem:**
> Change code → Forget to update docs → Docs lie → Confusion 6 months later
> **Docstring Solution:**
> Documentation IS the code → Sphinx just formats it nicely

NumPy Style Docstrings

> **</> Example**
>
> **Standard Format Forces You to Answer 4 Questions:**
>
> 0. enumiWhat does this function do?
>
> 0. enumiWhat inputs does it need? (Parameters section)
>
> 0. enumiWhat does it return? (Returns section)
>
> 0. enumiWhat errors can it raise? (Raises section)
>
> **Magic:** Write documentation ONCE in code → Appears in 3 places:
>
> - IDE autocomplete
>
> - Python `help(function)` output
>
> - Published HTML docs
>
> **Zero duplication!**

Enforcement: Missing Docstrings Fail Build

> **⚠ Strict Checking**
>
> **What Triggers Warnings/Errors:**
>
> - Missing docstring for public function
>
> - Missing parameter descriptions
>
> - Return type not documented
>
> - Example code doesn't run
>
> **Policy:** Function without proper docstring CANNOT be merged
> **Why?** Prevents 6 months of confusion when someone asks "What is this parameter?" and original author is gone

Learning Paths: Five Audience Types

> **⚠ Five Paths for Five Audiences**
>
> **Path 0: Complete Beginners**
>
> - **Background:** Zero programming/control theory
> - **Duration:** 125 hours over 4-6 months (about a semester)
> - **Content:** Python, physics, calculus, linear algebra, control theory, SMC fundamentals
> - **Location:** `.ai_workspace/edu/beginner-roadmap.md`
> - **Phases:** 5 phases (Computing basics → Math foundations → Physics → Control theory → SMC)
>
> **Path 1: Quick Start**
>
> - **Background:** Knows Python, wants immediate results
> - **Duration:** 1-2 hours
> - **Content:** Install → Run `python simulate.py` → See pendulum stabilize
> - **Location:** `docs/guides/getting-started.md` + Tutorial 01
>
> **Path 2: Advanced Usage**
>
> - **Content:** Custom controllers, PSO tuning, batch simulations
>
> **Path 3: Research Workflows**
>
> - **Content:** Running benchmarks, analyzing results, writing papers
>
> **Path 4: Production Deployment**
>
> - **Content:** Thread safety, memory management, HIL integration

Decision Tree in README.md

> **</> Example**
>
> **User Self-Navigation:**
>
> - "Have you programmed in Python before?" No → Path 0
> - Yes → "Do you know control theory?" No → Path 0 theory section
> - Yes → "Do you want to run a quick demo?" Yes → Path 1
> - "Do you want to write custom controllers?" Yes → Path 2
>
> **Each user finds their appropriate entry point**

Cross-References and Linkage

> **💡 Key Concept**
>
> **Challenge:** With 706 files, how do you prevent broken links?
> **Solution:** Sphinx cross-reference syntax + automated validation

Sphinx Reference Syntax

Internal Links:

```
lstnumber:ref:'section-label'
lstnumber:doc:'path/to/file'
lstnumber:class:'ClassName'
lstnumber:func:'function_name'
```

Sphinx validates ALL references during build
Emits warnings for broken links

External Links:

```
lstnumber[NumPy Docs](https://numpy.org/doc/)
```

Plain markdown syntax
Weekly link checker script validates external URLs
Files issue if dead links found

> **💡 Pro Tip**
>
> **Reorganize Docs?** Update references → Sphinx catches broken links → Fix → Rebuild passes
> Cannot manually verify 706 files - automated validation is CRITICAL!

Code Examples: Runnable Tutorials

</> Five Numbered Tutorials

Location: `examples/` subdirectory
Format: Runnable Python scripts with extensive comments

Tutorial	What It Teaches
01: `tutorial_01_quick_start.py`	Run simulation with default settings
02: `tutorial_02_custom_gains.py`	Modify controller gains and compare
03: `tutorial_03_pso_tuning.py`	Optimize gains with PSO
04: `tutorial_04_batch_simulation.py`	Run Monte Carlo validation
05: `tutorial_05_hil_setup.py`	Configure hardware-in-the-loop

Each Tutorial Has:

- Runnable Python script (copy-paste and run)

- Corresponding markdown in `sphinx_docs/examples/` (concepts + expected output)

Keeping Examples Up-to-Date

♀ Key Concept

Problem: Code changes → Examples break → Users confused
Solution: CI runs all tutorial scripts as integration tests
If tutorial fails → Build fails → Prevents merging broken examples

</> Example

Additional Validation:
Script checks if code blocks in markdown MATCH actual tutorial script files
Prevents copy-paste errors and drift between docs and code

Search and Indexing

♀ Key Concept

Challenge: 706 files - how do users find what they need?
Solution: Sphinx JavaScript search index

Search Granularity: Section-Level

What Gets Indexed:

- Every heading

- Every function name

- Every class name

- Significant terms

Example Search: "sliding surface"
Results:

- Theory pages

- API reference for sliding surface module

- Tutorial examples

What Does NOT Get Indexed:

- Code snippets (unless manually marked)

Manual Index Entry:

```
lstnumber.. index:: PSO; cost function
```

Makes "PSO cost function" appear in search

</> Example

Search: "boundary layer chattering"
Direct Links To:

- Section in Classical SMC theory page (boundary layer explanation)

- API reference for boundary layer parameter

- Benchmark results showing chattering reduction (MT-6)

Maintenance: Keeping 706 Files Fresh

> **💡 Key Concept**
>
> **Challenge:** How do you prevent 706 files from going stale?
> **Answer:** Three freshness mechanisms

> **Three Freshness Mechanisms**
>
> **Mechanism 1: Automated API Reference**
> NEVER goes stale - generated from code
> Change function signature → Docs update automatically on next build
> Zero manual work, zero synchronization problems
> **Mechanism 2: Link Validation**
> Weekly cron job checks all internal and external links
> Files GitHub issue if broken links found
> **Mechanism 3: Deprecation Warnings**
> Mark function as deprecated in docstring:
>
> ```
> lstnumber """
> lstnumber .. deprecated:: 0.8.0
> lstnumber     Use :func:`new_function` instead.
> lstnumber     Removed in version 1.0.
> lstnumber """
> ```
>
> Sphinx generates prominent warning box in HTML

Architecture Docs: Manual Review Required

> **⚠ Common Pitfall**
>
> **Policy:** Any PR that changes architecture MUST update corresponding `architecture/` docs
> **PR Template Checklist:** "Updated architecture docs? Yes/No"
> Reviewer verifies before approving
> **Why?** Conceptual docs explaining design decisions require human review (can't be automated)

Documentation Debt: Quarterly Audits

> **</> Example**
>
> **Audit Process:**
>
> enumiScript lists files in `src/` without corresponding docs in `sphinx_docs/`
>
> 0. enumiList goes on backlog
>
> 0. enumiPrioritize based on usage:
>
>   - Undocumented + Unused → Document or deprecate
>   - Undocumented + Used → HIGH PRIORITY, document immediately
>
> **Prevents slow decay into undocumented codebase**

Documentation Build Workflow

> **⚙ Three Build Scenarios**
>
> **Scenario 1: During Development**
> Write code → Write docstring → Run `sphinx-build` → Verify API reference looks correct
> **Scenario 2: Before Committing**
> Pre-commit hook runs `sphinx-build -W`
> Fails commit if documentation errors
> **Scenario 3: In CI/CD**
> GitHub Actions builds docs on every merge to main
> Deploys to GitHub Pages: `https://thesadeq.github.io/dip-smc-pso/`
> Users always see latest docs matching main branch

Incremental Builds: Speed Optimization

First Build:

- All 706 files parsed

- Time: 30 seconds

Subsequent Builds:

- Sphinx caches parsed content

- Only rebuilds changed files

- Time: 2-3 seconds (1 file changed)

Key Takeaways

☰ Quick Summary

706 Files Organized: 8 categories (API, controllers, benchmarks, architecture, development, deployment, examples, reviewers)

Sphinx System: Gathers scattered docs → Organizes → Creates TOC → Builds index → Cross-references → Searchable HTML

API Reference: Auto-generated from NumPy-style docstrings (change code → docs update automatically)

5 Learning Paths: Path 0 (beginners, 125 hrs), Path 1 (quick start, 1-2 hrs), Paths 2-4 (advanced, research, production)

Cross-References: Sphinx validates ALL internal links during build (broken links → warnings → build fails)

Runnable Tutorials: 5 numbered Python scripts in `examples/`, CI runs as integration tests

Freshness: (1) Automated API, (2) Weekly link validation, (3) Deprecation warnings

Path 0 Beginner Roadmap: 5 phases over 4-6 months (Computing → Math → Physics → Control theory → SMC)

Quarterly Audits: List undocumented modules → Prioritize by usage → Document or deprecate

Build Workflow: Development (verify) → Pre-commit (enforce) → CI/CD (deploy to GitHub Pages)

Search: Section-level indexing (headings, function names, class names, manual index entries)

Documentation as Design Tool: Write docstring → Often realize function too complex → Simplifies design

Quick Reference: Documentation Commands

🔖 Build Sphinx Documentation

```
lstnumberStrict mode (warnings are errors) sphinx-build -W docs docs/_build
lstnumberIncremental build (only changed files) sphinx-build -M html docs docs/_build (automatic caching)
lstnumberServe locally python -m http.server 9000  -directory
    docs/_build/html Navigate to http://localhost:9000
```

🔖 Docstring Template (NumPy Style)

```
lstnumberExtended description explaining what the function does and why.
lstnumberParameters ----- param1 :  type Description of param1 param2 :  type Description of param2
lstnumberReturns ---- return_type Description of return value
lstnumberRaises --- ErrorType When this error occurs """  Implementation
```

🔖 Cross-Reference Syntax

```
lstnumberLink to document :doc:'path/to/file'
lstnumberLink to API :class:'ClassName' :func:'module.function_name' :meth:'ClassName.method_name'
lstnumberExternal link (markdown) [NumPy Documentation](https://numpy.org/doc/)
```

What's Next?

> **💡 Key Concept**
>
> **E011: Configuration & Deployment**
> How `config.yaml` validates parameters, Pydantic ensures type safety, deploying in different environments
> **Remember:** Documentation is not a chore - it's a design tool! If you can't explain it clearly, you shouldn't build it.