

WORKSHOP · ~2 HOURS

Files, not chats

Claude Code as a co-scientist — for mathematicians

A hands-on workshop for researchers in computational mathematics — numerical PDEs, optimization, and friends. The four-file architecture (CLAUDE.md, plans/, LOGBOOK.md, STATUS.md), skills, version control, testing, and literature workflows — with runnable exercises.

Contributors

Rebecca Durst · Srini Eswar · Sven Leyffer · Andreas Wilke

Who this workshop is for

You don't need any AI experience. You *do* need the kind of work that benefits from a focused collaborator.

You're in the right room if

- You write or apply numerical solvers.
- You're comfortable with Python and the command line.
- You want less yak-shaving (plot scripts, log parsing, refactors) and more research.

Out of scope

- Training or fine-tuning models.
- Building chatbots or web apps.
- "Replace your solver with an LLM."
We won't.

What you'll leave with

Central insight: durable AI collaboration is built from **files, not from chat history**. The whole workshop is a tour of which files to keep and how they fit together.

- ❶ **The four-file architecture** — CLAUDE.md (conventions), plans/ (forward-looking), LOGBOOK.md (durable knowledge), STATUS.md (current state).
- ❷ **Skills** — packaging recurring domain logic (KKT checks, performance profiles) as reusable artifacts.
- ❸ **Sustainable habits** — version control, recognizing whack-a-mole loops, ending sessions cleanly.
- ❹ **A capstone** — an open-ended project applying everything to your own research, or one of several provided briefs.

How the workshop is structured

Two stages: short guided exercises first, then an open-ended capstone where you put the tools to work on your own research.

Stage 1 · Exercises

- Small, self-contained examples showing how individual Claude features work — CLAUDE.md, plan mode, skills, LOGBOOK.md.
- *Not* “you must only use it this way” prescriptions — starting points.
- Keep asking: *how could a tool like this help me in my own research?*

Stage 2 · Capstone

- Open-ended time to try the tools on your own — your code, your problem.
- Push past what you saw in the exercises.
- Goal: actually accomplish a piece of research with Claude as collaborator, not just rehearse a feature.

Setting up each exercise

Every exercise ships a README.md (instructions) and a SOLUTION.md (walkthrough). Those are for *you* — not for Claude.

- Keep README.md and SOLUTION.md **outside** the directory where you run claude. If Claude can see them, it reads ahead and the exercise loses its point.
- As you work, **manually copy only the code and data files you need** into your working directory — e.g. max_conopt.py, matrix_A.npy, laplace_square.py.

Rule of thumb: Claude should see the *problem*, never the *solution*.

Optimization

exercises-opt/01-claude-md/
exercises-opt/02-planning/
exercises-opt/03-skills/
exercises-opt/04-logbook/

PDE / FEM

exercises-pde/01-claude-md/
exercises-pde/02-planning/
exercises-pde/03-skills/
exercises-pde/04-logbook/

Linear algebra

exercises-lin_alg/
01-claude-md/
exercises-lin_alg/
02-planning/
exercises-lin_alg/
03-skills/
exercises-lin_alg/
04-logbook/

What is Claude Code?

An **agentic coding assistant** in your terminal: it reads and writes files, runs commands you approve, and works through multi-step tasks the way you would.

- Not autocomplete — it plans, reads context, edits files, runs tests.
- You stay in control: every edit and command is reviewable.
- Runs locally; nothing leaves your machine without your action.

Our framing throughout: **Claude Code as a co-scientist** — a collaborator who has read the manual for every library you use but needs you to specify the problem precisely. The four files are how you brief it.

Where it actually helps you

Routine work it absorbs

- Plotting helpers (semilog convergence, performance profiles).
- Parsing solver logs into a tidy DataFrame.
- Translating between Pyomo $\leftrightarrow$ AMPL $\leftrightarrow$ JuMP.
- CUTEst-style test harnesses.
- $\text{\LaTeX}$ -ing derivations to paper-ready form.

Research work it accelerates

- “What line-search rule does Wächter–Biegler 2006 use?” — answered from the codebase.
- Rescaling a primal-dual update.
- Large CUTEst sweeps with consistent logging.
- Glue between PETSc, your driver, and your benchmark DB.

How a session works

Everything Claude reads and writes in a session lives in one finite buffer — the **context window**.

- Every prompt, file read, and tool output goes into the same window.
- Large but bounded — the context window is model-specific.
- Near the limit, Claude Code **auto-compacts**: older messages are summarized to make room.
- Compaction is lossy — details from early turns can fade. Keep anything that matters on disk.

Long sessions are fine — but because compaction is lossy, treat the running conversation as ephemeral. Anything that matters belongs on disk.

What persists, what doesn't

	Survives a new session?	Survives auto-compact?
Files in your project (code, LOGBOOK.md)	Yes	Yes — on disk
CLAUDE.md	Yes — auto-loaded each session	Yes — kept in context
Skills	Yes — loaded on demand by description match	Re-loaded as needed
The conversation itself	Only with --resume	Compressed to a summary
"Mental notes" Claude has in-conversation	No — they vanish	Often lost

If a fact matters tomorrow, write it to LOGBOOK.md today. If it's a reusable procedure, lift it into a skill. **The conversation is scratch paper.**

Save, resume, clear

```
# Resume the most recent session in this project  
claude --continue  
  
# Pick from a list of past sessions  
claude --resume  
  
# Inside a session  
/clear      # wipe the conversation; keep CLAUDE.md, skills, files  
/compact    # manually summarize older turns to free context space
```

Sessions are stored locally under `~/.claude/projects/ihashi/`. They're plain files; you can grep them.

Don't lean on `--resume` to remember decisions — it's brittle. Persist anything that matters to files. We cover the end-of-session ritual with LOGBOOK.md and STATUS.md in Part 4.

PART 1 . THE BRIEF

CLAUDE.md

A markdown file at the root of your project. Claude Code reads it at the start of every session. Treat it as the briefing document for a new collaborator.

CLAUDE.md, in practice

Two details make it more than a static readme:

- Keep a personal one at `~/.claude/CLAUDE.md` for cross-project preferences.
- It can reference other files: see `memory/solvers.md` for our notation.

Mental model: the README that's specifically for the AI — it sits beside your README, doesn't replace it.

Why mathematicians need one

Without one, every session starts from zero. With one, Claude knows your conventions before it writes a line — notation, tooling, tolerances, tests, and hard “don’ts”:

- **Notation.** x primal, λ equality mult., μ inequality mult.
- **Tooling & tolerances.** Pyomo / IPOPT / BARON / petsc4py; tolerances default to $1e-8$.
- **Don’ts.** “No `scipy.optimize.minimize` for constrained problems — it ignores active sets near the boundary in our setup.”

Each exercise supplement opens with the domain-specific version of this list.

Anatomy of a good CLAUDE.md

```
# Project: Inertia-Corrected Interior-Point Method
## Goal
Develop a primal-dual IPM with inertia correction for nonconvex NLPs.
## Stack
- Python 3.11, Pyomo for modeling, casadi for AD where Pyomo is awkward
- petsc4py for sparse linear algebra; mumps via PETSc as the default solver
- pytest for tests; matplotlib for figures (semilog by default)
## Commands
- 'pytest -q tests/'
- 'python bench/run.py --suite cutest_small'
## Conventions
- Variables: x (primal), y (eq mult), z (ineq mult). Use these names.
- Tolerances default to 1e-8.
- Never call scipy.optimize for constrained problems (see logbook/decisions.md).
## Pointers
- logbook/decisions.md - solver and parameter choices
- logbook/dead-ends.md - things we tried and why they failed
- docs/derivations.tex - the math in publishable form
```

Exercise 1

Please refer to your supplemental slide deck. Open your track's exercise:

`exercises-opt/01-claude-md/` `exercises-pde/01-claude-md/`
`exercises-lin_alg/01-claude-md/`

PART 2 · THINK FIRST

Planning

Multi-step algorithmic work goes badly without a plan. Plan mode forces Claude to lay out every step before touching a file.

Why plan first?

Research code has tightly coupled steps — formulation, discretization, scheme, parameters, verification. Without a plan, the assistant fixes one and breaks another.

- A written specification before any code — the plan is itself a deliverable.
- You catch wrong assumptions on a screen, not in a debugger three hours later.

Same discipline as writing pseudocode before implementing a new method — same payoff.

Plan mode in Claude Code

- Press Shift+Tab twice — the footer turns into *plan mode*.
- State the task; Claude reads context, asks clarifying questions, proposes a plan.
- Approve to execute, or revise.

Or start directly by typing `plan: jtask` at the prompt.

```
plan: implement a filter line search for our IPM driver,  
      then add a CUTest-style test for problem HS071 with  
      tolerances matching logbook/decisions.md.
```

What a good plan looks like

Plan: filter line search for IPM

1. Read driver.py and identify the current Armijo step.
2. Add 'filter.py' with the (theta, phi) pair and a switching condition.
3. Modify driver to call filter.accept(...) instead of armijo.accept(...).
4. Update tests/test_linesearch.py: add filter-acceptance tests on
(i) a synthetic 2D problem, (ii) HS071 from CUTEst.
5. Run pytest -q tests/test_linesearch.py and report.

Files to be modified: driver.py, filter.py (new), tests/test_linesearch.py

Open questions:

- Filter envelope parameters: use Wachter-Biegler defaults?
- Should we record filter contents in LOGBOOK.md after each run?

Read the open questions before approving — that's where most surprises live.

Plans as artifacts: the four-file architecture

A plan is the fourth kind of file — alongside CLAUDE.md, LOGBOOK.md, STATUS.md.

File	Says	Lifecycle
CLAUDE.md	How the project works	Stable; rarely edited
plans/jdateĭ-jslugĭ.md	What's going to happen next	Created → active → executed/abandoned
LOGBOOK.md	What's been learned	Append-only; references plans by filename
STATUS.md	Where I am right now	Overwritten each session; points at active plan

```
project/
|-- CLAUDE.md
|-- LOGBOOK.md    # references plan outcomes
|-- STATUS.md     # points at the active plan
'-- plans/
    |-- 2026-04-08-filter-linesearch.md    executed
    |-- 2026-04-22-tikhonov.md             abandoned
    '-- 2026-05-07-tao-implementation.md  active
```

Prompts that set up and maintain the architecture

```
# Set up the structure in a fresh project
> create CLAUDE.md, LOGBOOK.md, STATUS.md, and a plans/ directory
  in this project, using the workshop conventions. Leave each file
  as a clearly-marked template I'll fill in.

# At the end of a planning session
> save the plan to plans/$(date +%Y-%m-%d)-tao-implementation.md
  and update STATUS.md to point to it as the active plan. Don't
  implement yet.

# At the end of an execution session
> append a dated decision to LOGBOOK.md citing
  plans/2026-05-07-tao-implementation.md, then overwrite STATUS.md
  with the next concrete step inside that plan.

# When abandoning a plan
> add an "Abandoned: 2026-05-09, reason: ..." header to
  plans/2026-04-22-tikhonov.md, and append a Dead Ends entry in
  LOGBOOK.md that references it.
```

Exercise 2

Please refer to your supplemental slide deck. Open your track's exercise:

`exercises-opt/02-planning/` `exercises-pde/02-planning/`
`exercises-lin_alg/02-planning/`

PART 3 · REUSABLE EXPERTISE

Skills

A skill is a folder. It contains instructions and helper scripts for one specific kind of task. Claude loads it on demand when the task fits.

What are skills?

- A **skill** is a folder named after the capability, with a SKILL.md at its root — YAML front-matter (name, description) plus instructions, snippets, references.
- The description triggers it: when the task matches, Claude loads the body.
- Lives in `.claude/skills/` (project) or `~/.claude/skills/` (global).

Skills package “the thing my lab always does” — once — and reuse it across projects, students, and papers.

Anatomy of a skill

```
.claude/skills/kkt-checker/  
|-- SKILL.md           # front-matter + instructions  
|-- check_kkt.py       # the actual implementation  
'-- references/  
    '-- nokedal-wright-12.4.md
```

```
---  
name: kkt-checker  
origin: workshop  
description: |  
    Verify that a candidate primal-dual point  $(x, y, z)$  satisfies the KKT  
    conditions for an NLP  $\min f(x)$  s.t.  $g(x) = 0, h(x) \leq 0$ . Use this  
    whenever the user reports a "solution" and wants to confirm it is  
    actually a KKT point, or wants residuals component-by-component.  
---  
# KKT checker  
Run 'python .claude/skills/kkt-checker/check_kkt.py --problem PATH ...'  
Returns max residuals for stationarity, feasibility, complementarity.
```

origin: tags skills you wrote (workshop, mygroup) versus imported ones (community).

Built-in skills you already have

Document-creation built-ins

- **pdf** — extract text and tables from papers; build/fill PDFs.
- **docx** — produce Word reports (referee replies, grant text).
- **pptx** — generate slides from results.
- **xlsx** — turn a benchmark log into a workbook with charts.

When custom beats built-in

- The same domain check appears across projects.
- The recipe involves exact CLI calls (PETSc, GAMS) you don't want to re-discover.
- You want to encode “house style” — figure dimensions, log binning, etc.

Exercise 3

Please refer to your supplemental slide deck. Open your track's exercise:

`exercises-opt/03-skills/` `exercises-pde/03-skills/`
`exercises-lin_alg/03-skills/`

PART 4 · THE DIARY

LOGBOOK.md

Long-running research generates decisions, dead-ends, and parameter folklore. LOGBOOK.md is where those facts live so the next session can use them.

CLAUDE.md vs LOGBOOK.md

	CLAUDE.md	LOGBOOK.md
Lifespan	Stable across the project	Evolves with the research
Content	Conventions, commands, do/don't	Decisions, parameter values, dead-ends
Edited by	You, deliberately	You and Claude — at the end of each session

Small projects: one LOGBOOK.md. As it grows: a logbook/ directory (solvers.md, experiments.md, dead-ends.md) with an INDEX.md; promote cite-worthy decisions to a numbered ADR in decisions/.

We call it LOGBOOK.md, not MEMORY.md: Claude Code auto-generates its own machine-local MEMORY.md — keep the two separate.

What to record

- **Decisions.** “IPOPT over SNOPT for license reasons — 2026-03-12, issue #41.”
- **Parameter folklore.** “mu_init=1e-2 fastest on our cohort; 1e-4 diverges on the catalysis problem.”
- **Dead ends.** “Newton + Tikhonov on inverse Poisson converges but multipliers are noise. Don’t repeat.”
- **Open questions.** “Is the inertia-correction threshold problem-dependent? Re-check HS071, HS099.”
- **Run logs.** “Best so far: runs/2026-04-22T1430, $\|\nabla f\| = 3.7\text{e-}9$.”

Keep it append-only and dated — LOGBOOK.md is a log, not a document you rewrite. Capturing these entries is part of the end-of-session ritual.

Handoff: STATUS.md, not LOGBOOK.md

LOGBOOK.md holds durable knowledge and grows. **Don't put session handoff in it** — “mid-debug on X, next step Y” is true today, noise next week. Use a small, separate, *overwritten* file:

```
# STATUS - 2026-05-07 17:42

## What I'm working on
Filter line search for the IPM driver. PR branch: filter-linesearch.

## Current state
- driver.py and filter.py compile, unit tests pass on synthetic 2D.
- HS071 fails with KeyError: 'theta_max' in filter.accept(...).
- Haven't read filter.py:43-67 yet.

## Next step
Read filter.py:43-67. If theta_max not set in __init__, add it.
Then re-run pytest tests/test_linesearch.py -k HS071.
```

End-of-session ritual: append a dated entry to LOGBOOK.md and overwrite STATUS.md with where we are now. Two minutes.

Handoff mechanisms, ordered by reliability

Mechanism	Reliability	Notes
STATUS.md you wrote	Highest	You control it; survives anything.
Branch name	High	<code>git branch --show-current</code> tells you what's in flight.
Last commit message	High	A WIP commit is a one-line handoff for free.
<code>claude --continue</code>	Medium	Brings back the conversation. Brittle: may already be compacted; gone if you switch machines.

The pattern: rely most on what's in *files* (top three rows). Rely least on what's only in chat state (last row). CLAUDE.md, LOGBOOK.md, STATUS.md — the deck of files — is what survives whatever a session throws at you.

Exercise 4

Please refer to your supplemental slide deck. Open your track's exercise:

`exercises-opt/04-logbook/` `exercises-pde/04-logbook/`
`exercises-lin_alg/04-logbook/`

Report back

Before the capstone, each team shares what came out of the exercises — 2–3 minutes each.

Share

- What did you get working? (CLAUDE.md, a plan, a skill, LOGBOOK.md)
- What surprised you — good or bad?
- Where did Claude go wrong, and how did you catch it?

Discuss

- Which file — CLAUDE.md, plans/, LOGBOOK.md, STATUS.md — earned its keep?
- What will you try on your own research?
- Open questions for the room.

Capstone: your code, your problem

Open-ended time. Your code, your problem — Claude as collaborator. Or pick a provided brief:

BQPD-CPP A thread-safe C++ port of Fletcher's robust QP solver.

CUTEst-C A clean, dependency-free C implementation of CUTEst.

ConDFO A derivative-free solver for constrained problems.

OptiUNO Run-time-configurable Newton-like solvers, hunting better combinations.

PrecisionBench Mixed-precision benchmarking for numerical linear algebra.

SciCompSkills A hallucination-resistant verified-literature-search skill.

SolverBench A reproducible benchmark of solvers & preconditioners for PDEs.

VoroMesh2D Voronoi-based interstitial meshing in 2D.

diffBLAS Adapt differentiated BLAS routines to AD activity contexts.

optSolverBench Benchmark optimizers & solvers for PDE-constrained optimization.

PART 5 · THE FIFTH ARTIFACT

Literature

Reading and citing the literature is daily work for computational mathematicians — and where Claude as a co-scientist earns its keep.

Treat it as a first-class artifact, alongside the four files.

What Claude does well — and doesn't

Lean on it for extraction and lookup; keep the judgment yourself — and never trust a citation from memory.

	Claude does well	Claude does badly
PDFs	Extract text, equations, tables (built-in pdf skill)	Decide what a result <i>really means</i> — still you
Notation	Translate Nocedal–Wright / Conn–Gould–Toint / Boyd–Vandenberghe	Catch subtle B- vs. M- vs. S-stationarity without your check
Citations	Resolve fragments via arxiv / semantic-scholar	Emit citations from memory — often plausible and wrong

```
project/
|-- CLAUDE.md
|-- LOGBOOK.md   # cites plans/ AND literature/
|-- STATUS.md
|-- plans/
'-- literature/
    |-- wachter-biegler-2006.md
    |-- nocedal-wright-12.md
    '-- INDEX.md   # contents page once it grows past ~10
```

When the corpus outgrows context: RAG

Your group's PDFs, lab wiki, and shared notes don't fit in one Claude session.

Retrieval-Augmented Generation (RAG): index the corpus into a vector database; for each question, retrieve the most relevant chunks; feed only those into the model.

- Rely on what you actually have, not on what Claude was trained on.
- Citations come back as document IDs you can verify.
- The corpus grows without inflating your context window.

```
Your lab MediaWiki -> wiki-rag (Milvus + LangGraph)
                        | exposes /v1/chat/completions
                        v
                    Custom tool wrapper
                        | tool: query_lab_wiki(question)
                        v
                    Claude Code session
```

Pattern: any RAG that speaks the OpenAI API can become a Claude Code tool in ten lines.

PART 6 · POWER FEATURES

Beyond the basics

Four mechanisms that change what's possible: checkpoints for reversible exploration, subagents for context-isolated work, hooks for deterministic enforcement, and non-interactive mode for fan-out at scale.

Checkpoints — try, rewind, try again

Every action Claude takes creates a checkpoint. Press Esc Esc (or run `/rewind`) to open a menu and restore prior state.

- **Restore conversation only** — get back the chat, keep the code.
- **Restore code only** — undo Claude's file edits, keep the chat.
- **Restore both** — full rewind to a prior moment.
- **Persists across sessions** — close the terminal, rewind tomorrow.

Two reverting tools at different scales: **git** for committed history, **/rewind** for in-session backtracks. Use them together.

Caveat: `/rewind` tracks changes *made by Claude*. External edits, build artifacts, and database state are out of scope. Not a replacement for git.

Subagents and hooks

	Skill	Subagent	Hook
Lives in	.claude/skills/	.claude/agents/	.claude/settings.json
Context	Loads into your main session	Separate window — reports a summary	None — just runs a script
Best for	Recurring domain logic (KKT check)	Investigative reads (200-file solver)	Things that must happen <i>every time</i>
Trigger	Description-match on the task	You ask Claude to delegate	Specific events (PostToolUse, etc.)
Reliability	Advisory — usually loaded	Advisory — when delegated	Deterministic — guaranteed to fire

- **Subagent.** A solver-archaeologist (tools: Read, Grep, Glob) reads 200 files of inherited code and writes a one-page summary — your main session never sees the 200 files.
- **Hook.** Promote “run pytest -q after every edit” from a CLAUDE.md rule (advisory) to a hook (deterministic) — it fires whether Claude remembers or not.

Non-interactive mode — fan-out at scale

`claude -p "prompt"` runs Claude without a session — single shot, machine-readable output. This makes Claude usable as a step in a pipeline.

```
# Run a solver across the CUTEst small cohort, log per-problem
for prob in $(cat cutest_small.txt); do
  claude -p "solve $prob with mu_init=1e-2 using our IPM driver. \
    Report status, iters, KKT residual as JSON." \
    --allowedTools "Bash(python *),Read,Write" \
    --output-format json >> runs/cohort_2026-05-07.jsonl
done

# Pipe data in directly
cat error.log | claude -p "extract the failing test names as a list"
```

`--allowedTools` scopes permissions; `--output-format json` for parseable results; `--permission-mode auto` lets a classifier handle approvals.

PART 7 · HABITS

Working sustainably

Version control, reverting, and recognizing when you're whack-a-mole-ing your own code. The unglamorous habits that make the rest of the workshop pay off.

Git hygiene with an AI collaborator

- **Start every session with a clean working tree** (or a fresh branch). Any change Claude makes should be visible in `git diff`.
- **Commit small, atomic changes.** One logical step per commit makes reverting precise.
- **Review the diff before continuing.** Don't prompt on top of code you haven't read.
- **Branch for risky work.** A solver rewrite goes on its own branch.

```
# Before a session
git status                # confirm clean
git checkout -b filter-linesearch # branch for this task

# After Claude makes changes
git diff                  # always read this
git add -p                # stage hunks selectively
git commit -m "Add filter line search; HS071 test passes"
```

Reverting when Claude went the wrong way

```
# Undo edits to one file (uncommitted)
git restore path/to/file.py

# Undo all uncommitted changes
git restore .

# Set aside half-baked work for later
git stash push -m "WIP filter rewrite"
git stash pop

# Throw away the last commit but keep its changes
git reset --soft HEAD~1

# Nuclear: drop the last commit and its changes (only if pushed nowhere)
git reset --hard HEAD~1

# Try a different approach in parallel without disturbing main
git worktree add ../alt-approach
cd ../alt-approach && claude
```

Worktrees are handy with Claude: one tree exploring a regularization scheme, another an SQP variant, without merge headaches.

Verification first — tests are how Claude checks itself

The highest-leverage thing you can do is give Claude a way to verify its own work. Without it, you are the only feedback loop; tests let Claude run, see, and fix.

	As a programmer	As a co-scientist
What it checks	Code does what it's supposed to	Results haven't degraded
Example	<code>assert filter.accept(x) == True</code>	"On HS071, IPOPT converges in ≤ 50 iters with <code>mu_init=1e-2</code> "
Frequency	Every commit	Nightly, or before a paper
Lives in	tests/	bench/ (or tests/ with a marker)

Skills as oracles: `kkt-checker`, `perf-profile`, and `convergence-rate-fitter` aren't only one-off tools — they're the predicate that powers an entire family of tests. One predicate, many invocations.

Test patterns with Claude in the loop

- **Test-first prompting.** “Write the test for X first, confirm it fails, then implement X.” Forces specification before code.
- **Hard rule in CLAUDE.md.** “Run `pytest -q` after every edit; surface failures before continuing.”
- **Test the dead end, not just the success.** Assert the failure mode so a future fix can’t silently re-enable the bug.
- **Cite tests in commits and decisions.** LOGBOOK.md decisions reference the test that catches their regression.

```
# In CLAUDE.md
## Conventions
- After any code edit: run ‘pytest -q tests/’. Surface failures
  in the next reply; don’t claim done.
- For changes touching solver behavior, also run
  ‘python bench/run.py --suite cohort_smoke’ and report iters/time.
```

The whack-a-mole loop — how to spot it

Symptom: every fix breaks something else. Fix A $\rightarrow$ B fails $\rightarrow$ fix B $\rightarrow$ C fails $\rightarrow$ fix C $\rightarrow$ A fails again.

Signs to watch for

- The same files reappear in git status across attempts.
- Tests flip between passing and failing on the same code.
- Claude proposes a fix you rejected two turns ago.
- Diffs grow but the bug count is stable.

```
git log --oneline -20      # any file touched repeatedly?  
git log -p path/to/disputed.py  # are the same lines flipping back?
```

Underlying causes

- A missing invariant that no test enforces.
- Conflicting requirements between two modules.
- Silent fallback paths (a try/except hiding the failure).

Breaking out of a stuck session

- ❶ **Stop prompting “fix this too.”** More turns won't help; step back.
- ❷ **Ask Claude to summarize.** summarize everything we've tried and why each attempt failed. Read it.
- ❸ **Step away from the keyboard.** Write down — on paper — the actual invariants the code must satisfy.
- ❹ **Reset.** /clear. Fresh session with the invariants explicit in the prompt or appended to CLAUDE.md.
- ❺ **Re-plan.** Plan mode, with the new invariants. Don't accept a plan that doesn't address them.
- ❻ **Test the invariants.** Add a regression test for each. The loop becomes a stable solution.

Mathematician's variant: the loop usually means you haven't formalized something. Once you write the invariant in math, the fix becomes obvious — even to you.

Seven habits worth keeping

- 1 **Set up the four-file architecture early.** CLAUDE.md, plans/, LOGBOOK.md, STATUS.md.
One job per file.
- 2 **Plan before you build.** Plan mode for anything beyond a one-line edit. Save the plan to plans/`idate`-`slug`.md before implementing.
- 3 **Promote recurring tasks to skills.** Anything you do twice becomes a SKILL.md.
- 4 **Commit before you prompt.** A clean tree means every change is visible in git diff and reversible in one command.
- 5 **Test both code and results.** Unit tests in tests/; experimental-regression tests in bench/.
- 6 **End every session with the two-line ritual.** Append a dated entry to LOGBOOK.md; overwrite STATUS.md with the next concrete step.
- 7 **When you're whack-a-mole-ing, stop and formalize.** Loops mean an invariant is unwritten. Write it, test it, and the loop ends.

Files vs. built-ins: how this maps to Claude Code

Two of the four files are the official path; two deliberately turn ephemeral built-ins into durable, versioned files.

This deck	Claude Code natively
CLAUDE.md	The native memory file — hierarchy + @import. Same thing.
Skills	Native, loaded on demand. Same thing.
plans/	Plan mode is a permission mode — ephemeral , never written to disk. We persist it.
LOGBOOK.md	CLAUDE.md <i>is</i> the memory; we split durable facts into a file it points at (named to avoid clashing with Claude Code's auto MEMORY.md).
STATUS.md	--resume / checkpoints are machine-local — a committed file crosses machines, CI, and collaborators.

Same diagnosis as the official guide — context degrades as it fills — but a different remedy: **write it to disk.**

Where to go from here

- **Authoritative reference:** docs.claude.com/en/best-practices — Anthropic's official guide. Our four-file architecture and literature material are opinion-laden extensions of it.
- Documentation: docs.claude.com/claude-code
- Source & issues: github.com/anthropics/claude-code
- Skills examples: the `.claude/skills/` in this workshop's exercises.
- A power-user reference setup: github.com/affaan-m/everything-claude-code — useful as a layout example, not a starting point.
- Optimization-specific:
 - PETSc/TAO: petsc.org — the TAO manual chapter is short and excellent.
 - Pyomo: pyomo.readthedocs.io CasADi: casadi.org
 - CUTEst: github.com/ralna/CUTEst

Questions, or stories of what worked: keep notes in LOGBOOK.md. The next session will thank you.