From Source-Bound Proofs to Reproducible Research Artifacts: An NSRW Case Study

Yun Kwansub

Version 0.6 — 31 October 2026

Abstract

Formal research depends on more than the availability of a proof file: the file must refer to the intended source definitions, compile against an identified environment, and remain connected to the execution described in a publication. We present the implementation and research use of the NavierStokes Research Workbench (NSRW), organized around a construction-specific investigation of mass and angular histories. The investigation progresses from moment comparisons to actual repair-coefficient positivity and a same-point mass-zero/angular-positive theorem. Its software separates development compilation, fixed-checkpoint replay, archive checking and mathematical interpretation. We describe input identity, fresh external-module generation, binary log preservation and explicit failure boundaries, and analyze actual packaging errors, proof elaboration errors and an archive-preservation loss. A retained replay compiled fourteen external modules after one source-target build. In a separate local check for this manuscript, sixty-four focused software tests passed. These observations establish concrete implementation behavior and a reproducible-input contract, rather than comparative performance or independently reproduced mathematics. The case illustrates how research software can support the discovery and composition of formal consequences while keeping failed experiments and incomplete evidence distinguishable.

Keywords: research software; formalization; reproducible computation; dependency management; failure preservation.

1 Introduction

Scientific software is not merely a container for final results. It also mediates the transition from a proposed mathematical argument to a checked consequence, and from that consequence to a record another researcher can inspect. In formal mathematics, a declaration that appears correct in an editor can still be uncompiled, imported under the wrong module root, or compiled against an unintended dependency. Conversely, a compilation failure may be a packaging defect rather than a mathematical counterexample.

NSRW was developed within an investigation of an outgoing-profile construction supplied as Lean source [1]. The research question concerns a prospective ratio built from mass and angular histories. Its outcome is a construction-specific statement: each fixed Profile admits, for every sufficiently small positive parameter, one interior first-repair point where mass vanishes and angular history is positive. The accompanying mathematical manuscript [2] explains the proof. Here the subject is the executable research process that produced and recorded the formal propositions.

The contribution is a concrete implementation account: a source-bound proof dependency graph, separate development and committed replay operations, preserved execution outputs, and tests of specified rejection paths. We use actual research incidents to explain why those

boundaries were introduced. We do not evaluate NSRW as a general-purpose replacement for proof assistants, build systems or artifact repositories.

2 Context and related work

Guidance on reproducible computational research emphasizes recording how results were produced and preserving exact program versions [3]. Scientific-software practice also treats version control and testing as integral parts of research [4]. NSRW applies those practices to an environment in which mathematical declarations, rather than numerical samples alone, are the principal outputs.

Lean 4 supplies the formal language and checking infrastructure [5]. NSRW consumes that infrastructure; it does not supply a new proof kernel. The additional software identifies the inputs to a bounded run, invokes the required build and external compilation sequence, records outputs and checks the retained archive.

ACM's artifact policy distinguishes artifact evaluation, availability and validation of results [6]. These categories motivate a useful separation in this study: providing pinned inputs, retaining a successful reference execution and obtaining an external reproduction are different achievements. No ACM badge is sought or asserted here.

These references provide methodological context. The comparisons in this paper are functional comparisons of responsibilities, not empirical rankings of competing systems.

3 Study design and materials

3.1 Research case

The case concerns two source histories M and J and the prospective normalized ratio J/(M H). The mathematical investigation has three logically different components:

1. comparison of main and template moments, followed by a source-linked coefficient calculation;
2. transfer from main-only positivity to the actual coefficient of one fixed Profile near zero;
3. source geometry, boundary signs, an interior mass zero and a weighted-tail argument for positive angular history at that zero.

The proof modules are inspected as research inputs, not as interchangeable labels for a repository-wide verified state. The final theorem uses a common Profile, parameter, radius and source radial coordinate for both histories.

3.2 Evidence examined

The primary materials are the source files, replay wrapper, development engine, targeted regression tests, dated execution records and two final proof receipts. The reference execution is the retained run of 12 October 2026 at NSRW revision `3a50b5f50b034319a469b9de68f556be70fff02c` [7, 8]. The upstream source revision is `8937a8f4cbc7abaab5e9e97d1cc7f5d2319d9538` with Lean 4.34.0-rc2.

This is a retrospective single-project implementation study of the October 2026 investigation, supplemented by a local software check performed on 14 October 2026. It contains no controlled comparison, recruited reviewer study or independent external reproduction experiment. Execution records and subsequent author reviews share an operator trust domain. Counts below describe named stages and tests, not samples from a performance study.

4 Implementation

4.1 Source binding and dependency graph

The mathematical dependency graph is explicit in Lean imports. The actual-coefficient decomposition imports the upstream outgoing Profile directly. Later positivity combines that decomposition with the independently derived main-only coefficient result. Conditional mass geometry can be developed without importing the small-parameter positivity theorem; the final composition brings the two branches together.

The replay compiles the full external dependency order:

```
upstream OutgoingProfile build
  -> moment positivity -> strict main comparison -> template comparison
  -> moment-difference sign -> main-only coefficient positivity
  -> actual decomposition -> positivity at zero and continuity
  -> positive parameter neighborhood
  -> local mass identities -> boundary signs -> conditional interior zero
  -> source-facing mass-zero theorem
  -> conditional angular comparison -> common-point source composition
```

Fourteen external modules are generated in this sequence. A standalone development run may compile a smaller required subgraph. Full replay is a preservation and input-binding decision, not a reason to add dependencies to the mathematical modules.

4.2 Development and committed replay

Development compilation answers whether the selected source bytes compile in the specified environment. It must permit an active research worktree without pretending that those bytes are already committed. The development engine copies inputs into a new run directory and places generated modules in a distinct artifact directory.

Committed replay adds a different requirement: the execution checkout must match an expected commit, be tracked-clean and provide the exact committed proof and driver bytes. `scripts/run-p2-z-replay.py` binds fourteen proof inputs, a shared source-identity helper, the development engine and the wrapper: seventeen committed inputs in total. Earlier dependency blobs are pinned rather than resolved from the current branch tip.

The wrapper checks that the engine still names the expected dependency and final theorem surfaces. This prevents an apparently valid invocation from silently running a different or smaller calculation. The execution engine remains responsible for the actual compilation sequence.

4.3 Identity across hosts

Committed Git blob identity and runtime byte identity serve different purposes. The blob identifies version-controlled content portably. Runtime SHA-256 records the executor's actual bytes. Text checkout line endings may change a raw digest without changing the intended Git content; those two observations are not collapsed.

The source helper checks the pinned source revision, committed dependency-manifest blob, toolchain declaration, observed Lean version and manifest-listed dependency revisions. It also rejects untracked Lean-like shadow files outside the bounded build/cache exclusions. The filesystem inspection includes ignored files rather than relying only on a Git query that hides them. These are checks of specified paths and identities, not a proof against all possible host compromise.

An important distinction remains: a named Git object and a hash are identifiers. They do not establish that an unauthenticated third party performed the run or that the host was uncompromised.

4.4 Compilation and failure recording

The source target is built before the external modules. Each external artifact must be absent before its compilation and present afterward; its digest is recorded. The source target artifact is observed before and after external compilation to detect an unexpected change. The run also checks source and checkout identities again after execution.

The drivers capture stdout and stderr as binary files. Readable previews can decode with replacement, but the raw bytes remain the execution record. Each stage records its command and exit behavior. The final wrapper requires the exact expected stage, input and artifact sets and checks retained bytes against their recorded digests.

At the outer wrapper boundary, exceptions produce error metadata and a preserved archive. An executable/path failure is not interpreted as mathematical failure. All execution metadata leaves mathematical claim interpretation unresolved; a later source-and-evidence review specifies which named propositions the successful compilation supports.

4.5 Axiom-surface inspection

The final modules print nineteen named axiom surfaces. The wrapper requires one occurrence per named theorem and the expected three axioms: `propext`, `Classical.choice` and `Quot.sound`. Missing or duplicate output, a different surface or `sorryAx` is rejected.

This is a precisely bounded observation. It detects specified output defects and placeholder exposure, not semantic mistakes in every imported definition. The proof assistant, source-object correspondence and manuscript interpretation address different parts of the research claim.

5 Observed results

5.1 Reference compilation

Observation	Recorded value	Interpretation
Source target	`NavierStokes.OutgoingProfile`	Required upstream target built in the reference run
Compilation stages	15, all exit 0	One source-target stage and fourteen external modules
External artifacts	14 fresh modules	Not reuse of previously compiled external proof outputs
Committed inputs	17	Explicit proof/helper/engine/wrapper identity surface
Final printed axiom surfaces	19	Expected set, no observed `sorryAx`
Stage stderr	Empty in all stages	Does not imply absence of every stdout warning
Original archive	815,996 bytes	Distinct retained run, not a public derivative
Independent external replay	None recorded	Not supplied by the reference run or the focused tests

The original archive's SHA-256 is `bd109a6f0db1eaaa1e6054d5ba16f8e07696434f7ffebb5527128f721267d74b`. The retained archive checker compared sixty recorded member digests and the specified stage, input, artifact and axiom observations [7, 8]. The reference source build

reused an upstream cache. The external modules were fresh; the entire upstream library was not rebuilt from scratch.

The retained source identity records its commit, dependency-manifest blob, manifest-listed checkout revisions and observed toolchain. The source-target artifact also has a pre/post digest. These observations are not a digest inventory of every imported cache file or a complete cache-origin attestation. No cold-build comparison was performed. A future cold-build study could address a different reproducibility question, but its timings and memory use are not inferred from this run.

5.2 Focused software check for manuscript preparation

On 14 October 2026, the existing Windows Python 3.14.3 environment ran:

```
python -m pytest tests/test_p2_z_replay.py tests/test_p2_zplus_j_composition_development.py -o
    addopts= -q
```

The invocation also supplied a fresh local test directory and a JUnit output path. All 64 tests passed. The JUnit record has SHA-256 `9855e25082030acce15b3ecbaf715226f8f80c80dffa6fb7c18463f290ff6055`.

The tests include synthetic perturbations of stage completion, stderr, claim metadata, logs, input and artifact sets, engine dependency declarations, source identity, checkout identity and named axiom output. They check non-overwriting behavior and preservation of failures. The synthetic cases do not invoke Lean; their passing result is evidence about wrapper behavior, not a second formal proof execution.

Table 1 separates the three evidence populations used in this paper.

Evidence population	**Procedure**	**Observed result**	**Question addressed**
Formal reference execution	One pinned-source build plus fourteen fresh external compilations	Fifteen stages exited zero; final output surfaces checked	Did the recorded inputs compile in that execution?
Focused software tests	Sixty-four cases across two test modules, including success, mutation and boundary cases	Sixty-four passed	Did the tested driver and wrapper controls behave as expected?
External reproduction	Execution by a separate investigator	Not performed in the available record	Can someone outside this operator domain reproduce the result?

The test categories include input/checkout binding, declared dependency and theorem surfaces, output freshness/non-overwriting, stage/log/artifact integrity, axiom-output parsing, and failure preservation. The total is sixty-four tests, not sixty-four independent attacks or sixty-four formal compilations.

6 Research incidents and their resolution

6.1 Invocation failures without mathematical interpretation

Earlier normalized-moment development attempts encountered an invalid include option, an unavailable compiled dependency and an input outside the selected source root. These failures required module-root and search-path corrections. Treating them as mathematical failures

would have misclassified the experiment; reusing their incomplete outputs as successful evidence would have been equally incorrect.

The recovered procedure compiled the dependency in the same run, assigned its logical module name using the external source root and made the fresh artifact available to the importing module. This is a packaging repair, distinct from changing the theorem.

6.2 Elaboration repair in the weighted-tail proof

The first angular-comparison development run failed with an unnecessary tactic and type-inference problems in two weighted inequalities. The corrected proof removed the redundant tactic and provided explicit types for the nonnegative base and multiplied inequality. The conditional theorem's premises and conclusion were retained.

The corrected run on 11 October compiled the source target and three external modules, with twelve printed axiom surfaces. It has its own preserved successful archive. The first failed execution archive was subsequently absent from the inspected runtime and local preservation locations; copies of its inputs do not recover that archive [9]. The success is a distinct execution, not a reinterpretation of the missing failed record.

6.3 Loss of an earlier replay archive

A later 11 October full replay was reported successful, but its outer archive was not available in the inspected preservation locations. The 12 October replay generated new evidence for the same fixed inputs. The records maintain the earlier preservation gap and do not call the new archive a reconstruction of the old one.

This incident is informative for the software design: compilation success and durable preservation are separate outcomes. An ephemeral notebook output is an observation, not a substitute for retained run bytes.

Table 2 gives the observable signal and the scope of each intervention. It summarizes documented incidents rather than reporting a newly constructed experimental corpus.

Incident	**Observed failure signal**	**Minimum intervention**	**What was retained**	**Subsequent check**
External-module invocation	Invalid include option, missing compiled dependency, or input outside source root	Correct logical module root/search path and compile the dependency in the run	The selected scaffold proof bytes during invocation repair	Separate durable compile of the corrected invocation
Weighted-tail elaboration	Redundant tactic and two unresolved inequality elaborations	Remove the tactic; add explicit base/inequality types	Conditional theorem premises and conclusion	New retry compilation and its separately preserved archive
Earlier replay preservation	Expected original ZIP absent at inspected locations	Run a new clean replay after authorization	Old run identity and its unresolved archive gap	New archive size/digest and internal checks for the new run only

6.4 A rejected research assumption

The source analysis rejected the study's own endpoint-first-zero candidate. Support termination and endpoint cancellation imply a terminal zero interval before the endpoint in the stated source analysis. This is recorded as an analytic consequence, not a separately compiled plateau theorem [10]. It changed the research target toward interior repair geometry.

That example is important because NSRW did not merely confirm a fixed proposal. A source-derived contradiction redirected the investigation. It is not a report that an upstream theorem was disproved.

7 Reproduction and distribution

An external replay requires the exact execution checkout, the pinned source environment, the earlier dependency-base checkout and a new output directory outside those checkouts. The following is a **POSIX-shell command template for Linux/Colab**, not a PowerShell command or evidence of native Windows Lean replay. The retained runner uses the Linux toolchain environment described in the execution record.

```
python scripts/run-p2-z-replay.py \
  --source-root <pinned-source-checkout> \
  --nsrw-root <clean-execution-checkout> \
  --dependency-base <clean-checkout-at-10d5a1f517b8665137f2354746d70752f8ef67fd> \
  --output-dir <new-run-directory> \
  --expected-nsrw-commit 3a50b5f50b034319a469b9de68f556be70fff02c
```

Angle-bracket values are host-specific substitutions, not executable literals. The runner does not install Lean, update the manifest or recover a missing cache. Environment preparation is a separate operation. New logs, timings and generated artifact hashes are run-dependent; input identity is what remains pinned. The Windows Python test invocation in Section 5.2 is a separate host-side software check and is not a Windows example of this formal replay.

Distribution has a second boundary. The private original archive contains raw logs, runtime metadata and compiled outputs. The selected reproduction asset published with software release v0.4.1 contains exact inputs, post-execution receipts, selected axiom lines, an allowlisted stage summary and explicit transformation metadata. It excludes raw logs, runtime metadata, binary outputs and caches. Consequently it supports inspection of the distributed subset and a new replay, not inspection of every byte of the reference execution.

The GitHub CI configuration has Python, research-control and scoped upstream Lean lanes. Those lanes do not replace this P2 full-graph replay. A hosted CI execution at revision 6de19be9c413a4f7e85db1697c59908f50887195 completed successfully across six jobs; it is a separately identified software observation, not a rerun of the reference formal execution [11]. A software CI badge and a formal execution record answer different questions.

7.1 Subsequent quantitative extension

After the reference execution analyzed here, a separate replay at revision `7f29ebf0e2b894bf598144628230ae09df436c3d` supported four quantitative-margin and six normalization/plateau propositions. A dated receipt accepted those ten named exact-real propositions on 22 October 2026 [12]. They extend the local angular-history result with a positive lower bound at admissible first-window mass zeros. This follow-up has its own input and execution identity; it does not change the fifteen stages, nineteen axiom surfaces or sixty-four software tests reported for the earlier study. Neither execution constitutes an explicit numerical certificate or independent external reproduction.

7.2 A bounded transition to exact-rational computation

A subsequent software experiment evaluates only the core scalars required by the normalized lower-bound argument, at revision `adbaeeb82df3bdb9d6d6ea5a041fadc8f35f1b9b`. Its exact input uses $\lambda = 10^{-12}$; $P = m = 1$ and $\eta = 2^{-20}$ are recorded, while the reset bound K is unassigned. It computes $\ell = e^{-23/20}$, $v = e^{-17/20}$, $(v - \ell)/4$, the positive weight gap G, and the main-coefficient denominator D. The factor $\exp(6499999999987)$ is retained symbolically in a lower-bound expression for the second-bump moment, not a two-sided enclosure of that moment.

Scalar arithmetic uses Python `Fraction` integers and outward dyadic rounding. For $0 \le y \le 1/2$, truncating the positive exponential series after degree N gives the remainder bound

$$0 \le R_N(y) \le \frac{a_{N+1}}{1 - y/(N+2)}, \qquad a_k = \frac{y^k}{k!}.$$

Range reduction, interval squaring and positive reciprocals cover the required arguments. Small positive `expm1` arguments are summed directly. At precision b, endpoints lie on the absolute grid 2^{-b}; this is not a floating-point unit in the last place. The implementation document gives the domains and propagation rules [13].

Records at 128, 256 and 512 fractional bits were produced and recomputed on 31 October 2026. One local producer timing per precision was approximately 0.010, 0.015 and 0.033 seconds; these are single-run observations, not comparative benchmarks. Fifty-five focused tests cover a separate tighter positive-series enclosure calculation, actual CLI file I/O, malformed and duplicate-key JSON, a forged dyadic endpoint and attempted scope promotion. The full local software suite reported 645 passed, 3 skipped and 22 subtests, with 90.53% statement coverage. These checks do not re-execute Lean proofs.

The verifier shares the producer algorithm, so successful recomputation is not an independent arithmetic implementation or external reproduction. No affine-coefficient integral, numerical $q > 0$, explicit source Profile, mass-zero bracket or numerical J certificate is computed. This small executable link from exact-real normalizations to core-scalar enclosures leaves coefficient evaluation and source-instance admissibility explicit.

8 Discussion

Three implementation decisions were especially useful in this case. Exact dependency surfaces made smaller development runs possible without obscuring later full replay. Separating compilation from interpretation prevented metadata from turning a program exit into an automatically expanded mathematical claim. Preserving failed and successful runs under distinct identities prevented repairs from erasing the research path.

These benefits are observed functions of the implementation, not quantified productivity gains. The study does not measure developer time, false-negative rates, cross-platform performance or improvement over a competing tool. Manual source reading and manuscript review remain necessary. The archive checker and executor also belong to the same project and trust domain.

Maintenance should remain proportional to the research. A new schema or generalized framework is not needed merely to record another named proposition. The existing scripts, source imports and short execution records provide the relevant boundaries; public documentation should explain the mathematics before enumerating those controls.

9 Conclusion

NSRW demonstrates an executable path from source-defined mathematical objects to composed formal consequences and a preserved fixed-input research record. The case includes both a substantive mass/angular mismatch result and concrete failures of invocation, proof elaboration and preservation. The implementation makes those events distinguishable and gives another researcher an explicit replay contract. Independent reproduction and broader comparative evaluation remain opportunities for assessing the software, rather than outcomes implied by the retained successful run.

Availability and author disclosure

Code inputs are fixed by the revision in Section 3. Dated documents [7, 8, 9, 10] have their own document lineage, separate from the execution revision. The software is publicly archived under DOI https://doi.org/10.5281/zenodo.23381778. GitHub release v0.4.1 (https://github.com/flamehaven01/NavierStokes-Research-Workbench/releases/tag/v0.4.1) provides the selected reproduction ZIP and the companion P01 v0.4 PDF and source. That release predates the subsequent quantitative extension and does not contain its proof or replay inputs. This P02 manuscript is distributed separately in Markdown, standalone LaTeX and PDF, with its own versioned checksum manifest; no separate P02 DOI is assigned in this manuscript. The private original archive and the published selected reproduction asset are distinct records. The reproduction guide (`research-papers/REPRODUCE.md`) specifies the distributed evidence and execution procedure. The original proof files, replay runner and receipts were not changed to prepare this manuscript. AI assistance supported research, code development and writing; Yun Kwansub is the named human author. This is an author-reviewed single-project study, not an independent methods evaluation.

Copyright © 2026 Yun Kwansub. The authored manuscript text and its LaTeX representation are licensed under Creative Commons Attribution 4.0 International. NSRW code and Lean proof files retain their MIT license; third-party material retains its own applicable terms. See license scope (`research-papers/LICENSE-PAPERS.md`).

References

[1] OpenAI and repository contributors. *NavierStokesAndEuler*. Revision `8937a8f4cbc7abaab5e9e97d1cc7f5d2319d9538`. Source.

[2] Yun Kwansub. *A First-Repair Mass Zero Without Angular-Moment Cancellation*. Companion manuscript, version 0.5, 2026. Manuscript (`research-papers/P01_MassZero_AngularMismatch.md`).

[3] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor and Eivind Hovig. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9(10): e1003285, 2013. doi:10.1371/journal.pcbi.1003285.

[4] Greg Wilson et al. Best Practices for Scientific Computing. *PLOS Biology* 12(1): e1001745, 2014. doi:10.1371/journal.pbio.1001745.

[5] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction—CADE 28*, LNCS 12699, 2021. doi:10.1007/978-3-030-79876-5_37.

[6] ACM. *Artifact Review and Badging*, version 1.1, 2020. Policy.

[7] NSRW. *Z phase recovery replay*, 12 October 2026. The distribution includes a bounded public index (`colab/Z_PHASE_REPLAY_2026-10-12.md`) of the recorded execution, not its raw runtime log or full private metadata.

[8] NSRW. *P2 Z-core* and *P2 Z+J Colab execution receipts*, 12 October 2026. Mass-zero receipt (`docs/stage_receipts/P2_Z_CORE_COLAB_2026-10-12.md`); angular/source-composition receipt (`docs/stage_receipts/P2_ZPLUS_J_COLAB_2026-10-12.md`).

[9] NSRW. *Development and preservation history*, 11–12 October 2026. Selected documentary summary (`research-papers/supplement/DEVELOPMENT_HISTORY.md`). This summary identifies its documentary sources and is not an original runtime log.

[10] NSRW. *P2 dilation-ratio obstruction study*, at execution revision `3a50b5f50b034319a469b9de68f556be70fff02c`, section C3 on the terminal zero plateau. Fixed research analysis.

[11] NSRW. *Software CI at revision* `6de19be9c413a4f7e85db1697c59908f50887195`. https://github.com/flamehaven01/NavierStokes-Research-Workbench/actions/runs/39750987817. Observed 31 October 2026; this is not independent mathematical reproduction.

[12] NSRW. *P1 quantitative-margin and normalization receipt*, 22 October 2026. https://github.com/flamehaven01/NavierStokes-Research-Workbench/blob/6de19be9c413a4f7e85db1697c59908f50887195/docs/stage_receipts/P1_QUANTITATIVE_MARGIN_COLAB_2026-10-22.md. Execution revision `7f29ebf0e2b894bf598144628230ae09df436c3d`; same-project review of ten named propositions.

[13] NSRW. *Exact-rational core-scalar method*, 31 October 2026. https://github.com/flamehaven01/NavierStokes-Research-Workbench/blob/v0.5.0/docs/P2_CORE_SCALAR_METHOD_2026-10-31.md. Code checkpoint `adbaeeb82df3bdb9d6d6ea5a041fadc8f35f1b9b`.

A Inspected implementation locations

Responsibility	**Repository file**
Fixed execution binding and archive review	`scripts/run-p2-z-replay.py`
Source identity and bounded shadow-input checks	`colab/run-c02-development.py`
Full external composition engine	`colab/run-zplus-j-composition-development.py`
Wrapper perturbation and preservation tests	`tests/test_p2_z_replay.py`
Development-engine failure tests	`tests/test_p2_zplus_j_composition_development.py`
Local archive checker	`colab/check-z-replay-archive.py`
Current CI configuration	`.github/workflows/ci.yml`

B Measurement boundary

The 64-test check ran on the existing Windows environment without new installations or a Lean invocation. Its JUnit output is retained locally under the manuscript-preparation evidence directory. The reported reference formal compilation is the 12 October run, not this 14 October software check. The test result is not evidence of a newly executed GitHub CI matrix, external attestation or benchmark superiority.