

CORE SERIES I

Projector-Resolved Seed Calculus I

Typed operator systems, seed rank, and exact packet coordinates

---

Lean and evidence guide

Komorebi P.

Release edition / 15 October 2026

Reading scope

This guide connects the current Paper I manuscript to its formal sources, coverage map, and verification evidence. The release navigation in Section 2 identifies the current source tree and distinguishes complete correspondences from partial or unmapped records.

The tutorial and its complete 35-row contract describe the preserved v1.0.2 component: 20 source files, 84 mapped targets, and 1,126 historical namespace entries. Those counts belong to that component. The later continuation has a larger source closure and a separate map to the expanded manuscript's 108 records. The two inventories are retained with their exact scopes.

The release evidence is indexed separately from the historical receipts. Rendering this guide is a presentation check. Compiler acceptance, logical kernel replay, source identity, and source-to-statement correspondence are reported as distinct checks.

Contents

1 Reading the three companions together **4**
1.1 What the numbers count 4
1.2 Suggested reading routes 4

2 Current release: sources, scope, and evidence **5**
2.1 The continuation beyond the historical contract 5
2.2 Release modules and presentation conventions 6
2.3 Accepted release verification 7
2.4 Current proof-level correspondence index 7

3 The type system as mathematical bookkeeping **9**
3.1 Proof terms and the tactics used here 10

4 The source architecture **10**

5 Exact packet coordinates, from assumptions to inverse laws **11**
5.1 Why the two directions use different hypotheses 12
5.2 Four coordinates and two parity packets 13

6 Transfer blocks, support, and transport **14**
6.1 Reconstruction and multiplication 14
6.2 Why nonzero packets appear in converse statements 14
6.3 A reachability obstruction is a lower-bound certificate 15
6.4 Coordinate transport and strict naturality 15

7 Generation and seed rank **15**
7.1 The one-seed presentation . . . 16
7.2 Surjections, image algebras, and two product actions . . . 17
7.3 A complete constructive example: full matrix cyclicity . . . 18

8 Certificate inputs and what their proofs establish **18**
8.1 The split-semisimple formula . . . 18
8.2 Single-operator seed number: the upper and lower halves . . . 19
8.3 The cyclic-matrix bridge . . . 20
8.4 Graphs, paths, and cycles . . . 20

9 Topology and mathematical algorithms **20**
9.1 Bounded packets and closed-hull obstructions . . . 20
9.2 Dense realizations and completed presentations . . . 21
9.3 Orbit closure: why termination is not code extraction . . . 21
9.4 The elliptic example and scalar discipline . . . 23

10 Replaying the preserved verification **23**
10.1 A normal provisioned Lake environment . . . 23
10.2 The portable-kit verifier . . . 23
10.3 The two deliberate rejection controls . . . 24

11 Interpreting the evidence **25**
11.1 The namespace-wide gate . . . 25
11.2 What each layer establishes . . . 26

12 The expanded paper and machine-native crosswalk **26**

13 How to extend the formal development **27**
13.1 Continuing the correspondence work . . . 28
13.2 Troubleshooting a replay . . . 28

14 The complete 35-row contract **29**
14.1 Row 1: Universal one-seed presentation . . . 29
14.2 Row 2: Multi-seed presentation . . . 29
14.3 Row 3: Cyclic characterization . . . 30
14.4 Row 4: Quotient monotonicity . . . 30
14.5 Row 5: Image-algebra invariance . . . 30

14.6 Row 6: Shared direct-sum bound . . . 31
14.7 Row 7: Independent direct-sum formula . . . 31
14.8 Row 8: Tensor-product bound . . . 31
14.9 Row 9: Scalar quotient obstruction . . . 31
14.10 Row 10: Semisimple multiplicity formula . . . 32
14.11 Row 11: Full matrix algebra . . . 32
14.12 Row 12: Multiplicity obstruction . . . 32
14.13 Row 13: Single-operator seed rank . . . 33
14.14 Row 14: Cyclic matrix criterion . . . 33
14.15 Row 15: Exact packet coordinates . . . 33
14.16 Row 16: Packetwise equality . . . 34
14.17 Row 17: Topological packet isomorphism . . . 34
14.18 Row 18: Completion preserves a finite packet split . . . 34
14.19 Row 19: Exact operator reconstruction . . . 35
14.20 Row 20: Packet matrix multiplication . . . 35
14.21 Row 21: Packet powers . . . 35
14.22 Row 22: Adjoint reversal . . . 35
14.23 Row 23: Algebraic full block representation . . . 36
14.24 Row 24: Dimension count . . . 36
14.25 Row 25: Algebraic packet support pattern . . . 36
14.26 Row 26: Reachability obstruction . . . 37
14.27 Row 27: Coordinate transport . . . 37
14.28 Row 28: Naturality of generation and packets . . . 37
14.29 Row 29: Dense realization theorem . . . 38
14.30 Row 30: Typed reachability certificate . . . 38
14.31 Row 31: Bilateral-shift seed . . . 38
14.32 Row 32: Graph seed rank, path, and cycle . . . 39
14.33 Row 33: Orbit-hull algorithm . . . 39
14.34 Row 34: Transfer-block algorithm . . . 40
14.35 Row 35: Exact graph seed-rank algorithm . . . 40

A Direct local imports **41**

B Pinned dependencies **41**

C Guide source, data, and reproducibility **43**

1 Reading the three companions together

The traditional paper develops the mathematical language and its examples. The machine-native paper gives stable records, typed dependencies, semantic profiles, and executable validation interfaces. This guide makes the Lean development usable: where the definitions live, what each theorem actually accepts, how its proof works, and what evidence connects the delivered source to an accepted kernel run.

The historical component taught in Sections 3–14 is the 20-file project preserved at `source/traditional_v1.3.1/formalization/lean/` in the final package. Its formal-component version is 1.0.2. It was originally aligned to the corrected journal manuscript 1.3.0 and retained unchanged in distribution 1.3.1. The traditional final edition also contains an expanded core with 108 stable records. Those records are a different inventory from the 35 Lean contract rows.

An exact scoped result should remain exact when republished. The guide therefore retains both the scope and the exclusions of every contract row. For example, historical row 10 proves a split-semisimple formula after algebra and module equivalences are supplied. The later continuation constructs compatible decompositions and establishes the general division-ring formula. The new result is documented through its own declarations and evidence; it does not change the inputs of historical row 10.

1.1 What the numbers count

Quantity	Meaning
20 source files	Local Lean files, including the import aggregator, focused Mathlib import file, and standalone audit.
35 contract rows	Mathematical statement families and specifications in the normative v1.0.2 map.
84 mapped targets	Distinct Lean declaration names used by those rows. They include definitions and equivalences as well as theorems.
1,126 entries	Project-namespace entries inspected by the historical audit, including generated helpers.
108 stable records	The expanded manuscript inventory. The current correspondence map and the machine payload's historical eight-record map have different evidence scopes.

None of these figures is a count of 1,126 independent new theorems. The useful unit of mathematical coverage is a statement together with its hypotheses, its precise conclusion, its declaration target, and the evidence attached to that target.

1.2 Suggested reading routes

A reader new to Lean can begin with Sections 3–5, which unpack the syntax through exact packet coordinates. A mathematician checking coverage can go directly to the exhaustive map in Section 14, then inspect the relevant source type and proof. A contributor preparing a build should read Sections 10 and 11 before interpreting any successful command.

The displayed Lean listings are excerpts from the preserved source. Their surrounding imports, namespace declarations, and section variables remain in the original files. They are reading excerpts, not advertised standalone files. The editable companion includes their source ranges and hashes in `generated/excerpts.json`.

2 Current release: sources, scope, and evidence

The release preserves one current statement inventory for all three companions. The traditional manuscript supplies the statements and proofs; the machine catalog reproduces those statements with stable identifiers and source locators; the formal correspondence map records the exact Lean support for each statement. The current release files are:

File or directory	Purpose
`formal/current/`	Flat formal source closure, including the continuation modules and their earlier dependencies.
`formal/coverage/COVERAGE_108.json`	All 108 current statements, source proofs and hashes, declaration mappings, scopes, and record classifications.
`formal/evidence/`	Release compiler, replay, audit, identity, and boundary-check receipts, as indexed by the release verification record.
`editable/traditional/`	Current mathematical source, including the final clarification of the split-semisimple corollary.
`source/traditional_v1.3.1/`	Preserved historical source and evidence for the 35-row tutorial contract.

The current inventory has 69 proof-level records. Definitions, classical inputs, boundaries, and documentary records are listed separately in the map and are excluded from this proof-level count.

Status	Covered	Partial	Not mapped
Proof-level records	69	0	0

"Covered" means a complete correspondence at the recorded scope. "Partial" includes substantive missing proofs and consequences not yet assembled from existing declarations. "Not mapped" means that this source closure has no established complete declaration for the record. These statuses are not a percentage of independent theorems: the records differ in scope and may share mathematical dependencies.

2.1 The continuation beyond the historical contract

The accepted Block 08 checkpoint, dated 12 October 2026, contains 54 production modules. It freshly compiled eight additions, authenticated 46 predecessor source/object pairs, and replayed the whole logical closure. Its 1,634 source-owned entries comprise 1,627 logical declarations and seven separately inventoried generated partial runtime helpers. The latter were not proof-replayed. The recorded transitive axiom set is limited to `propext`, `Classical.choice`, and `Quot.sound`. These are the quantities for that checkpoint, not a claim that all 54 source modules were freshly compiled by it.

The continuation proves the intrinsic primary quotient and generator count, the geometric formula, scalar extension, and the six-way cyclic criterion. It also proves noncommutative module-top reduction and the nonsplit semisimple formula, including actual compatible decompositions from the Wedderburn–Artin existence theorem. The final existence theorems do not take a desired rank equality or a compatible decomposition as an input. They are noncomputable mathematical constructions, not executable decomposition algorithms.

Current source layer	Mathematical entry points
`PaperICompletion/`, `PaperICompletion3/`	Spectral certificates, invariant factors, primary seed formulas, and operator-level bridges.
`PaperICompletion4/`	Intrinsic primary decomposition, residue quotients, and the operator seed formula.
`PaperICompletion5/`	Linear primary eigenspaces, the geometric seed formula, and its cyclic criterion.
`PaperICompletion6/`	Scalar extension, one invariant factor, and the six-way cyclic criterion.
`PaperICompletion7/`	Nakayama, module-top reduction, and the local residue division-ring case.
`PaperICompletion8/`	Division-ring row generation, products, regular multiplicities, compatible presentations, and semisimple existence/formulas.

The original `KomorebiVerification/` modules remain the foundation for packet coordinates, transport, seed rank, matrix models, and bounded-packet analysis. Their explanations occupy the following tutorial sections. Current declarations may strengthen those earlier results; the original theorem signatures retain their own input boundaries.

2.2 Release modules and presentation conventions

The `PaperIRelease/` layer extends the preceding continuation. Its source files are grouped by mathematical construction:

Source family	Construction
`GrammarAlgebra`, `GrammarTyped`, `GrammarModules`, `GrammarEquivalence`, `GrammarIdeal`	Typed realizations, module categories, comparison functors, and quotient-algebra presentations.
`SeedOperations`, `ExactSequences`, `ScalarExtension`, `PolynomialModule`, `PolynomialMonic`	Generator bounds, exact sequences, scalar extension, and canonical polynomial-module wrappers.
`PacketAggregation`, `PacketDescent`, `PacketExactSequence`, `PacketUniversal`, `TensorPackets`	Actual range packets, quotient and restriction maps, exactness, universal properties, and tensor constructions.
`AlgebraSupport`, `AlgebraPreorder`, `AlgebraQuotient`, `PacketCorners`, `BoundedStar`	Support algebras, diagonal and quotient structure, local actions, and bounded adjoints.
`GraphReachability`, `GraphSources`, `GraphSourceCount`, `PacketPureRank`, `PacketSourceRank`	Transfer reachability, source components, and packet-pure generator counts.
`TopologySeedRank`, `TopologyPresentations`, `TopologyCountablePackets`, `TopologyCategories`	Closed hulls, locally convex presentations, unconditional packets, and continuous categorical maps.
`CategoryPackets`, `CategoryUniversal`, `CategoryExport`	Packet-system categories, naturality, and preservation of equations under functors.

These are source-navigation entries. The current map supplies the exact declaration names and hypotheses, and the release receipt determines the accepted verification scope.

For the typed-grammar correspondence, the finite-vertex path algebra uses its standard vertex-and-arrow presentation: vertex idempotents are pairwise annihilating and sum to one, and each arrow lies in its target/source corner. The formal `PathAlgebra` names that presentation. `GrammarIdeal` takes an arbitrary two-sided ideal and constructs the comparison between the iterated ideal quotient and the combined presentation; its two comparison maps are proved inverse. Thus the presentation convention is explicit when reading the manuscript notation kQ/I. It is not an assumed module-category equivalence or a supplied conclusion about generators.

2.3 Accepted release verification

The accepted release receipt reports complete correspondences for all 69 proof-level records in the current 108-record inventory. Each conclusion retains the hypotheses and presentation conventions recorded in the correspondence map. Definitions, boundaries, and classical inputs are classified separately. The release includes targeted new semantic reviews for 47 proof-level records and retains accepted predecessor correspondences for 22 others. The total of 69 accepted correspondences is not a claim that every record received a new independent audit during this release.

The release audit checks 116 formal source modules plus the retained boundary suite. Including that suite, it inventories 3513 source-owned entries and checks 3500 logical declarations. Of these, 3499 are checked by logical replay and one declaration sharing an upstream name receives a fresh-name kernel body check after exact type/value comparison. It inventories 13 runtime helpers separately and records 12 rejected negative controls. The allowed transitive axiom set is `Classical.choice`, `Quot.sound`, `propext`. The receipt is `formal/evidence/RELEASE_VERIFICATION.json`; its detailed compiler and logical-replay evidence is under `formal/evidence/whole_closure/`.

Fresh computational checks pass for 96 exact finite or symbolic visual assertions, 33 companion tests, and 10 regenerated PDF/PNG figure pairs. These bounded computations have their own scope and do not replace the formal declarations.

The pinned upstream dependency cache is reused. Logical replay uses Lean's official kernel; it is not an independently implemented checker. The current source-to-manuscript correspondence is a separate reviewed mathematical artifact.

2.4 Current proof-level correspondence index

The following index is generated from the release map. The machine companion supplies the current source statements, and the JSON map supplies every declaration name and exact scope note. Every covered status must be read with that scope note before reusing a theorem.

Stable record	Current status
`PSC-I-T-TYPED-MODULE-EQUIVALENCE`	covered
`PSC-I-P-ACTION-FACTORIZATION`	covered
`PSC-I-P-ALGEBRA-MODULE-GENERATION`	covered
`PSC-I-T-FREE-PRESENTATION`	covered
`PSC-I-C-MINIMAL-FREE-COVER`	covered
`PSC-I-C-ONE-SEED-PRESENTATION`	covered
`PSC-I-P-ANNIHILATOR-TWO-SIDED`	covered
`PSC-I-T-COMPLETED-PRESENTATION`	covered

Stable record	Current status
PSC-I-P-CYCLIC-CHARACTERIZATION	covered
PSC-I-P-ACTING-ALGEBRA-MONOTONICITY	covered
PSC-I-T-QUOTIENT-MONOTONICITY	covered
PSC-I-C-ISOMORPHISM-INVARIANCE	covered
PSC-I-T-EXACT-SEQUENCE-BOUNDS	covered
PSC-I-P-SHARED-DIRECT-SUM	covered
PSC-I-T-INDEPENDENT-DIRECT-SUM	covered
PSC-I-P-TENSOR-BOUND	covered
PSC-I-P-SCALAR-EXTENSION-MONOTONICITY	covered
PSC-I-P-SCALAR-QUOTIENT-WITNESS	covered
PSC-I-T-MODULE-TOP	covered
PSC-I-C-LOCAL-ALGEBRA-FORMULA	covered
PSC-I-L-REGULAR-MODULE-MULTIPLICITIES	covered
PSC-I-T-SEMISIMPLE-SEED-RANK	covered
PSC-I-C-ALGEBRAICALLY-CLOSED-FORMULA	covered
PSC-I-C-FULL-MATRIX	covered
PSC-I-C-MULTIPLICITY-OBSTRUCTION	covered
PSC-I-T-SINGLE-OPERATOR	covered
PSC-I-C-INVARIANT-FACTOR-COUNT	covered
PSC-I-C-ALGEBRAICALLY-CLOSED-SINGLE-OPERATOR	covered
PSC-I-C-CYCLIC-MATRIX	covered
PSC-I-L-PACKET-AGGREGATION	covered
PSC-I-T-EXACT-PACKET-COORDINATES	covered
PSC-I-C-PACKETWISE-EQUALITY	covered
PSC-I-C-PACKET-UNIVERSAL-PROPERTY	covered
PSC-I-T-TENSOR-PACKET-SYSTEM	covered
PSC-I-P-DUAL-PACKET-SYSTEM	covered
PSC-I-T-BANACH-PACKET-ISOMORPHISM	covered
PSC-I-P-CONDITIONED-COORDINATES	covered
PSC-I-C-COMPLETION-PACKET-SPLIT	covered
PSC-I-L-UNCONDITIONAL-COMPLETENESS	covered
PSC-I-T-COUNTABLE-UNCONDITIONAL-PACKETS	covered
PSC-I-T-OPERATOR-RECONSTRUCTION	covered
PSC-I-T-BLOCK-MULTIPLICATION	covered
PSC-I-C-BLOCK-POWERS	covered
PSC-I-C-ADJOINT-REVERSAL	covered
PSC-I-C-TENSOR-TRANSFER-BLOCKS	covered
PSC-I-T-BLOCK-INTERTWINING	covered
PSC-I-T-BLOCK-ALGEBRA	covered
PSC-I-C-PACKET-SUPPORT-PROPAGATION	covered
PSC-I-C-DIMENSION-COUNT	covered
PSC-I-P-BLOCK-DIAGONAL-COMMUTANT	covered
PSC-I-T-SUPPORT-PATTERN-ALGEBRA	covered
PSC-I-C-SUPPORT-PATTERN-STAR	covered
PSC-I-T-PREORDER-ALGEBRA-DECOMPOSITION	covered
PSC-I-T-ALGEBRA-REACHABILITY-SUPPORT	covered
PSC-I-P-REACHABILITY-OBSTRUCTION	covered
PSC-I-P-PACKET-PURE-RANK-BOUNDS	covered
PSC-I-T-SOURCE-COMPONENT-SEED-RANK	covered
PSC-I-C-SOURCE-COMPONENT-COUNT	covered
PSC-I-T-COORDINATE-TRANSPORT	covered

Stable record	Current status
`PSC-I-T-NATURALITY`	covered
`PSC-I-P-SEED-SYSTEM-CATEGORY`	covered
`PSC-I-T-PACKET-EXACT-SEQUENCE`	covered
`PSC-I-T-QUOTIENT-PACKET-DESCENT`	covered
`PSC-I-C-QUOTIENT-PACKET-DESCENT`	covered
`PSC-I-C-TOPOLOGICAL-PACKET-EXACT-SEQUENCE`	covered
`PSC-I-T-DENSE-REALIZATION`	covered
`PSC-I-T-QUOTIENT-TOPOLOGY-COMPLETION`	covered
`PSC-I-T-REALIZATION-EXPORT`	covered
`PSC-I-P-REALIZATION-EXPORT`	covered

3 The type system as mathematical bookkeeping

Lean's types carry assumptions that would otherwise be easy to leave implicit. A statement over a semiring and a module is not automatically a statement about a Hilbert space. A linear equivalence is not automatically a continuous linear equivalence. An endomorphism over the ground field is not automatically linear over a larger acting algebra.

Lean expression	Mathematical role
`[SemiringR]`	An instance supplying the semiring laws for the coefficient object. Square brackets request typeclass synthesis.
`[ModuleRM]`	The scalar action and its laws. The module is over the named scalar object, not an unspecified field.
`M→ₗ[R]N`	An R-linear map, with additive and scalar compatibility bundled into the value.
`Module.EndRM`	The endomorphism type $\mathrm{End}_R(M)$.
`M≃ₗ[R]N`	A linear equivalence with inverse laws.
`X→L[k]Y`	A continuous linear map between normed spaces. The uppercase L is significant.
`X≃L[k]Y`	A continuous linear equivalence.
`M→ₛₗ[σ]N`	A semilinear map along the specified scalar homomorphism σ.
`LinearMap.rangef`	The range as a submodule. An element includes a witness that it is attained by f.
`[Fintypeι]`	A finite enumeration of the index type, allowing finite sums over all indices.
`ι→₀A`	Finitely supported coefficient families, the free A-module on the label type.
`IsLeastSR`	Both $R \in S$ and $R \leq r$ for every $r \in S$. This packages upper and lower bounds.

Curly-braced arguments are implicit parameters inferred from use. Round-bracketed arguments are explicit inputs. A section's `variable` declarations can supply parameters to all later declarations that use them. Therefore a theorem copied without its section context can look less constrained than its actual elaborated type. The preserved `mapped_declarations.log` prints every mapped type with universe information and is the right companion when checking the complete signature.

The package disables automatic implicit-variable invention with `autoImplicit=false` and `relaxedAutoImplicit=false`. This makes a misspelled or undeclared variable an error instead of silently introducing a new parameter. It does not remove explicit hypotheses, nor prove that a chosen formal statement is the intended prose statement.

3.1 Proof terms and the tactics used here

`exacth` closes the current goal with an existing proof of its type. `rfl` proves a definitional equality after reduction. `rw` substitutes proved equalities. `simp` applies a collection of simplification lemmas and produces a proof term. `funext` reduces equality of functions to equality at each input. `Subtype.ext` reduces equality of range elements to equality of their underlying vectors.

The command `classical` makes classical instances available during elaboration. It is not a placeholder. Some resulting declarations depend on `Classical.choice`; the audit records this explicitly. A `noncomputabledef` can define a mathematically precise object through choice or noncomputable library constructions. It does not by itself provide executable numerical code.

These distinctions matter particularly for graph rank. A formula defined using a finite exact spectrum may be mathematically well-defined and formally verified while still not being an extracted eigensolver.

4 The source architecture

The project is organized around a finite algebraic core, seed generation, concrete matrix models, operator and packet completion, and a separate analytic component. `MathlibImports.lean` centralizes the focused external imports. This replaced earlier blanket imports without changing the mathematics of the pre-existing nineteen source files.

Layer	Main responsibility
Projector core	`ProjectorCore`, `ExactCoordinates`, `FibreRefinement`, and `ResidueProjectors`: finite masks, range coordinates, exact reconstruction, and refinement.
Transport and blocks	`PaperITransport` and `PaperIBlockCompletion`: conjugation, naturality, operator-level block identities, and support patterns.
Seed rank	`SeedRankBasics` and `SeedRankOperations`: generation, free presentations, cardinal minima, semilinear transport, products, tensor bounds, and scalar quotients.
Matrix realizations	`SemisimpleSeedRank`, `FullMatrixCyclicity`, `SemisimpleCompletion`: concrete row-space bounds and their connection to canonical seed rank.
One operator	`PathGraphKrylov` and `SingleOperatorGraph`: polynomial orbits, exact minimum certificates, paths, and cycles.
Packet completion	`PacketNaturalityCompletion`: dependent block algebra, support, reachability, strict morphisms, and the typed-quiver certificate.
Analysis	`TopologicalCompletion`: bounded projectors, closure, dense realization, completion, and the bilateral shift.
Models and specifications	`EllipticAlgorithms`: the elliptic quotient model and the three finite mathematical algorithm specifications.

Layer	Main responsibility
Audit	`KernelAudit`: concrete instantiations, the exact mapped-target list, and the namespace-wide axiom gate.

The historical direct-import table is generated from the preserved twenty files in Appendix A. It is an import graph, not a claim that every imported theorem is a logical dependency of every declaration in the importing module. A declaration's transitive axiom set is obtained separately from the Lean environment.

5 Exact packet coordinates, from assumptions to inverse laws

The starting object is a finite family of linear projectors. Its structure records three separate laws: each map is idempotent, different maps annihilate one another, and their finite sum is the identity.

Source: `KomorebiVerification/ProjectorCore.lean`.

```
structure LinearProjectorFamily
    (R : Type u) (M : Type v) (ι : Type w)
    [Semiring R] [AddCommMonoid M] [Module R M] [Fintype ι] where
  proj : ι → Module.End R M
  idempotent : ∀ i x, proj i (proj i x) = proj i x
  orthogonal : ∀ {i j}, i ≠ j → ∀ x, proj i (proj j x) = 0
  complete : ∀ x, (∑ i, proj i x) = x
```

Here orthogonality means pairwise annihilation. An inner product is not among the assumptions, so this word is algebraic at this stage. Self-adjointness appears only in the later adjoint result.

Write P_i for the maps. The intended coordinates are

$$\mathcal{P} = \prod_{i \in I} \operatorname{range}(P_i), \qquad \mathsf{A}(x) = (P_i x)_i, \qquad \mathsf{S}((x_i)_i) = \sum_i x_i.$$

The type of the packet space is crucial. It is the product of independently defined ranges. It is not defined as the image of analysis, and synthesis is the ordinary sum. Thus the reverse inverse law must be proved.

Source: `KomorebiVerification/ExactCoordinates.lean`.

```
/-- The explicit product of the projector ranges. -/
abbrev PacketSpace (P : LinearProjectorFamily R M ι) : Type max v w :=
  (i : ι) → LinearMap.range (P.proj i)

/-- Analysis, with each projected value certified to lie in its projector range. -/
def rangeAnalysis (P : LinearProjectorFamily R M ι) : M →ₗ[R] PacketSpace P where
  toFun x i := ⟨P.proj i x, ⟨x, rfl⟩⟩
  map_add' x y := by
    funext i
    apply Subtype.ext
    exact (P.proj i).map_add x y
  map_smul' c x := by
    funext i
```

```
    apply Subtype.ext
    exact (P.proj i).map_smul c x

/-- Synthesis is the direct finite sum of packet values. -/
def rangeSynthesis (P : LinearProjectorFamily R M ι) : PacketSpace P →ₗ[R] M where
  toFun packets := ∑ i, (packets i : M)
  map_add' packets packets' := by
    simp only [Pi.add_apply, Submodule.coe_add, Finset.sum_add_distrib]
  map_smul' c packets := by
    simp only [Pi.smul_apply, SetLike.val_smul, Finset.smul_sum, RingHom.id_apply]
```

The term `⟨P.projix,⟨x,rfl⟩⟩` has two layers. The outer pair contains the vector and its range-membership proof. The inner pair gives the preimage x and the equality showing that projecting it gives the required vector. The coercion `(packetsi:M)` later forgets this proof component and returns the underlying vector for summation.

5.1 Why the two directions use different hypotheses

Synthesis after analysis is exactly completeness:

$$\mathsf{SA}(x) = \sum_i P_i x = x.$$

Analysis after synthesis uses the range conditions as well. If $x_j \in \mathrm{range}(P_j)$, choose a preimage y_j. Idempotence gives $P_j x_j = x_j$, while pairwise annihilation gives $P_i x_j = 0$ for $i \neq j$. Consequently

$$P_i\Big(\sum_j x_j\Big) = \sum_j P_i x_j = x_i.$$

The source follows this reasoning directly.

Source: `KomorebiVerification/ExactCoordinates.lean`.

```
/-- Completeness proves that synthesizing analyzed coordinates returns the input. -/
@[simp] theorem rangeSynthesis_rangeAnalysis
    (P : LinearProjectorFamily R M ι) (x : M) :
    P.rangeSynthesis (P.rangeAnalysis x) = x := by
  change (∑ i, P.proj i x) = x
  exact P.complete x

/-- Idempotence and orthogonality prove that analysis recovers every range packet. -/
@[simp] theorem rangeAnalysis_rangeSynthesis
    (P : LinearProjectorFamily R M ι) (packets : PacketSpace P) :
    P.rangeAnalysis (P.rangeSynthesis packets) = packets := by
  classical
  funext i
  apply Subtype.ext
  change P.proj i (∑ j, (packets j : M)) = (packets i : M)
  rw [map_sum, Finset.sum_eq_single i]
  · exact P.proj_packet_self packets i
  · intro j _ hji
    exact P.proj_packet_eq_zero packets hji.symm
  · simp
```

After `funexti`, the goal concerns one packet. `Subtype.ext` removes the irrelevant distinction between the proof components of range elements. `map_sum` distributes the linear projector over the finite sum. `Finset.sum_eq_singlei` separates the surviving diagonal term from the annihilated off-diagonal terms. The final `simp` handles the finite-set side condition.

Once the two laws are established, the linear equivalence merely packages them.

Source: `KomorebiVerification/ExactCoordinates.lean`.

```
/-- Exact linear coordinates in the independent product of projector ranges. -/
def exactPacketCoordinates
    (P : LinearProjectorFamily R M ι) : M ≃ₗ[R] PacketSpace P where
  toFun := P.rangeAnalysis
  invFun := P.rangeSynthesis
  left_inv := P.rangeSynthesis_rangeAnalysis
  right_inv := P.rangeAnalysis_rangeSynthesis
  map_add' := (P.rangeAnalysis).map_add
  map_smul' := (P.rangeAnalysis).map_smul
```

There is no inverse selected by choice at this step. Both maps are already explicit. The proof is stronger than recovery of vectors that happen to lie in the image of analysis: it covers every tuple of valid range packets.

5.2 Four coordinates and two parity packets

The standalone audit specializes the construction to $\mathbb{Q}^4$. Fine labels are $0, 1, 2, 3$ and the fine address is the identity. The coarse address is reduction modulo two. For

$$x = (a, b, c, d), \qquad Q_0 x = (a, 0, c, 0), \qquad Q_1 x = (0, b, 0, d),$$

the fine projections isolate one coordinate, and

$$Q_0 = P_0 + P_2, \qquad Q_1 = P_1 + P_3.$$

The general fibre constructor proves its laws from masks, rather than requiring the user to supply them for each instance.

Source: `KomorebiVerification/FibreRefinement.lean`.

```
/-- Every finite address map induces a complete orthogonal projector family. -/
def fibreLinearProjectorFamily
    [Semiring R] [AddCommMonoid M] [Module R M]
    [DecidableEq β] [Fintype β]
    (address : α → β) :
    Komorebi.Foundation.LinearProjectorFamily R (α → M) β where
  proj := fibreLinearProj (R := R) address
  idempotent b x := fibreProj_idempotent address b x
  orthogonal hbc x := fibreProj_orthogonal address hbc x
  complete x := fibreProj_reconstruction address x
```

The audit then proves the parity formula by specializing the independent coarse/fine refinement theorem.

Source: `KomorebiVerification/KernelAudit.lean`.

```
theorem parity_refinement (b : Fin 2) (x : Fin 4 → ℚ) :
    fibreProj parityAddress b x =
      ∑ c, if parityAddress c = b then fibreProj fineAddress c x else 0 := by
  simpa [fineAddress] using
    (fibreProj_refinement fineAddress parityAddress b x)
```

This is an instance of the theorem for an arbitrary input vector, not a comparison of a few sampled rational vectors. The same audit instantiates reconstruction of an arbitrary linear operator on $\mathbb{Q}^4$ from all sixteen fine blocks.

6 Transfer blocks, support, and transport

For source projectors P_i, target projectors Q_j, and a linear map $T : M \to N$, the output-first convention is

$$T_{ji} = Q_j T P_i.$$

The declaration `Komorebi.Foundation.transferBlock` has arguments source family, target family, map, output label, input label. Getting the last two arguments reversed can produce a well-typed but different map when source and target spaces coincide. The guide uses output-first indices throughout.

6.1 Reconstruction and multiplication

Completeness of both families gives

$$T = \sum_j \sum_i Q_j T P_i.$$

For $U : N \to L$ and a third family S_k, completeness of the middle family gives

$$(UT)_{ki} = \sum_j U_{kj} T_{ji}.$$

The earlier core proves pointwise identities. `PaperIBlockCompletion` promotes them to equality of linear maps using extensionality. Contract rows 19 and 20 are classified as stronger than the single-space manuscript instances because the compiled maps may have different source, middle, and target modules.

The block-algebra development then restricts each map to the actual source packet and gives it codomain the actual target packet. This yields a dependent family

$$B_{ji} : \mathrm{range}(P_i) \longrightarrow_R \mathrm{range}(P_j),$$

which is genuinely the full packet-to-packet Hom space. Its multiplication is the finite sum of composable maps. The equivalence with ambient endomorphisms is not a definition of the block space as the image of ambient operators. Both restriction and synthesis are separately constructed and proved inverse.

6.2 Why nonzero packets appear in converse statements

If a support relation is transitive, supported blocks are closed under multiplication. Conversely, to detect a missing compositional edge from universal multiplication closure, one needs blocks whose composite

is nonzero. Contract row 25 works over a field, with every packet range nontrivial and arbitrary packet-Hom blocks available. Those hypotheses provide the witnesses used by the converse.

A zero packet can make a relation edge mathematically invisible. The expanded edition therefore explicitly includes the necessary nonzero-packet assumptions in its corrected $*$-support and source-component-count corollaries. Those editorial corrections do not create new Lean theorems. Row 25 already states its nontriviality scope, and all twenty formal source files remain unchanged.

6.3 A reachability obstruction is a lower-bound certificate

The reachability module first shows that word orbits of packet-supported seeds remain in the support reachable from those seeds. If a nonzero packet is unreachable, the generated algebraic hull is proper. The conclusion depends on the seed support as well as the graph. It is not a general rule saying that the number of graph sources equals unrestricted seed rank.

The typed-quiver model makes the distinction concrete. It uses $U = k^2$, $V = k^3$, the embedding $A(u_1, u_2) = (u_1, u_2, 0)$, and the projection $B(v_1, v_2, v_3) = (v_1, v_2)$. Although $BA = 1_U$, the vector $(0, 0, 1)$ in V is outside range(A). No family supported solely at U generates the whole representation. A source seed and the missing V-direction give the displayed generating pair. Contract row 30 is the certificate for this model, not a classification of arbitrary quivers.

6.4 Coordinate transport and strict naturality

A linear equivalence $C : M \simeq N$ transports projectors by $P_i \mapsto CP_iC^{-1}$ and operators by $T \mapsto CTC^{-1}$. Blocks and their zero/nonzero pattern then transport exactly. The inverse matters: a noninjective realization can kill a nonzero block.

The strict-naturality module also handles semilinear realized actions, with explicit algebra maps and packet-label maps. Its generated-hull statement uses the exact image algebra. Surjectivity gives the appropriate rank inequality; equality and complete edge correspondence require the stronger bijectivity hypotheses recorded in row 28. The injective label map excludes packet merging from that particular strict interface.

7 Generation and seed rank

Seed rank depends on an acting algebra. It is not simply vector-space dimension. One vector generates the natural module of a full matrix algebra, whereas a scalar action on k^n needs n independent generators.

The foundational definitions use the existing span-rank API for the top submodule.

Source: `KomorebiVerification/SeedRankBasics.lean`.

```
variable (A : Type u) (M : Type v)
variable [Semiring A] [AddCommMonoid M] [Module A M]

/-- A set of seeds generates `M` when its `A`-linear span is all of `M`. -/
def Generates (s : Set M) : Prop :=
  Submodule.span A s = ⊤

/-- A family generates when its range is a generating set. -/
```

```
def FamilyGenerates {ι : Type w} (v : ι → M) : Prop :=
  Generates A M (Set.range v)

/-- A module is cyclic when one element generates it. -/
def Cyclic : Prop :=
  ∃ x : M, Generates A M {x}

/--
The seed rank is the least cardinality of a set whose `A`-span is `M`.

Because it is cardinal-valued, it remains meaningful for modules that are not
finitely generated.  In particular, the zero module has seed rank zero.
-/
noncomputable def seedRank : Cardinal :=
  (⊤ : Submodule A M).spanRank

/-- The natural-valued minimum used for finite-generation corollaries. -/
noncomputable def seedFinrank : ℕ :=
  (⊤ : Submodule A M).spanFinrank
```

The cardinal-valued rank makes sense without finite generation. The natural-valued `seedFinrank` is the finite-generation minimum only under the hypotheses used by the finite-rank theorems. The zero module has rank zero. The cyclic/rank-one equivalence therefore includes a nontriviality hypothesis.

7.1 The one-seed presentation

For a seed $s \in M$, the presentation map is $a \mapsto a \cdot s$. Its range is exactly the cyclic submodule. Therefore s generates if and only if this map is surjective.

Source: `KomorebiVerification/SeedRankBasics.lean`.

```
/-- The universal one-seed presentation map, `a ↦ a • x`. -/
def oneSeedMap (x : M) : A →ₗ[A] M :=
  LinearMap.toSpanSingleton A M x

@[simp]
theorem oneSeedMap_apply (x : M) (a : A) :
    oneSeedMap A M x a = a • x := by
  rfl

/-- The image of the one-seed map is exactly the cyclic submodule it spans. -/
theorem range_oneSeedMap (x : M) :
    LinearMap.range (oneSeedMap A M x) = Submodule.span A {x} := by
  exact LinearMap.range_toSpanSingleton x

/-- A seed generates exactly when its presentation map is surjective. -/
theorem generates_singleton_iff_surjective (x : M) :
    Generates A M {x} ↔ Function.Surjective (oneSeedMap A M x) := by
  rw [Generates, ← range_oneSeedMap A M x, LinearMap.range_eq_top]
```

The first isomorphism theorem then supplies the quotient-to-range equivalence.

Source: KomorebiVerification/SeedRankBasics.lean.

```
/-- First-isomorphism presentation for one seed, with its exact range. -/
noncomputable def oneSeedQuotientEquivRange (x : M) :
    (A ⧸ LinearMap.ker (oneSeedMap A M x)) ≃ₗ[A]
      LinearMap.range (oneSeedMap A M x) :=
  (oneSeedMap A M x).quotKerEquivRange

/--
The range-valued quotient equivalence is the canonical presentation map:
the class of `a` is sent to the range element represented by `a • x`.
This is the explicit glue between the first-isomorphism construction and the
formula used in the manuscript.
-/
@[simp]
theorem oneSeedQuotientEquivRange_apply_mk (x : M) (a : A) :
    (oneSeedQuotientEquivRange A M x) (Submodule.Quotient.mk a) =
      ⟨a • x, by
        exact ⟨a, rfl⟩⟩ := by
  rfl
```

The displayed application theorem is an important connection to the paper: the equivalence sends the class of a to the actual vector $a \cdot s$. It is not merely an unspecified isomorphism between abstract objects with the right dimensions.

For a noncommutative acting algebra, the kernel here is a left submodule of the regular module. The theorem does not promote it to a two-sided ideal. Likewise, no norm or bounded completion is manufactured by this algebraic construction. The analytic completion result needs a supplied continuous isometry with dense range.

For several seeds, `Finsupp.linearCombination` evaluates a finitely supported coefficient family. This works for arbitrary label types. The finite-minimum theorem states that, when the module is finitely generated, its seed finrank is the least n admitting a surjection from $\mathtt{Finn}\to_0\mathtt{A}$.

7.2 Surjections, image algebras, and two product actions

Under a surjective semilinear map $f : M \to N$ along $\sigma : A \to B$, an A-generating set maps to a B-generating set. Coefficients in the image of σ are already available in B, so the rank inequality does not require σ itself to be onto. An equality statement requires more: either the scalar map is surjective or the target action is literally restricted to the image algebra.

For products there are two different actions. A shared algebra acts diagonally on $M \times N$, producing an upper bound by the sum of ranks. Independent actions use $A \times B$ and yield the maximum of the two ranks. Coordinate idempotents in $A \times B$ separate the two components, which is precisely what a general shared action may fail to do.

The tensor theorem proves an upper bound under its stated commuting-action and scalar-tower laws. The scalar-quotient theorem gives a lower bound when the action on a finite-dimensional quotient factors through a character. A dimension bound for an arbitrary quotient would be a different statement.

7.3 A complete constructive example: full matrix cyclicity

Suppose $x \neq 0$ in k^n. Choose a coordinate j with $x_j \neq 0$. For any target vector y, make a matrix with only column j nonzero and entries $a_{ij} = y_i/x_j$. Then $Ax = y$. The source is short enough to read in full.

Source: `KomorebiVerification/FullMatrixCyclicity.lean`.

```
/-- A vector is cyclic for the natural action of the full matrix algebra. -/
def IsCyclic (x : ι → K) : Prop :=
  ∀ y : ι → K, ∃ A : Matrix ι ι K, Matrix.mulVec A x = y

/-- Every nonzero vector is cyclic under the full matrix algebra. -/
theorem isCyclic_of_ne_zero {x : ι → K} (hx : x ≠ 0) : IsCyclic x := by
  obtain ⟨j, hj⟩ : ∃ j, x j ≠ 0 := by
    by_contra h
    apply hx
    funext i
    by_contra hi
    exact h ⟨i, hi⟩
  intro y
  let A : Matrix ι ι K := fun i k ↦ if k = j then y i / x j else 0
  refine ⟨A, ?_⟩
  funext i
  simp [A, Matrix.mulVec, dotProduct, hj]
```

The initial contradiction argument finds the nonzero coordinate. The matrix is then explicitly defined and supplied as the witness to the existential goal. Function extensionality reduces $Ax = y$ to its coordinates; simplification evaluates the finite sum and cancels the nonzero denominator. Later declarations connect this concrete cyclicity predicate to `Generates` and canonical seed rank. Row 11 records the result for $n > 0$.

8 Certificate inputs and what their proofs establish

Several important statements are deliberately certificate-explicit. Their hypotheses encode mathematical data that a separate construction or classical theorem would supply. The Lean proof checks the implication from those data. The certificate is not a project axiom, and it is not a proof of its own existence from bare input.

8.1 The split-semisimple formula

The concrete block module consists of $d \times m$ matrices under left multiplication by $M_d(k)$. Each seed supplies at most d independent row directions. The exact count is

$$\left\lceil \frac{m}{d} \right\rceil \qquad (d > 0).$$

For a finite product of blocks, independent actions combine by the maximum. `SemisimpleCompletion` connects these row-space arguments to `seedRank`. The abstract theorem consumes an algebra equivalence to the product of full matrix algebras and a compatible semilinear module equivalence to the product of matrix modules.

The output is a genuine rank formula for the supplied realization. This preserved 20-file component does not derive a Wedderburn decomposition from an arbitrary semisimple algebra, nor the nonsplit division-algebra form. The later continuation supplies that existence integration and the nonsplit formula, as described in Section 2. A reader applying historical row 10 should still identify its supplied equivalences before quoting its conclusion.

8.2 Single-operator seed number: the upper and lower halves

For an operator T, the formalization first works with polynomial orbits $p(T)v$. It then proves a bridge to the genuine image algebra $k[T] = \text{Algebra.adjoin}_k\{T\}$. This prevents an informal change of acting algebra from being hidden in the seed-number notation.

The primary-decomposition theorem uses an attained bound R on block multiplicities. For each seed index $a < R$, it adds the ath block generator from each primary packet, inserting zero when that packet has fewer blocks. Polynomial extraction certificates recover the individual block generators from these padded sums. Thus R seeds suffice. A surjective quotient on which T acts as a scalar gives the opposite bound.

Source: `KomorebiVerification/SingleOperatorGraph.lean`.

```
theorem singleOperator_seedNumber_isLeast_of_primary_decomposition
    (T : Module.End k X) (R : ℕ)
    (multiplicity : Λ → ℕ) (eigenvalue : Λ → k)
    (blockGenerator : ∀ eigen : Λ, Fin (multiplicity eigen) → X)
    (hbound : ∀ eigen, multiplicity eigen ≤ R)
    (maxEigen : Λ) (hmax : multiplicity maxEigen = R)
    (hblocks : FamilyGenerates T
      (fun z : Σ eigen : Λ, Fin (multiplicity eigen) =>
        blockGenerator z.1 z.2))
    (hextract : ∀ (eigen : Λ) (j : Fin (multiplicity eigen)),
      ∃ p : k[X],
        aeval T p
          (paddedBlockSum R multiplicity blockGenerator
            ⟨j.val, lt_of_lt_of_le j.isLt (hbound eigen)⟩) =
          blockGenerator eigen j)
    (q : ∀ eigen : Λ, X →ₗ[k] (Fin (multiplicity eigen) → k))
    (hq : ∀ eigen, Function.Surjective (q eigen))
    (hqT : ∀ (eigen : Λ) (p : k[X]) (x : X),
      q eigen (aeval T p x) = p.eval (eigenvalue eigen) • q eigen x) :
    IsLeast {r : ℕ | SeedNumberLE T r} R := by
  constructor
  · exact ⟨paddedBlockSum R multiplicity blockGenerator,
      paddedBlockSums_generate T R multiplicity blockGenerator hbound hblocks hextract⟩
  · intro r hr
    obtain ⟨v, hv⟩ := hr
    have hle := multiplicity_le_of_generates T v (multiplicity maxEigen)
      (q maxEigen) (hq maxEigen) (eigenvalue maxEigen) (hqT maxEigen) hv
    simpa [hmax] using hle
```

The signature has three groups of data. First, `hbound` and `hmax` make R an attained maximum. Second, `hblocks` and `hextract` supply the orbit-generation and extraction information for the upper bound. Third, `q`, `hq`, and `hqT` supply the scalar quotient and polynomial intertwining law for the lower bound.

The proof constructs an `IsLeast` pair. Its first component is the padded generating family. Its second component takes an arbitrary generating family and uses the quotient at `maxEigen` to force its cardinality to be at least R. The hard distinction is visible in the type: the existence of these quotient and extraction certificates is an input. The file does not independently construct general Jordan form or Chinese-remainder primary projectors from a bare matrix.

8.3 The cyclic-matrix bridge

Row 14 derives the four-way cyclicity statement after two canonical-form bridges have been supplied: geometric multiplicity agrees with block multiplicity, and equality of minimal and characteristic polynomials is equivalent to one block per eigenvalue. Lean verifies their use in the equivalence chain. It does not reconstruct those bridges from arbitrary matrix data in this component.

The row is marked stronger because its compiled implication omits a separately inherited finite-label hypothesis from the manuscript. Stronger refers to this precise change in the typed statement. It does not mean that the supplied canonical-form hypotheses disappeared.

8.4 Graphs, paths, and cycles

The general graph formula takes an eigenbasis grouped by multiplicity, an attained maximum, and polynomial projectors or distinct eigenvalue labels. It does not infer an eigenbasis from the word "symmetric." The source also proves two concrete families independently: a path endpoint generates over any field, and a complex simple cycle has minimum two.

For the path, the endpoint Krylov matrix has zero entries below its diagonal and ones on the diagonal: a walk of length r cannot reach a vertex farther than r, and exactly one length-r walk reaches vertex r. Hence its determinant is one. This yields cyclicity without a numerical eigendecomposition.

For cycles, two adjacent delta vectors generate by the adjacency recurrence. Over $\mathbb{C}$, two distinct Fourier characters with the same adjacency eigenvalue provide a two-dimensional scalar quotient, supplying the lower bound. The resulting minimum-two theorem applies to the simple cycle on $n + 3$ vertices.

9 Topology and mathematical algorithms

9.1 Bounded packets and closed-hull obstructions

The analytic structure replaces linear maps by continuous linear maps. Bounded idempotents have closed ranges. Finite analysis and synthesis become continuous linear maps and then a continuous linear equivalence. The algebraic inverse-law pattern remains visible, while continuity is carried in the types.

For a dense submodule $M \subseteq X$ and a bounded idempotent P, the closure of $P(M)$ is the full range of P. The theorem does not require that M be invariant under P. This is why the source map records the exact scope rather than summarizing the statement only as "completion preserves packets."

The topological obstruction is especially transparent:

Source: `KomorebiVerification/TopologicalCompletion.lean`.

```
/-- If a continuous packet projector annihilates an algebraic generated
hull, then it also annihilates its closed topological hull. -/
theorem closure_le_ker_of_le_ker
```

```
    (Q : X →L[𝕜] X) (H : Submodule 𝕜 X) (hH : H ≤ Q.ker) :
    H.topologicalClosure ≤ Q.ker :=
  Submodule.topologicalClosure_minimal H hH (ContinuousLinearMap.isClosed_ker Q)

/-- A nonzero unreachable packet prevents topological generation. -/
theorem topological_reachability_obstruction
    (Q : X →L[𝕜] X) (H : Submodule 𝕜 X)
    (hQ : Q ≠ 0) (hH : H ≤ Q.ker) :
    H.topologicalClosure ≠ ⊤ := by
  intro htop
  apply hQ
  apply ContinuousLinearMap.ext
  intro x
  have hx : x ∈ H.topologicalClosure := by rw [htop]; trivial
  exact (closure_le_ker_of_le_ker Q H hH hx : Q x = 0)
```

The kernel of a continuous linear map is closed. If an algebraic hull lies in that kernel, so does its closure. If the map is nonzero, the closure cannot be the whole space. This analytic theorem consumes the kernel inclusion; it does not independently derive graph reachability. The algebraic and analytic arguments join at that explicit inclusion.

9.2 Dense realizations and completed presentations

A continuous surjection intertwining the stated operator families sends a dense source hull to a dense target hull. Dense range already suffices for that density conclusion. Exact equality of a source packet's image with a target packet requires the surjectivity used by the packet-range theorem. Neither conclusion needs an unspoken open-mapping argument.

For completed free presentations, the supplied map is a continuous linear isometry with dense range into a complete target. Its completion extension is an isometry with closed and dense range, hence surjective. This gives a continuous linear equivalence. Further theorems prove intertwining of supplied bounded operators and uniqueness of the continuous extension.

The bilateral-shift result uses an actual reindexing representation on $\ell^2(\mathbb{Z})$, not an assumed abstract shift action. Its translates of δ_0 recover all delta vectors, whose linear span is dense. The final statement says that one is the least finite seed cardinality for dense generation by this representation. It does not establish an unrestricted cardinal-valued Laurent-algebra rank.

9.3 Orbit closure: why termination is not code extraction

The finite algorithm specification begins with the span of the seeds and repeatedly adds the images under the input operators:

$$W_0 = \operatorname{span}(S), \qquad W_{n+1} = W_n + \sum_i T_i(W_n).$$

Each iterate lies in the least invariant hull. If two successive iterates agree, the stopped subspace is invariant and contains the seeds, so it equals that hull. Finite dimensionality makes the ascending chain eventually constant.

Source: `KomorebiVerification/EllipticAlgorithms.lean`.

```
/-- A stopped closure pass is exactly the operator hull. -/
theorem orbitClosureIter_eq_orbitHull_of_stop
    (operators : ι → Module.End k V) (seeds : Set V) (n : ℕ)
    (hStop : orbitClosureIter operators seeds (n + 1) =
      orbitClosureIter operators seeds n) :
    orbitClosureIter operators seeds n = orbitHull operators seeds := by
  apply le_antisymm
  · exact orbitClosureIter_le_orbitHull operators seeds n
  · apply orbitHull_le operators seeds
    · exact seeds_subset_orbitClosureIter operators seeds n
    · apply invariant_of_orbitClosureStep_eq operators
      simpa [orbitClosureIter] using hStop

/--
In finite dimension the ascending closure algorithm stops after finitely many
passes, and its stopped output is exactly the least invariant seed hull.
-/
theorem orbitHull_algorithm_terminates_and_correct
    [FiniteDimensional k V]
    (operators : ι → Module.End k V) (seeds : Set V) :
    ∃ n : ℕ,
      orbitClosureIter operators seeds (n + 1) =
        orbitClosureIter operators seeds n ∧
      orbitClosureIter operators seeds n = orbitHull operators seeds := by
  have hEventually := eventuallyConst_of_isNoetherian
    (orbitClosureChain operators seeds)
  rw [Filter.eventuallyConst_atTop] at hEventually
  obtain ⟨n, hn⟩ := hEventually
  have hStop : orbitClosureIter operators seeds (n + 1) =
      orbitClosureIter operators seeds n := by
    exact hn (n + 1) (Nat.le_succ n)
  exact ⟨n, hStop,
    orbitClosureIter_eq_orbitHull_of_stop operators seeds n hStop⟩
```

The proof's termination argument uses the Noetherian property of the submodule lattice. It establishes existence of a stopping index and correctness of the stopped subspace. It does not extract a basis-maintenance loop, a decidable rank test, or a complexity bound. Those belong to an implementation refinement that would require its own proof.

The transfer-block specification is similarly exact at the mathematical level: its table consists of the blocks, summing it gives the original map, and its support predicate is block nonvanishing. It does not verify floating-point tolerances or a sparse edge-list generator.

The exact graph-rank specification takes the finite-spectrum maximum of exact eigenspace dimensions. Its correctness theorem consumes an eigenbasis, projector-polynomial data, spectrum coverage, eigenspace dimensions, and an attained maximum. Its definition is marked `noncomputable`. A Python experiment and this specification can support the same formula while remaining different forms of evidence.

9.4 The elliptic example and scalar discipline

The same source module includes a concrete algebraic model for

$$k[x, y]/(y^2 - (x^3 - x)).$$

It constructs quotient and quadratic normal-form realizations, explicit two-coefficient packet coordinates, and multiplication operators. It proves the relation $M_y^2 = M_x^3 - M_x$ and one-seed generation of the regular module.

The coefficient projectors are linear over the ground field, but not over the full coordinate algebra. The file contains explicit witnesses to that failure. This is an instructive type boundary: the acting algebra for generation may be larger than the scalar algebra over which packet maps are linear. These additional checked declarations are part of the namespace audit but are not counted as extra rows in the 35-row contract.

10 Replaying the preserved verification

The source pins Lean to `leanprover/lean4:v4.34.0-rc2`. The exact Mathlib revision is `aeb72f1e9981b6d35b426b16dca7759be3e7ba06`. Every transitive package pin is listed in Appendix B. A matching installed compiler and dependency cache are prerequisites. The final paper package does not itself supply that compiler or the complete cache.

The following commands are quoted from the component's documented workflow, adapted to the final package layout. They were not run as part of authoring this guide. Use a working copy for new output and keep the preserved release intact.

10.1 A normal provisioned Lake environment

From the final package root:

```
cd source/traditional_v1.3.1/formalization/lean
lake --wfail build
lake env lean -DwarningAsError=true \
  KomorebiVerification/KernelAudit.lean
```

The audit is standalone; it is not imported by the root library aggregator. Consequently a successful library build is not the same thing as a successful explicit audit. These two commands also do not perform every identity check and negative control of the full verifier.

Avoid changing the manifest with `lakeupdate` when reproducing the historical dependency state. A newer compiler or newer Mathlib can be useful in a separate porting branch, but its success would be a different verification event with different inputs.

10.2 The portable-kit verifier

For the matching Linux x86-64 kit used by the preserved workflow, begin in the final package root. Set `KIT` to the separately provisioned portable kit and use a new empty output directory.

```
KIT="/path/to/Komorebi_Portable_Lean_4.34.0-rc2"
cd source/traditional_v1.3.1
bash "$KIT/run.sh" lake -d "$KIT/workspace" env python3 \
```

```
  "$PWD/scripts/verify_lean.py" \
  --project "$PWD/formalization/lean" \
  --kit "$KIT" \
  --out "$PWD/verification/lean_recheck"
```

The kit is external to this paper package. The verifier checks both manifest revisions and the actual Git HEAD of every cached dependency. It compares toolchain pins and checks the reported compiler version. These checks prevent a conveniently available but different cache from silently standing in for the documented one.

The verifier discovers all local Lean files, resolves their local import graph, and compiles them in dependency order to a fresh `objects/` directory. It refuses an output directory that already contains local `.olean` files. That restriction makes the local compilation claim stronger than a build that could reuse prior project objects.

Compiler invocations use:

```
lean -j1 -DwarningAsError=true \
  -DautoImplicit=false -DrelaxedAutoImplicit=false \
  -DmaxSynthPendingDepth=3
```

Each invocation has a recorded command, return code, duration, and log hash. The project modules are newly compiled in that workflow, while matching upstream cached dependencies are reused. A report of this local fresh build does not claim that Mathlib or Lean itself was independently rebuilt.

10.3 The two deliberate rejection controls

After compilation, map reconciliation, and the audit, the verifier asks Lean to reject two small examples. The first asserts a false equality and tries to prove it by decision. The second uses a placeholder, which warnings-as-errors must reject.

```
-- FalseEquality.lean: expected rejection
import Mathlib.Data.Nat.Basic
example : (0 : Nat) = 1 := by decide

-- Placeholder.lean: expected rejection
import Mathlib.Data.Nat.Basic
example : (0 : Nat) = 1 := by sorry
```

These files are negative controls under verification directories. They are not part of the main library and are not unfinished proofs in it. A nonzero exit code alone is insufficient for a passing control: the verifier also checks for the intended diagnostic and rejects timeouts, unknown imports, missing objects, and unsupported-option failures. This ensures that a broken compiler setup is not mistaken for logical rejection.

The preserved final run recorded both controls with exit code 1 and the intended diagnostic. The larger archived build-job totals elsewhere in the distribution describe other runs and must not be reused as this run's local-module count.

11 Interpreting the evidence

The preserved successful execution began at 03:34:26 UTC and finished at 03:36:11 UTC on 12 September 2026. Its authoritative machine record is `source/traditional_v1.3.1/verification/lean/fresh_verification.json`. The source hash list, raw logs, and exact command records belong to that event even when the containing package has a later editorial date.

The preserved guide data's static check compares the delivered twenty source hashes against that record, compares all twenty-four execution-log hashes, reconciles the 35 rows and 84 distinct targets with `KernelAudit.lean`, and checks the historical printed axiom-report inventory. These checks passed. They establish identity and internal consistency of the delivered evidence. They do not perform another elaboration or kernel run.

11.1 The namespace-wide gate

The audit enumerates imported declarations owned by the two project namespaces. It rejects unsafe declarations, custom axiom declarations, and opaque project declarations. For every enumerated declaration it collects transitive axioms and accepts only `propext`, `Classical.choice`, and `Quot.sound`.

Source: `KomorebiVerification/KernelAudit.lean`.

```
run_cmd do
  let env ← getEnv
  let mut audited : Nat := 0
  for (name, info) in env.checked.get.constants.toList do
    if Name.isPrefixOf `Komorebi name || Name.isPrefixOf `KomorebiVerification name then
      audited := audited + 1
      if info.isUnsafe then
        throwError "unsafe project declaration: {name}"
      match info with
      | .axiomInfo _ =>
          throwError "project axiom declaration: {name}"
      | .opaqueInfo _ =>
          throwError "opaque project declaration: {name}"
      | _ => pure ()
      let axioms ← Lean.collectAxioms name
      for axiomName in axioms do
        unless axiomName == ``propext || axiomName == ``Classical.choice ||
            axiomName == ``Quot.sound do
          throwError "unexpected axiom {axiomName} in project declaration {name}"
  if audited == 0 then
    throwError "kernel audit found no project declarations"
  logInfo m!"FAIL-CLOSED KERNEL AUDIT PASS: {audited} project declarations; allowed axioms only:
   ↪  propext, Classical.choice, Quot.sound"
```

The zero-entry check matters: an audit that accidentally imports nothing must fail. The final count is an environment-inventory count, not a count of handwritten mathematical theorems. The separate mapped-target pass resolves each of the eighty-four names, prints its type, and checks its axiom report.

The source-policy scanner also rejects a specified list of forbidden constructs after removing nested comments and string contents. This catches prohibited source forms early. Its role is complementary to the environment audit. Neither a string scan nor a metadata label alone replaces kernel acceptance.

11.2 What each layer establishes

Evidence layer	Interpretation
File hash identity	The present bytes equal the bytes identified by a record. This says nothing by itself about the correctness of those bytes.
Compiler acceptance	The declarations elaborate and their proof terms are accepted in the stated environment. The exact hypotheses remain part of the result.
Axiom audit	The enumerated declarations have only the allowed transitive foundational dependencies.
Normative scope map	The editorial bridge between prose claims and particular compiled declarations, including exclusions. This bridge still requires mathematical reading.
Negative controls	The tested setup rejects the selected false proof and placeholder for the intended reasons. This is a useful pipeline check, not a proof of every possible failure mode.
Python tests	Finite implementation behavior checked by those tests. They are not a universal Lean theorem.
Machine record map	An explicit association between selected stable record IDs and attached evidence. A record's existence is not its proof.

The allowlisted principles are standard Lean/Mathlib foundations. The audit does not claim axiom-free mathematics in an absolute sense. It also does not prove historical priority, performance claims, or the absence of defects in the compiler, trusted kernel, or dependency cache. These are the ordinary boundaries of the recorded verification method, not gaps secretly filled by the count of mapped declarations.

12 The expanded paper and machine-native crosswalk

The machine companion carries its own older seven-source-file attached tranche, version 0.2.0, and an explicit eight-record map. Its inherited acceptance date is 2 September 2026. That tranche shares some finite-projector material with the larger twenty-file formal component, but it is a separate packaged evidence scope.

The current machine revision 1.0.1 retains the expanded inventory of 108 stable records while correcting two explicit nonzero-packet hypotheses in the source and corresponding statement records. A metadata repair or statement correction does not automatically enlarge the attached Lean map. The following eight mappings describe the attached machine tranche's own evidence allocation.

Stable machine record	Attached Lean subject
`PSC-I-D-PACKET-SYSTEM`	Finite linear-projector structure.
`PSC-I-D-ANALYSIS-SYNTHESIS`	Range packet space and the two coordinate maps.
`PSC-I-T-EXACT-PACKET-COORDINATES`	Range-restricted linear equivalence.
`PSC-I-C-PACKETWISE-EQUALITY`	Equality detected by the finite complete family.
`PSC-I-D-TRANSFER-BLOCK`	Definition of the typed block.

Stable machine record	Attached Lean subject
`PSC-I-T-OPERATOR-RECONSTRUCTION`	Pointwise finite reconstruction.
`PSC-I-T-BLOCK-MULTIPLICATION`	Pointwise finite composition identity.
`PSC-I-T-SUPPORT-PATTERN-ALGEBRA`	Transitive ordinary-matrix support is multiplicatively closed; not the whole expanded iff criterion.

The larger component can inform a broader semantic crosswalk. Rows 1–2 concern free presentations, rows 3–9 concern seed-rank operations, rows 10–14 concern split-semisimple and operator formulas, rows 15–29 concern packets, blocks, support, and transport. Rows 30–35 concern the supplementary worked models and algorithm specifications. This is a navigation aid. Where the expanded statement adds a division-ring generalization, an infinite family, an existence theorem, or a different categorical interface, the old scoped result must not be silently relabeled as a proof of the expanded statement.

The final package's `COMPANION_CROSSWALK.json` is the series-level navigation record, and `formal/coverage/COVERAGE_108.json` is the current formal correspondence map. The machine-readable `generated/theorem_map.json` accompanying this guide preserves all thirty-five contract scopes and their exact declaration lists. Unmapped expanded statements remain unmapped until an explicit type-level comparison and accepted proof support a new mapping.

13 How to extend the formal development

A useful extension adds a theorem with a clear source statement, a type that exposes its genuine inputs, and evidence that can be replayed. The easiest way to lose that clarity is to hide the desired conclusion inside an unnamed certificate and then announce that the original unconditional theorem has been formalized.

1. Identify the exact new statement and its relation to the current contract. Decide whether it is a new row, a stronger replacement, or a helper used by an existing row.
2. Inspect the current types before designing another abstraction. For finite packets, reuse range subtypes and actual Hom blocks. For topology, use continuous linear maps and state completeness where it is needed.
3. Separate supplied data from proved consequences. A new canonical-form constructor should have its own existence theorem, not only a theorem taking a canonical-form certificate as an argument.
4. Prove or import the bridge from the new representation to the manuscript's object. A result about a bespoke rank predicate needs a theorem relating it to canonical seed rank before it can replace the existing formula.
5. Compile the changed modules in the pinned environment, update the normative map deliberately, and update the audit inventory to exactly the desired mapped surface.
6. Run the explicit audit and the full verifier with a new output directory. Preserve prior records and record new hashes rather than overwriting the date of the old evidence.
7. Review the new source-to-prose correspondence. Kernel acceptance proves the typed theorem. The review establishes that this is the intended theorem and that its public summary retains the hypotheses.

13.1 Continuing the correspondence work

The continuation has established the primary decomposition and quotient constructions, the geometric formula, scalar extension, the six-way cyclic criterion, module-top reduction, and the nonsplit semisimple formula. Further work is determined by the current coverage map in Section 2; these established results must not be treated as missing certificate constructors. For graph rank, an implementation refinement would relate executable spectral or algebraic data to the exact certificates, with a separate proof of each algorithmic step. For the expanded packet theory, countable unconditional sums require a topology and convergence interface different from the finite `Fintype` API.

These are different tasks from increasing a declaration count. Each closes a named input boundary or extends an exact scope. A completed extension should say which boundary has changed and exhibit the theorem that changed it.

13.2 Troubleshooting a replay

An unknown module or missing object indicates an environment or import-path problem, not a false mathematical theorem. A pin mismatch means the cached dependency state is different from the recorded release. A refusal of nonempty output means the verifier is protecting the local rebuild claim; choose a new output directory. A warning-as-error failure requires reading the diagnostic before deciding whether the source, compiler version, or style policy changed.

A mapped-name failure after a rename is not solved by silently removing the map entry. Either preserve the public declaration name, or update the map and record the new correspondence. A new unexpected axiom requires tracing its transitive dependency before acceptance. A changed source hash is expected after a deliberate edit, but it ends byte-identity with the old accepted source and requires fresh evidence for the changed file.

14 The complete 35-row contract

This section reproduces the normative map's statement titles, declaration names, exact scopes, and exclusions. The numbering belongs to the corrected journal formalization contract. It is not the numbering of the expanded traditional paper or the stable machine-record inventory.

"Exact" means exact at the scope printed in the row. "Stronger" means the mapped declaration removes a specified hypothesis, treats a more general typed setting, or includes an additional endpoint. Rows 10, 13, 14, 32, and 35 are certificate-explicit. The precise certificate boundaries appear both here and in Section 8.

The long qualified names are intentionally retained. They allow a reader or a tool to search the source, issue `#check`, and compare the printed type without guessing a namespace.

14.1 Row 1: Universal one-seed presentation

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankBasics.oneSeedQuotientEquivRange`

`Komorebi.SeedRankBasics.oneSeedQuotientEquivRange_apply_mk`

`Komorebi.SeedRankBasics.generates_singleton_iff_surjective`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.completionEquivOfIsometryDense`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.completionEquivOfIsometryDense_intertwines`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.completionEquivOfIsometryDense_unique`

Exact scope. For the module map `a↦a•s`, the quotient by its kernel is linearly equivalent to its exact range; the glue theorem verifies that the class of `a` maps to `a•s`; and singleton generation is equivalent to surjectivity. If the induced map from the normed quotient is supplied as a continuous linear isometry with dense range into a Banach target, it extends uniquely to a continuous linear equivalence from the completion and intertwines supplied bounded operators.

Boundary. No norm, isometry, density, or bounded extension is constructed automatically on the algebraic quotient. The annihilator is only the typed left submodule/left-ideal object; no two-sided ideal claim is made.

14.2 Row 2: Multi-seed presentation

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankBasics.multiSeedQuotientEquivRange`

`Komorebi.SeedRankBasics.multiSeedQuotientEquivRange_apply_mk`

`Komorebi.SeedRankBasics.multiSeedQuotientEquiv`

`Komorebi.SeedRankOperations.seedFinrank_isLeast_free_surjection`

Exact scope. An arbitrary Finsupp-indexed family has the free-module quotient presentation; the glue

theorem verifies that a coefficient family maps to its finite linear combination; and a generating family gives the quotient onto the whole module. For a finite module, `seedFinrank` is the least `n` admitting a surjection from $\texttt{Finn}\to_0\texttt{A}$.

Boundary. The natural-valued minimum is asserted only under module finiteness. No finite basis or executable presentation matrix is produced.

14.3 Row 3: Cyclic characterization

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.cyclic_iff_seedRank_eq_one`

Exact scope. For every nontrivial module over a semiring, unrestricted cardinal-valued seed rank equals one exactly when a cyclic vector exists.

Boundary. The zero module is excluded here and is governed separately by `seedRank_eq_zero_iff`.

14.4 Row 4: Quotient monotonicity

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.seedRank_le_of_surjective_semilinear`

Exact scope. For an arbitrary ring homomorphism `σ:A→+*B`, a surjective `σ`-semilinear map gives `seedRankBN≤seedRankAM`; `σ` need not be onto.

Boundary. No equality is claimed from surjectivity of the module map alone.

14.5 Row 5: Image-algebra invariance

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.seedRank_imageAlgebra_eq`

Exact scope. For a `k`-algebra homomorphism $\texttt{φ:A}\to_{\text{a}}\texttt{[k]B}$ and a bijective `φ`-semilinear map, the target action is restricted literally to `φ.range`, and its seed rank equals the source seed rank.

Boundary. Equality for the full target algebra `B` requires `φ` to be surjective; it is not inferred otherwise.

14.6 Row 6: Shared direct-sum bound

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.seedRank_prod_le_add`

Exact scope. One semiring acting diagonally on `M×N` gives the cardinal inequality `seedRankA(M×N)≤ seedRankAM+seedRankAN`.

Boundary. This is only an upper bound; it does not identify when equality holds.

14.7 Row 7: Independent direct-sum formula

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.seedRank_prod_productAlgebra`

Exact scope. With the coordinatewise action of `A×B` on `M×N`, the cardinal seed rank is the maximum of the two component ranks.

Boundary. This formula is not for a single shared diagonal action.

14.8 Row 8: Tensor-product bound

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.tensorOperator_smul_tmul`

`Komorebi.SeedRankOperations.seedRank_tensorProduct_le_mul`

Exact scope. Over a commutative semiring `k`, with the stated algebra/module, scalar-tower, and commuting-action laws, the action of `A⊗[k]B` on `X⊗[k]Y` satisfies the cardinal product bound and realizes `(a⊗b)(x⊗y)=ax⊗by`.

Boundary. No equality, completed tensor product, or alternative topological tensor norm is claimed.

14.9 Row 9: Scalar quotient obstruction

Contract classification: Exact.

Declaration targets

`Komorebi.SeedRankOperations.finrank_le_seedRank_of_scalar_quotient`

Exact scope. For a field `k`, a character `χ:A→ₐ[k]k`, a finite-dimensional scalar module `Y`, and a surjective `χ`-semilinear map `X→Y`, `finrankkY` is a cardinal lower bound for `seedRankAX`.

Boundary. The target action must factor through the stated character; a general quotient does not yield the dimension bound.

14.10 Row 10: Semisimple multiplicity formula

Contract classification: Exact.

Declaration targets

`Komorebi.SemisimpleCompletion.semisimple_seedRank_eq_maxCeilDiv_of_equivalences`

`Komorebi.SemisimpleCompletion.splitProduct_seedRank_eq_maxCeilDiv`

Exact scope. For a finite product, positive block sizes `di`, a supplied algebra equivalence to the product of full matrix algebras, and a compatible supplied semilinear module equivalence to the product of `di×mi` matrix modules, seed rank is `max_i⌈mi/di⌉`.

Boundary. Lean does not prove the existence of the Wedderburn/module decompositions, nor the non-split division-algebra variant.

14.11 Row 11: Full matrix algebra

Contract classification: Exact.

Declaration targets

`Komorebi.SemisimpleCompletion.fullMatrix_every_nonzero_vector_generates`

`Komorebi.SemisimpleCompletion.fullMatrix_seedRank_eq_one`

Exact scope. Over a field and for `n>0`, every nonzero vector in the natural `M_n(k)`-module generates and the canonical cardinal seed rank is one.

Boundary. No statement is made for `n=0` or for a smaller acting subalgebra.

14.12 Row 12: Multiplicity obstruction

Contract classification: Exact.

Declaration targets

`Komorebi.SemisimpleCompletion.multiplicity_obstruction`

`Komorebi.SemisimpleCompletion.multiplicity_seedRank_formula`

Exact scope. For `d>0`, the left action of `M_d(k)` on the `d×m` matrix realization of `m` copies has seed rank `⌈m/d⌉`; every generating `r`-family satisfies `⌈m/d⌉≤r`.

Boundary. No general simple-module formula over a nontrivial endomorphism division algebra is included.

14.13 Row 13: Single-operator seed rank

Contract classification: Exact.

Declaration targets

`Komorebi.SingleOperatorGraph.singleOperator_seedNumber_isLeast_of_primary_decomposition`

`Komorebi.SingleOperatorGraph.singleOperator_operatorAlgebra_seedRanks_eq_of_primary_decomposition`

Exact scope. Certificate-explicit: a finite label family supplies block generators, an attained bound `R`, joint polynomial-orbit generation, polynomial extraction from padded seeds, and surjective scalar quotient maps with the evaluation intertwining law. Then `R` is the least polynomial seed number and both `seed Finrank` and cardinal `seedRank` over the genuine image algebra `k[T]=Algebra.adjoink{T}` equal `R`.

Boundary. Lean does not construct a primary/Jordan decomposition, Chinese-remainder projectors, or identify the supplied multiplicities with `dimker(T-λI)` from bare hypotheses.

14.14 Row 14: Cyclic matrix criterion

Contract classification: Stronger.

Declaration targets

`Komorebi.SingleOperatorGraph.cyclic_matrix_criterion_of_primary_decomposition`

`Komorebi.SingleOperatorGraph.cyclic_matrix_four_way_iff_of_primary_decomposition`

Exact scope. Certificate-explicit and finite-dimensional: from an exact positive seed-number certificate, a supplied equality between geometric and block multiplicities, and the supplied minpoly/charpoly canonical-form bridge, Lean proves the four-way equivalence among cyclicity, all geometric multiplicities one, one block per eigenvalue, and `minpoly=charpoly`. The compiled implication does not require the manuscript's separately inherited finite-label hypothesis.

Boundary. Neither canonical-form bridge is derived by Lean from an arbitrary matrix; the theorem verifies the equivalence after those bridges are supplied.

14.15 Row 15: Exact packet coordinates

Contract classification: Exact.

Declaration targets

`Komorebi.Foundation.LinearProjectorFamily.rangeSynthesis_rangeAnalysis`

`Komorebi.Foundation.LinearProjectorFamily.rangeAnalysis_rangeSynthesis`

`Komorebi.Foundation.LinearProjectorFamily.exactPacketCoordinates`

Exact scope. A finite complete, idempotent, pairwise-annihilating linear projector family gives mutually inverse analysis and finite-sum synthesis maps between the module and the independently defined dependent product of projector ranges.

Boundary. No infinite summation or topological convergence assertion is present in this algebraic declaration.

14.16 Row 16: Packetwise equality

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.Packets.eq_iff_projectors_eq`

Exact scope. For a finite linear projector family, `x=y` iff every projected packet coordinate agrees.

Boundary. Finiteness and completeness of the packet family remain essential hypotheses.

14.17 Row 17: Topological packet isomorphism

Contract classification: Exact.

Declaration targets

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.topological_packet_isomorphism`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.packetEquiv`

Exact scope. For a finite bounded projector family on a Banach space over a nontrivially normed field, all packet ranges are closed and analysis/synthesis form a continuous linear equivalence with the canonical finite dependent-product norm.

Boundary. No infinite packet family, unbounded projector, or assertion about every arbitrarily chosen equivalent product norm is included.

14.18 Row 18: Completion preserves a finite packet split

Contract classification: Exact.

Declaration targets

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.idempotentRange_eq_closure_map`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.packetRange_eq_closure_map`

Exact scope. If `M` is a dense submodule of a normed space and `P_i` is any bounded idempotent, the closure of `P_i(M)` equals the full range `P_i(X)`. The packet-family theorem is the direct specialization. No invariance of `M`, surrounding finite packet family, or completeness of `X` is required by the manuscript or the compiled declaration.

Boundary. Boundedness is required; no unbounded-projector or infinite conditional-sum claim is made.

14.19 Row 19: Exact operator reconstruction

Contract classification: Stronger.

Declaration targets

`Komorebi.Foundation.transferBlock_reconstruction_linearMap`

`Komorebi.Foundation.transferBlock_reconstruction_end`

Exact scope. The sum of all transfer blocks reconstructs an arbitrary linear map between modules carrying possibly different finite projector families; the endomorphism formula is the manuscript instance.

Boundary. This is algebraic finite reconstruction, not convergence of an infinite block expansion.

14.20 Row 20: Packet matrix multiplication

Contract classification: Stronger.

Declaration targets

`Komorebi.Foundation.transferBlock_comp_linearMap`

`Komorebi.Foundation.transferBlock_comp_end`

`Komorebi.Foundation.end_comp_eq_sum_transferBlock_comps`

Exact scope. The block of a composite is the finite sum of composite blocks, even for three different source/intermediate/target modules and projector families; the single-space operator identity is the manuscript instance.

Boundary. No infinite matrix multiplication or operator-norm convergence claim is made.

14.21 Row 21: Packet powers

Contract classification: Stronger.

Declaration targets

`Komorebi.PacketNaturalityCompletion.Packets.transferBlock_pow_eq_packetPathSum`

Exact scope. For every natural power `m`, including `m=0`, the transfer block of `T^m` equals the recursively defined finite sum over length-`m` packet paths.

Boundary. Only finite powers and finite packet sets are covered.

14.22 Row 22: Adjoint reversal

Contract classification: Stronger.

Declaration targets

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.adjoint_transferBlock`

Exact scope. On a complete real or complex inner-product space, bounded self-adjoint maps satisfy `(PβTPα)†=PαT†Pβ`. This is stronger than the packet-projector formulation because no idempotence, mutual annihilation, or completeness of a projector family is used.

Boundary. Self-adjointness of both outer maps and boundedness/completeness needed by the adjoint API are explicit.

14.23 Row 23: Algebraic full block representation

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.FullBlocks.fullBlockAlgEquiv`

`Komorebi.PacketNaturalityCompletion.FullBlocks.restrictBlocks_synthesizeBlocks`

`Komorebi.PacketNaturalityCompletion.FullBlocks.synthesizeBlocks_restrictBlocks`

`Komorebi.PacketNaturalityCompletion.FullBlocks.packetBlockAlgebra_mul_eq_blockMul`

Exact scope. A finite projector family over a commutative semiring identifies ambient endomorphisms with the genuine dependent family of all packet-to-packet linear maps; transported multiplication is exactly the output-first block product.

Boundary. This is the algebraic endomorphism algebra. Arbitrary algebraic blocks are not asserted to be bounded in a Banach realization.

14.24 Row 24: Dimension count

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.FullBlocks.end_finrank_eq_packet_sum_eq_square`

`Komorebi.PacketNaturalityCompletion.FullBlocks.finrank_eq_sum_packet_finrank`

Exact scope. For a finite-dimensional vector space over a field, endomorphism dimension equals the double sum of packet-dimension products, which equals the square of the packet-dimension sum.

Boundary. No dimension statement for infinite-dimensional or topological dimensions is made.

14.25 Row 25: Algebraic packet support pattern

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.FullBlocks.identity_blocks_supported_iff_reflexive`

`Komorebi.PacketNaturalityCompletion.FullBlocks.blockMulClosed_iff_transitive`

Exact scope. Over a field, with every packet range nontrivial and arbitrary packet-Hom blocks allowed, the identity is supported exactly for reflexive relations and universal multiplication closure holds exactly for transitive relations.

Boundary. The converse is not promoted to arbitrary bounded/continuous categories without enough nonzero composable maps.

14.26 Row 26: Reachability obstruction

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.Reachability.algebraicOrbitHull_le_reachable_support`

`Komorebi.PacketNaturalityCompletion.Reachability.unreachable_nonzero_packet_obstruction`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.closure_le_ker_of_le_ker`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.topological_reachability_obstruction`

Exact scope. Word orbits from packet-supported seeds stay in the reflexive-transitive reachable support, and a nontrivial unreachable packet makes the algebraic hull proper. Separately, if the algebraic hull lies in the kernel of a nonzero bounded projector, its closure also lies there and is not dense.

Boundary. The analytic theorem consumes the algebraically proved kernel inclusion; it does not infer graph reachability on its own. No general source-count lower bound is claimed for mixed-packet seeds.

14.27 Row 27: Coordinate transport

Contract classification: Exact.

Declaration targets

`Komorebi.Foundation.LinearProjectorFamily.conjugate`

`Komorebi.Foundation.conjugate_transferBlock`

`Komorebi.PacketNaturalityCompletion.Reachability.conjugate_transferEdge_iff`

`Komorebi.PacketNaturalityCompletion.Reachability.conjugate_reachableFrom_iff`

Exact scope. A linear equivalence transports a finite projector family and conjugates every transfer block. For the conjugated operator family, every zero/nonzero edge and every reflexive-transitive reachability statement is preserved.

Boundary. The full invariance assertion requires invertible coordinate transport, not an arbitrary non-injective map.

14.28 Row 28: Naturality of generation and packets

Contract classification: Stronger.

Declaration targets

`Komorebi.PacketNaturalityCompletion.StrictNaturality.generatedHull_natural`

`Komorebi.PacketNaturalityCompletion.StrictNaturality.strict_naturality_package`

`Komorebi.PacketNaturalityCompletion.StrictNaturality.strict_seedRank_le`

`Komorebi.PacketNaturalityCompletion.StrictNaturality.strict_seedRank_eq_of_bijective`

`Komorebi.PacketNaturalityCompletion.StrictNaturality.full_transfer_edge_zero_pattern_correspondence`

Exact scope. For a strict morphism over a field, generated hulls are natural over the exact image algebra, packet analysis and blocks intertwine, a surjective realization gives the cardinal rank inequality, bijective

realization/algebra maps give equality, and bijective packet transport gives complete edge/zero-pattern correspondence. These conclusions do not require the manuscript's finite-dimensional hypothesis.

Boundary. Packet merging is excluded by the injective label map. Full target-algebra rank equality and full target edge coverage require the stated surjectivity/bijectivity hypotheses.

14.29 Row 29: Dense realization theorem

Contract classification: Exact.

Declaration targets

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.dense_operatorHull_image`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.map_packetRange_eq`

Exact scope. For normed spaces and bounded operator families, a continuous surjection that sends each source operator into the target family and intertwines it sends a dense source operator hull to a dense target hull. A surjective map intertwining bounded projectors maps the source range onto the target range.

Boundary. No algebra-homomorphism law, openness, or completeness is needed or concluded. Surjectivity is sufficient for the stated conclusions; dense range alone suffices for the density conclusion, but not for exact range equality. The algebraic hulls need not be equal as sets. The range equality itself needs only an intertwining surjective map; it does not require idempotence.

14.30 Row 30: Typed reachability certificate

Contract classification: Exact.

Declaration targets

`Komorebi.PacketNaturalityCompletion.TypedQuiver.typed_reachability_certificate`

Exact scope. For the displayed `U=k`^{2}, `V=k`^{3} quiver realization, Lean proves `BA=id`, exhibits the nonzero missing direction in `kerB` but outside `rangeA`, proves every `U`-supported family insufficient, and proves that the displayed two seeds generate.

Boundary. This is a certificate for the stated finite model, not a classification or minimum-seed theorem for arbitrary quivers.

14.31 Row 31: Bilateral-shift seed

Contract classification: Exact.

Declaration targets

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.BilateralShiftRepresentation.concretePower_apply`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.concreteBilateralShift_topologicallyCyclic`

`KomorebiVerification.TopologicalCompletion.BoundedProjectorFamily.concreteBilateralShift_seedCardinality_isLeast`

Exact scope. For the explicit reindexing representation on $\ell^2(\mathbb{Z},\mathbb{K})$, `(S_mx)(n)=x(n-m)`, the orbit of δ_0 is dense and one is the least finite cardinality of a densely generating seed set.

Boundary. This does not define or equate an unrestricted cardinal-valued Laurent-algebra rank, and it does not formalize the later residue-packet discussion.

14.32 Row 32: Graph seed rank, path, and cycle

Contract classification: Exact.

Declaration targets

`Komorebi.SingleOperatorGraph.graph_seedNumber_eq_max_eigenspace_multiplicity`

`Komorebi.SingleOperatorGraph.graph_seedNumber_eq_max_eigenspace_multiplicity_of_distinct`

`Komorebi.SingleOperatorGraph.operatorAlgebra_seedFinrank_eq_of_seedNumber_isLeast`

`Komorebi.SingleOperatorGraph.operatorAlgebra_seedRank_eq_of_seedNumber_isLeast`

`Komorebi.SingleOperatorGraph.path_endpoint_seedNumber_isLeast_one`

`Komorebi.SingleOperatorGraph.complex_cycle_seedNumber_isLeast_two`

Exact scope. Certificate-explicit general formula: a finite eigenbasis grouped by multiplicity, an attained maximum `R`, and either explicit polynomial projectors or distinct eigenvalue labels prove that `R` is the least operator-seed number; the bridge declarations identify it with both canonical ranks over `k[A]`. Independently, every nonempty path (`n+1` vertices) has minimum one from an endpoint over any field, and every complex simple cycle (`n+3` vertices) has minimum two.

Boundary. Lean does not derive the eigenbasis from the bare assertion that a matrix is symmetric, nor identify an arbitrary graph's algebraic multiplicities without supplied spectral data. The unqualified spectral theorem remains cited classical input.

14.33 Row 33: Orbit-hull algorithm

Contract classification: Exact.

Declaration targets

`Komorebi.FiniteAlgorithms.orbitHull_algorithm_terminates_and_correct`

`Komorebi.FiniteAlgorithms.orbitClosureIter_eq_orbitHull_of_stop`

Exact scope. For a finite operator index on a finite-dimensional vector space over a field and any seed set, the ascending subspace iteration stops after finitely many passes, and the stopped subspace equals the least invariant subspace containing the seeds.

Boundary. The proof is a mathematical termination/correctness specification; it does not extract the paper's basis-maintenance/rank-test implementation or prove a complexity bound.

14.34 Row 34: Transfer-block algorithm

Contract classification: Exact.

Declaration targets

`Komorebi.FiniteAlgorithms.transferBlockTable`

`Komorebi.FiniteAlgorithms.assembleTransferBlockTable`

`Komorebi.FiniteAlgorithms.transferBlock_algorithm_correct`

`Komorebi.FiniteAlgorithms.transferGraph_iff_block_ne_zero`

Exact scope. For finite source and target projector families, the table is exactly all transfer blocks, summing it reconstructs the input linear map, and its support predicate is exactly block nonvanishing.

Boundary. No decidable zero-test, sparse edge-list extractor, floating-point tolerance, or runtime bound is verified.

14.35 Row 35: Exact graph seed-rank algorithm

Contract classification: Exact.

Declaration targets

`Komorebi.FiniteAlgorithms.exactGraphSeedRank`

`Komorebi.FiniteAlgorithms.exactGraphSeedRank_isLeast`

Exact scope. For a finite square matrix over a field, supplied exact eigenbasis, polynomial-projector, spectrum-coverage, eigenspace-dimension, and attained-maximum certificates prove that the finite-spectrum maximum computed by `exactGraphSeedRank` is the least operator-seed number.

Boundary. `exactGraphSeedRank` is noncomputable in Lean. No verified eigensolver, symbolic factorization routine, symmetry-to-diagonalization step, code extraction, or performance guarantee is claimed.

A Direct local imports

The prefix `KomorebiVerification.` is omitted from module names in this table. The root aggregator remains `KomorebiVerification`. The table is generated from the actual local import lines. External Mathlib imports are not expanded here.

Module	Direct local imports
`Basic`	`MathlibImports`
`EllipticAlgorithms`	`PaperIBlockCompletion`, `SeedRankBasics`, `SingleOperatorGraph`
`ExactCoordinates`	`ProjectorCore`
`FibreRefinement`	`ProjectorCore`
`FullMatrixCyclicity`	`MathlibImports`
`KernelAudit`	`Basic`, `ExactCoordinates`, `FibreRefinement`, `ResidueProjectors`, `PaperITransport`, `PaperIBlockCompletion`, `SeedRankBasics`, `SemisimpleSeedRank`, `FullMatrixCyclicity`, `PathGraphKrylov`, `SeedRankOperations`, `SemisimpleCompletion`, `SingleOperatorGraph`, `PacketNaturalityCompletion`, `TopologicalCompletion`, `EllipticAlgorithms`
`MathlibImports`	None; external imports only.
`PacketNaturalityCompletion`	`ExactCoordinates`, `PaperIBlockCompletion`, `PaperITransport`, `SeedRankBasics`
`PaperIBlockCompletion`	`ProjectorCore`
`PaperITransport`	`ProjectorCore`
`PathGraphKrylov`	`MathlibImports`
`ProjectorCore`	`MathlibImports`
`ResidueProjectors`	`FibreRefinement`
`SeedRankBasics`	`MathlibImports`
`SeedRankOperations`	`SeedRankBasics`
`SemisimpleCompletion`	`SeedRankBasics`, `SeedRankOperations`, `SemisimpleSeedRank`, `FullMatrixCyclicity`
`SemisimpleSeedRank`	`MathlibImports`
`SingleOperatorGraph`	`MathlibImports`, `PathGraphKrylov`, `SeedRankBasics`
`TopologicalCompletion`	`MathlibImports`
`KomorebiVerification`	`Basic`, `ProjectorCore`, `FibreRefinement`, `ExactCoordinates`, `ResidueProjectors`, `PaperITransport`, `PaperIBlockCompletion`, `SeedRankBasics`, `SeedRankOperations`, `SemisimpleSeedRank`, `SemisimpleCompletion`, `FullMatrixCyclicity`, `PathGraphKrylov`, `SingleOperatorGraph`, `PacketNaturalityCompletion`, `TopologicalCompletion`, `EllipticAlgorithms`

B Pinned dependencies

The compiler pin is `leanprover/lean4:v4.34.0-rc2`. The preserved accepted run reported compiler commit `6a10ac8c22beadecabdbb0919c2b50214762f91d`. Package revisions below come from the preserved

Lake manifest, not from a live dependency update.

Package	Exact revision
mathlib	`aeb72f1e9981b6d35b426b16dca7759be3e7ba06`
plausible	`d9598f07b1bc701f1e3aae163d2681c1fd978793`
LeanSearchClient	`ba67e212be1197b84c1f1f6299488a10a3002713`
importGraph	`d8823026ac7ef130c253089d95685f9877b95323`
proofwidgets	`a8acbfd87375ff4abe14ce09db5b7664d383bc7f`
aesop	`18889deb9e83ea7420ef51c160d6f88552e744e3`
Qq	`507746ab8f4b643ccdacb2ec4cdb5853fa9f8ab3`
batteries	`4cac2177c37f5530c4da76aa8e4307f3fc9e4dcb`
Cli	`ab3a82db9fea14cf0fd7f5a2de650f4b534640af`

C Guide source, data, and reproducibility

The editable guide directory contains this LaTeX source, exact source excerpts, generated contract and import tables, machine-readable JSON, and `build_guide_data.py`. The historical data generator is read-only with respect to the preserved formal component. It imports the preserved verifier's comment-aware scanning function, compares source and log hashes, reconciles map inventories, and extracts listings. It never invokes Lean.

From `editable/lean_guide/` in the final package:

```
python3 build_guide_data.py \
  --source ../../source/traditional_v1.3.1
xelatex -interaction=nonstopmode -halt-on-error \
  Paper_I_Lean_Guide_Final.tex
xelatex -interaction=nonstopmode -halt-on-error \
  Paper_I_Lean_Guide_Final.tex
xelatex -interaction=nonstopmode -halt-on-error \
  Paper_I_Lean_Guide_Final.tex
```

The current release section is generated separately from the reviewed inventory. From the release root:

```
python3 scripts/build_release_companion_data.py \
  --coverage formal/coverage/COVERAGE_108.json
```

Pass `--metadata` with the companion evidence JSON to reproduce the release verification summaries. This step preserves every coverage classification and never executes Lean. It also regenerates the current machine catalog from the same statements and proofs.

The PDF uses P052, Nimbus Sans, DejaVu Sans Mono, and Latin Modern Math fonts with XeLaTeX. A TeX installation must provide the packages listed in the preamble. Rendering the resulting PDF is a separate presentation check; it is not a Lean verification step.

Companion file	Purpose
`generated/static_identity_audit.json`	Executed static identity checks, including every compared source and log hash.
`generated/theorem_map.json`	All 35 contract rows and 84 target names with their scopes and exclusions.
`generated/local_imports.json`	Direct import graph of the 20 local files.
`generated/excerpts.json`	Source file, line range, full-source hash, and excerpt hash for each displayed listing.
`source/traditional_v1.3.1/verification/lean/`	Preserved successful compiler record and raw logs, relative to the package root.

The final guide was proofread in self-review mode against the preserved source definitions and theorem map. It does not modify a mathematical token in those source files. The guide's explanations, formulas, and scope summaries remain subordinate to the exact formal signatures and the normative exclusions recorded above.