u3lib: An AI-Ready Computational Framework for U(3) Symmetry Algebra, Wigner Coupling, and Racah Recoupling Coefficients

Daniel Langr,[1] Phong Dang,[2] Tomas Dytrych,[3,2] and Michela Negro[2]

[1] *Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic*
[2] *Department of Physics and Astronomy, Louisiana State University, Baton Rouge, Louisiana, USA*
[3] *Nuclear Physics Institute, Czech Academy of Sciences, Řež, Czech Republic*

u3lib is a C++ library for U(3)/SU(3) representation-theory calculations in nuclear structure and related applications. It evaluates Wigner coupling and Racah recoupling coefficients using the procedure of Dang *et al.*, which avoids the binomial coefficients and alternating sums used in earlier formulations. The library also provides irreducible representations (irreps), dimensions, Casimir invariants, conjugation, and tensor-product decompositions with outer multiplicities. It supports the canonical Gelfand-Tsetlin basis and the physical SU(3) ⊃ SO(3) basis, including inner multiplicities and the transformation between the two bases. Generator routines return matrix elements in the canonical basis and reduced matrix elements in the physical basis. Python bindings mirror the C++ API. Domain skills supplied with the repository allow an AI assistant with code-execution access to call these routines from natural-language requests.

PROGRAM SUMMARY

Program title: u3lib
Licensing provisions: MIT License.
Code repository: https://github.com/Deep-SANE/u3lib
Programming language: C++ (C++20), with Python bindings via pybind11.
External libraries: Eigen3 for QR-based null space solver; Boost for hash tables; OpenMP for multi-threading; fmt for formatting output; libwignernj for SU(2) coupling and recoupling coefficients; and KraNS for iterative sparse null-space solvers on large matrices (with an automatic dense-QR fallback for small or block-heavy cases). The Python extension additionally uses pybind11.
Nature of problem: Fast, accurate, and reproducible evaluation of Wigner reduced coupling coefficients for the two group-subgroup chains SU(3) ⊃ SO(3) and U(3) ⊃ U(2), Racah recoupling coefficients (6-U(3) and 9-U(3) coefficients), and the associated algebraic quantities (irrep dimensions, Casimirs, multiplicities, basis enumeration, generator matrix elements, and basis transformations) needed by symmetry-adapted calculations for nuclear structure and similar problems.
Solution method: The U(3) ⊃ U(2) reduced coupling and recoupling coefficients are built from the highest-weight U(3) ⊃ U(2) ⊃ U(1) full coupling coefficients, obtained as the orthonormal null space of the canonical raising generators, and propagated by the corresponding lowering generators [1]; the physical SU(3) ⊃ SO(3) coefficients follow from the physical-to-canonical basis transformation, also obtained as the orthonormal null space of the physical raising generator L_+ [1].

I. INTRODUCTION

Algorithms for computing coupling and recoupling coefficients for the unitary group U(3) and its special unitary subgroup SU(3) date back to two papers published in 1973 by Draayer and Akiyama, which introduced a general algorithm [2] and an associated Fortran program for their evaluation [3]. These coefficients underpin nuclear structure calculations using Elliott's SU(3) model [4, 5] and other symmetry-adapted shell-model approaches [6–8], where large numbers of coefficients make speed and accuracy essential. The same symmetry is used in nuclear-structure frameworks for medium-mass and heavy deformed nuclei, including the pseudo-SU(3) [9, 11, 13], proxy-SU(3) [10, 11], and quasi-SU(3) [12, 13] schemes, as well as the multiconfigurational dynamical symmetry (MUSY) [14]. Over the following five decades, the Draayer and Akiyama library was updated to improve numerical accuracy [15] and implemented in modern programming languages [16, 17]. These implementations retained the underlying algorithm based on hypergeometric functions with large factorials and alternating sums. Dang *et al.* introduced a different approach that does not involve these instability-prone functions by constructing the coefficients from the action of the group's raising generators using numerical null-space solvers [1, 18]. For the large sparse matrices arising in this approach, a dedicated iterative null-space solver, now known as KraNS, was developed [19].

The same algebraic tools also arise outside nuclear physics. In particle physics, flavor SU(3) underlies the eightfold-way classification of hadrons and the quark model [20, 21]. In quantum information, Schur transforms are constructed from successive Clebsch–Gordan transformations for d-level systems; for $d = 3$, this provides a potential application of U(3) coupling data to qutrit basis transformations [22]. In condensed-matter physics, symmetry-adapted numerical renormalization-group calculations of the SU(3) Anderson impurity model require tensor-product decompositions, coupling coefficients, and outer-multiplicity labels [23]. In quantum optics, SU(3) representation theory organizes multiphoton states and interference in three-channel passive interferometers [24]. In quantum simulations of SU(3) lattice gauge theory, basis states are organized by SU(3) irreps and Gelfand–Tsetlin patterns, and the gauge-invariant singlet state at each vertex is assembled from coupling coefficients [25]. These examples suggest uses for the computational framework presented here as a source of representation-theoretic building blocks, with application-specific modeling and any basis conversion supplied by the user.

In this paper, we present `u3lib`, a library for calculations with the U(3) symmetry algebra, including coupling and recoupling coefficients. The library is part of `DeepSANE`—**Deep**-learning and **S**ymmetry-adapted **A**b-initio **N**uclear modeling: an AI-ready **E**cosystem[1]. Alongside the coefficient routines, `u3lib` provides C++ objects for irreps and their Casimir invariants, the canonical Gelfand-Tsetlin U(3) $\supset$ U(2) $\supset$ U(1) and the physical SU(3) $\supset$ SO(3) bases, the generator matrix elements, the tensor-product decomposition, and the physical-to-canonical transformation. The C++ API is header-only; Python bindings mirror it one-to-one.

Large language models can help run scientific calculations from natural-language requests. To support their use with `u3lib`, we provide an agent-skill catalog that guides AI assistants through the library's capabilities. To use these routines correctly, an agent must follow group-theoretical notation, phase conventions, memory-lifetime rules, and thread-safety requirements. The repository-local skills make these requirements explicit alongside the source code and provide procedures for constructing and checking calculations. This approach follows the interest in reusable domain-specific skills for computational research seen in initiatives such as the U.S. Department of Energy (DOE) Genesis Mission and its scientific agent skill catalog[2]. We also evaluate whether this guidance improves task completion and reduces the resources needed to integrate the library, particularly as implementation complexity increases.

The remainder of this paper documents the repository structure (Sec. II), the classes and interface (Sec. III), the Python bindings (Sec. IV), the AI-ready interface and its benchmark evaluation (Sec. V), and the installation (Sec. VI).

II. LIBRARY STRUCTURE

`u3lib` is a header-only C++ library with build targets for examples, tests, and Python bindings. The top-level directories are:

- `include/`: the library itself (all algorithms live here; user programs include these headers directly):
  - `U3irrep.h`: the irrep classes `su3::irrep`, `u3::irrep`;
  - `U3gelfand.h`, `internal/gelfand/`: the Gelfand state class `u3::gelfand`;
  - `U3basis.h`: basis enumeration (`gelfandBasis`, `physicalBasis`), the inner multiplicity `Kappa`, the `u3::physicalvector` class, and the generator-action predicates;
  - `U3decomposition.h`: the outer multiplicity `Rho` and the tensor-product `decomposition`;
  - `U3U2xU3U2.h`, `U3SO3xU3SO3.h`: the canonical and physical coupling coefficients and their look-up tables;
  - `U3_U6Z6Coeffs.h`, `U3_U9Coeffs.h`: the 6-U(3) and 9-U(3) recoupling coefficients and their look-up tables;
  - `U3SO3_to_U3U2U1.h`: the physical-to-canonical transformation;
  - `U3generatorsRME.h`: the physical-generator reduced matrix elements `RME_L`, `RME_Q`;
  - `Definitions.h`: the structured coefficient containers;
  - `u3lib.h`: the umbrella header and the `initialize`/`finalize` lifecycle.
- `src/`: example driver programs (one `.cpp` per task) that call the headers and print results; compiled to `bin/`.

[1] See the `DeepSANE` GitHub organization: https://github.com/Deep-SANE
[2] See the DOE Genesis Mission skill catalog: https://gitlab.osti.gov/genesis/genesis-skills.

Header	Entry point	Task
`U3irrep.h`	`dim`, `C1/C2/C3`, `conjugate`	irrep dimension, Casimirs, conjugation
`U3basis.h`	`Kappa`, `gelfandBasis`, `physicalBasis`	inner multiplicity; basis enumeration
`U3decomposition.h`	`Rho`, `decomposition`	outer multiplicity; irrep product
`U3U2xU3U2.h`	`wigU3U2db`	U(3)⊃U(2) reduced coupling coefficients
`U3SO3xU3SO3.h`	`wigU3SO3db`	SU(3)⊃SO(3) reduced coupling coefficients
`U3_U6Z6Coeffs.h`	`u3u6db`, `u3z6db`	6-U(3) recoupling coefficients (three irreps)
`U3_U9Coeffs.h`	`u3u9db`	9-U(3) recoupling coefficients (four irreps)
`U3SO3_to_U3U2U1.h`	`phys_to_cano`	physical-to-canonical transformation
`U3generatorsRME.h`	`RME_L`, `RME_Q`	reduced matrix elements of **L**, **Q**

TABLE I: Main entry points of `u3lib`.

- `python/`: the pybind11 extension (`pyu3lib.cpp` → `u3lib.so`, built with `make python`) and a tutorial Jupyter notebook (`tutorial.ipynb`).
- `agents/`: the AI-agent skill catalog: an index (`AGENTS.md`), shared runtime instructions, and one skill per capability.
- `test/`: the regression suite (`test_*.cpp`, the `*_cases.h` case sets, and `reference/` documented values), run by `make test`.
- `doc/`: this manual (LaTeX source).
- `bin/`: build output (not source).

III. CLASSES AND LIBRARY INTERFACE

This section describes the public classes, methods, and interfaces for accessing coefficients. Types belong to the `u3` namespace, with SU(3) base types in `su3`. The algebraic types can be used without initialization. Coefficient databases require initialization as described below. Table I lists the main entry points.

A. Lifecycle

In `u3lib.h`:

- `initialize()` / `initialize(&argc,&argv)`: set up the caches of the SU(2) coefficients from `libwignernj`, and the U(3) coefficient databases;
- `finalize()`: destroy the databases;
- `initialize_thread()` / `finalize_thread()`: per-thread setup for multithreaded use;
- `clear()`: clear cached coefficients while keeping the databases available;
- `is_initialized()`: check whether the library is initialized.

Note on shared-backend lifecycle: if `u3lib` runs in the same process as another library that shares the `libwignernj` backend, initialize all of them at startup and finalize only after all coefficient work is complete. Calling `finalize()` while another user of the backend is still active destroys the shared state and causes runtime errors.

B. Irreps: `su3::irrep` and `u3::irrep`

(`U3irrep.h`.) `su3::irrep` stores an SU(3) irrep; `u3::irrep` derives from it and adds the third label n_{33} (fixing the total quanta).

Constructors:

- `su3::irrep()`: default (empty) SU(3) irrep.
- `su3::irrep(lambda,mu)`: SU(3) irrep from its labels.
- `u3::irrep()`: default (empty) U(3) irrep.
- `u3::irrep(n13,n23,n33)`: U(3) irrep from its three labels.
- `u3::irrep(lambda,mu)`: normalized representative $[\lambda{+}\mu, \mu, 0]$.
- `u3::irrep(su3::irrep, N)`: fix the total quanta N.

Label getters:

- `lam()` / `lambda()`: the SU(3) label λ.
- `mu()`: the SU(3) label μ.
- `n13()`, `n23()`, `n33()`: the three U(3) labels (only for `u3::irrep`).

Dimension and invariants:

- `dim()`: the dimension (both classes).
- `C2()`: the quadratic SU(3) Casimir (both classes, shift-invariant).
- `C3()`: the cubic SU(3) Casimir (both classes, shift-invariant).
- `C1()`: the first-order Casimir $= n_{13}{+}n_{23}{+}n_{33}$ (total quanta), on `u3::irrep` only; `su3::irrep` has no `C1()`.
- `quanta()`: synonym of `C1()`.

Other:

- `zero()`: test for the trivial irrep.
- `conjugate()`: the conjugate irrep (μ, λ).
- `conjugate_phase()`: the associated conjugation phase.
- `normalize()`: set $n_{33} = 0$ for `u3::irrep`.
- comparison operators and stream output.
- `writeBinary` / `readBinary`: serialization for binary files.

C. Gelfand states: `u3::gelfand`

(`U3gelfand.h`.) The canonical basis state $|n_{13}, n_{23}, n_{33};\ n_{12}, n_{22};\ n_{11}\rangle$.
Constructor:

- `u3::gelfand(n13,n23,n33,n12,n22,n11)`: build a Gelfand state from its labels.

Label getters and validity check:

- `n13()` ... `n11()`: the six Gelfand-pattern labels.
- `check()`: test the betweenness condition.
- `normalize()`: shift all quantum numbers by n_{33}.

Subgroup labels:

- `lam()` / `lambda()`: the SU(3) label λ.
- `mu()`: the SU(3) label μ.
- `epsilon()`: the U(1) weight.

- `Lambdax2()`: 2× the embedded SU(2) angular momentum.
- `MLambdax2()`: 2× its projection.

Conjugation:

- `conjugate()`: the conjugate Gelfand state.
- `conjugate_phase()`: the associated conjugation phase.

Generator matrix elements: Each returns the matrix element of a U(3) canonical generator E_{ij} acting on a Gelfand state; a `_12`/`_22` suffix selects the target branch (either on n_{12} or n_{22}, see [1]).

- `e11()`, `e22()`, `e33()`: the diagonal weight generators.
- `e12()`: the E_{12} element.
- `e21()`: the E_{21} element.
- `e23_12()`, `e23_22()`: the E_{23} element, per target branch.
- `e32_12()`, `e32_22()`: the E_{32} element, per target branch.
- `e13_12()`, `e13_22()`: the E_{13} element, per target branch.
- `e31_12()`, `e31_22()`: the E_{31} element, per target branch.

The raising-action predicates `E12`, `E23_12`, `E23_22`, `E13_12`, `E13_22` (in `U3basis.h`) identify the target state of each generator.

D. Physical states: `u3::physicalvector`

(`U3basis.h`.) The SU(3) ⊃ SO(3) reduced state $|(\lambda\mu)\,\kappa\,L\rangle$.
Constructor:

- `u3::physicalvector(irrep, L)`: a physical state of angular momentum L.

Accessors:

- `irr()`: the underlying irrep.
- `lam()` / `lambda()`: the SU(3) label λ.
- `mu()`: the SU(3) label μ.
- `n13()`, `n23()`, `n33()`: the U(3) labels.
- `L()`: the angular momentum.
- `kappa()`, `kappamax()`: the inner multiplicity of L.

E. Free functions

Multiplicities:

- `u3::Kappa(irrep,L)`: inner multiplicity $\kappa_{\max}$ (also `kmax`, `mult`).
- `u3::Rho(A,B,C)`: outer multiplicity $\rho_{\max}$ of irrep C in $A \otimes B$ product (`U3decomposition.h`).

Bases:

- `u3::gelfandBasis(irrep)`: all Gelfand states of the irrep.
- `u3::physicalBasis(irrep)`: all physical states with nonzero κ.

- `u3::findGelfand(g,gArr)`: the index of a state within the `gArr` Gelfand basis.

Decomposition:

- `u3::decomposition(A,B)`: the full Kronecker series, returning (list of C, list of $\rho_{\max}$) with $\sum_C \rho_{\max}(C) \dim C = \dim A \dim B$.

Physical generator reduced matrix elements: Each takes two `physicalvector` states (bra, ket) and two `std::vector<int>` out-parameters (which receive the zero-based multiplicity labels), and returns an `Eigen::MatrixXd` of shape $\kappa'_{\max} \times \kappa_{\max}$.

- `RME_L`: the angular-momentum RME, $\langle (\lambda\mu)\kappa' L' \| \mathbf{L} \| (\lambda\mu)\kappa L \rangle = \delta_{L'L}\sqrt{L(L{+}1)(2L{+}1)}$.
- `RME_Q`: the quadrupole RME, carrying an extra $\sqrt{2L'+1}$ (Sec. III H).

Transformation: Both expand a physical state `p` in the Gelfand basis `gArr`, returning the canonical-state indices and an amplitude matrix with one orthonormal column per κ.

- `u3::phys_to_cano(p,M,gArr)`: the expansion at a single projection M.
- `u3::phys_to_cano_allM(p,gArr)`: the expansion over all M.

F. Coefficient containers

(`Definitions.h`.) Every container provides `operator()(...)`, `operator[](i)`, a member `print()` method, and stream insertion via `operator<<`. The indexing operators return a writable reference for a non-const C++ container and a value for a const container. They access the same coefficients using different indexing schemes:

- `block(i0,...,id)` selects a coefficient by its multiplicity indices, in the order listed for each container below. Each index starts at zero and is smaller than its corresponding `*max()` count.
- `block[i]` selects a coefficient by a single flat index, $0 \le i <$ `block.size()`. For axis sizes $d_0, \ldots, d_{m-1}$, the equivalent flat index is $i = \sum_{j=0}^{m-1} i_j \prod_{k=j+1}^{m-1} d_k$, where an empty product is one. Thus, for a four-axis container, $i = ((i_0 d_1 + i_1)d_2 + i_2)d_3 + i_3$.
- `data()` exposes the underlying flat buffer; `size()` gives its number of coefficients; and `empty()` tests whether it contains no coefficients.
- `print(s)` and `s << block`: output a formatted table of all multiplicity indices and their corresponding values to the stream `s` (defaulting to `std::cout` for `print()`).

In the equations below, A, B, C, D denote U(3) irreps, and AB, BC, AC, BD, CD denote selected intermediate irreps in the indicated tensor products, rather than the tensor products themselves. The total irrep is denoted by T for three- and four-irrep recoupling. Each ρ labels a particular occurrence of an output irrep in its coupling channel; κ resolves repeated angular momenta within an SU(3) irrep. The multiplicity indices in these equations use the library's zero-based indexing.

`U3U2`: canonical coupling of $A \otimes B \to C$, reduced with respect to U(2): $\langle Aa; Bb \| Cc \rangle_\rho$, where a, b, c are their fixed U(2) sublabels $[n_{12}, n_{22}]$. Only the outer multiplicity varies within the container, so `block(rho)` and `block[rho]` are identical.

- `rhomax()`: the outer multiplicity $\rho_{\max}$.
- `operator()(rho)`: indexed by ρ.
- `print(s = std::cout)`: prints the coefficient row over ρ.

`U3SO3`: physical coupling of $A \otimes B \to C$, reduced with respect to SO(3): $\langle A\,\kappa_A L_A;\ B\,\kappa_B L_B \| C\,\kappa_C L_C \rangle_\rho$, where L_A, L_B, L_C are fixed angular momenta. The three inner multiplicities and the outer multiplicity are the four container axes.

- `kappaAmax()`: inner multiplicity $\kappa_{A,\max}$.
- `kappaBmax()`: inner multiplicity $\kappa_{B,\max}$.

- `kappaCmax()`: inner multiplicity $\kappa_{C,\max}$.
- `rhomax()`: the outer multiplicity $\rho_{\max}$.
- `operator()(kappaA,kappaB,kappaC,rho)`.
- `print(s = std::cout)`: tabular print over $(\kappa_A, \kappa_B, \kappa_C, \rho)$.

`U3U6`: U-recoupling: $U(A, B, T, C;\ AB, \rho_{AB}, \rho_{AB_C};\ BC, \rho_{BC}, \rho_{A_BC})$. The labels ρ_{AB_C} and ρ_{A_BC} resolve the final couplings $AB \otimes C \to T$ and $A \otimes BC \to T$, respectively.

- `rhoABmax()`: multiplicity of the $A \otimes B \to AB$ coupling.
- `rhoAB_Cmax()`: multiplicity of the $AB \otimes C$ coupling.
- `rhoBCmax()`: multiplicity of the $B \otimes C \to BC$ coupling.
- `rhoA_BCmax()`: multiplicity of the $A \otimes BC$ coupling.
- `operator()(rhoAB,rhoAB_C,rhoBC,rhoA_BC)`.
- `print(s = std::cout)`: tabular print over $(\rho_{AB}, \rho_{AB_C}, \rho_{BC}, \rho_{A_BC})$.

`U3Z6`: Z-recoupling: $Z(B, A, T, C;\ AB, \rho_{AB}, \rho_{AB_C};\ AC, \rho_{AC}, \rho_{AC_B})$. The labels ρ_{AB_C} and ρ_{AC_B} resolve the couplings $AB \otimes C \to T$ and $AC \otimes B \to T$, respectively.

- `rhoABmax()`: multiplicity of the $A \otimes B \to AB$ coupling.
- `rhoAB_Cmax()`: multiplicity of the $AB \otimes C$ coupling.
- `rhoACmax()`: multiplicity of the $A \otimes C \to AC$ channel.
- `rhoAC_Bmax()`: multiplicity of the $AC \otimes B$ channel.
- `operator()(rhoAB,rhoAB_C,rhoAC,rhoAC_B)`.
- `print(s = std::cout)`: tabular print over $(\rho_{AB}, \rho_{AB_C}, \rho_{AC}, \rho_{AC_B})$.

`U3U9`: 9-U(3) recoupling:

$$\begin{bmatrix} A & B & AB & \rho_{AB} \\ C & D & CD & \rho_{CD} \\ AC & BD & T & \rho_{AC_BD} \\ \rho_{AC} & \rho_{BD} & \rho_{AB_CD} & \end{bmatrix}. \tag{1}$$

The rightmost entries label the row couplings, and the bottom entries label the column couplings. In particular, ρ_{AB_CD} labels $AB \otimes CD \to T$ and is named `rhoABCD` in the code; ρ_{AC_BD} labels $AC \otimes BD \to T$ and is named `rhoACBD`. The container's axis order is $(\rho_{AC}, \rho_{BD}, \rho_{AB_CD}, \rho_{AB}, \rho_{CD}, \rho_{AC_BD})$.

- `rhoACmax()`: multiplicity of the $A \otimes C \to AC$ coupling.
- `rhoBDmax()`: multiplicity of the $B \otimes D \to BD$ coupling.
- `rhoABCDmax()`: multiplicity of the $AB \otimes CD \to T$ route.
- `rhoABmax()`: multiplicity of the $A \otimes B \to AB$ coupling.
- `rhoCDmax()`: multiplicity of the $C \otimes D \to CD$ coupling.
- `rhoACBDmax()`: multiplicity of the $AC \otimes BD \to T$ route.
- `operator()(rhoAC,rhoBD,rhoABCD,rhoAB,rhoCD,rhoACBD)`.
- `print(s = std::cout)`: tabular print over all six recoupling multiplicity axes.

In C++ the `U3U9` result is a view into a thread-local scratch buffer that the next database call overwrites; copy it before the next call. The other four own their buffers.

G. Coefficient databases

All coupling and recoupling coefficients are computed on demand and cached inside singleton database instances: `u3::wigU3U2db()`, `u3::wigU3SO3db()`, `u3::u3u6db()`, `u3::u3z6db()`, and `u3::u3u9db()`. The databases provide two ways to retrieve coefficients:

- **Structured container getters** (`get_U3U2`, `get_U3SO3`, `get_U3U6`, `get_U3Z6`, `get_U3U9`): return the dedicated multidimensional containers described above (`U3U2`, `U3SO3`, `U3U6`, `U3Z6`, `U3U9`). These containers support multi-index coordinate access via `operator()`, linear flat indexing via `operator[]`, shape accessors (`*max()`), and formatted stream printing.
- **Direct and flat-buffer getters** (`get`): return contiguous buffers without container wrapper overhead.

The available getters across the five databases are:

- **Canonical coupling database** (`u3::wigU3U2db()`):
  - `get_U3U2(` $n^a_{13}, n^a_{23}, n^a_{33}, n^a_{12}, n^a_{22},\ \ldots^b,\ \ldots^c$ `)` → `U3U2`: structured container over ρ.
  - `get(` $n^a_{13}, n^a_{23}, n^a_{33}, n^a_{12}, n^a_{22},\ \ldots^b,\ \ldots^c$ `)` → `Eigen::RowVectorXd`: direct row vector of reduced coupling coefficients $\langle Aa; Bb||Cc\rangle_\rho$ across all outer multiplicities $\rho = 0, \ldots, \rho_{\max} - 1$. Returns an empty vector if the coupling is forbidden or outside branching rules.
- **Physical coupling database** (`u3::wigU3SO3db()`):
  - `get_U3SO3(` $\lambda_1, \mu_1, L_1,\ \ \lambda_2, \mu_2, L_2,\ \ \lambda_3, \mu_3, L_3$ `)` → `U3SO3`: structured 4D container indexed by $(\kappa_A, \kappa_B, \kappa_C, \rho)$.
  - `get(` $\lambda_1, \mu_1, L_1,\ \lambda_2, \mu_2, L_2,\ \lambda_3, \mu_3, L_3$ `)` → `std::vector<double>`: flat coefficient buffer laid out in row-major order of $(\kappa_A, \kappa_B, \kappa_C, \rho)$, with size $\kappa_{A,\max}\kappa_{B,\max}\kappa_{C,\max}\rho_{\max}$.
- **U-recoupling database** (`u3::u3u6db()`):
  - `get_U3U6(` $A, B, ABC, C, AB,\ \rho_{AB}, \rho_{AB_C},\ BC, \rho_{BC}, \rho_{A_BC}$ `)` → `U3U6`: structured container indexed by $(\rho_{AB}, \rho_{AB_C}, \rho_{BC}, \rho_{A_BC})$.
  - `get(` $A, B, ABC, C, AB,\ \rho_{AB}, \rho_{AB_C},\ BC, \rho_{BC}, \rho_{A_BC}$ `)` → `std::vector<double>`: flat buffer of length $\rho_{AB}\rho_{AB_C}\rho_{BC}\rho_{A_BC}$.
  - `get(` $n^a_{13}, n^a_{23}, n^a_{33},\ \ldots,\ n^{bc}_{13}, n^{bc}_{23}, n^{bc}_{33}$ `)` (18 integers) → `std::vector<double>`: raw-integer overload taking the partition labels of A, B, ABC, C, AB, BC directly, computing intermediate ρ maxima internally.
- **Z-recoupling database** (`u3::u3z6db()`):
  - `get_U3Z6(` $B, A, ABC, C, AB,\ \rho_{AB}, \rho_{AB_C},\ AC, \rho_{AC}, \rho_{AC_B}$ `)` → `U3Z6`: structured container indexed by $(\rho_{AB}, \rho_{AB_C}, \rho_{AC}, \rho_{AC_B})$ (note the leading B, A exchanging the first pair).
  - `get(` $B, A, ABC, C, AB,\ \rho_{AB}, \rho_{AB_C},\ AC, \rho_{AC}, \rho_{AC_B}$ `)` → `std::vector<double>`: flat buffer of length $\rho_{AB}\rho_{AB_C}\rho_{AC}\rho_{AC_B}$.
  - `get(` $n^b_{13}, n^b_{23}, n^b_{33},\ \ldots,\ n^{ac}_{13}, n^{ac}_{23}, n^{ac}_{33}$ `)` (18 integers) → `std::vector<double>`: raw-integer overload taking the partition labels of B, A, ABC, C, AB, AC directly.
- **9-**U(3) **recoupling database** (`u3::u3u9db()`):
  - `get_U3U9(` $A, B, C, D, AB, CD, AC, BD, ABCD,\ \rho_{AC}, \rho_{BD}, \rho_{ABCD}, \rho_{AB}, \rho_{CD}, \rho_{ACBD}$ `)` → `U3U9`: structured container view. Multiplicity parameters follow the container's axis order: column intermediate channels $(\rho_{AC}, \rho_{BD}, \rho_{ABCD})$ followed by row intermediate channels $(\rho_{AB}, \rho_{CD}, \rho_{ACBD})$.
  - `get(` $A, B, C, D, AB, CD, AC, BD, ABCD,\ \rho_{AB}, \rho_{CD}, \rho_{ABCD}, \rho_{AC}, \rho_{BD}, \rho_{ACBD}$ `)` → `std::span<const double>` (or `std::vector<double>`): flat buffer of length $\rho_{AC}\rho_{BD}\rho_{ABCD}\rho_{AB}\rho_{CD}\rho_{ACBD}$. **Note the multiplicity ordering difference**: in `get`, intermediate row channels $(\rho_{AB}, \rho_{CD}, \rho_{ABCD})$ precede column channels $(\rho_{AC}, \rho_{BD}, \rho_{ACBD})$, whereas in `get_U3U9` the column channels appear first to match internal storage strides.
  - `get(` $n^a_{13}, n^a_{23}, n^a_{33},\ \ldots,\ n^{abcd}_{13}, n^{abcd}_{23}, n^{abcd}_{33}$ `)` (27 integers) → `std::vector<double>`: raw-integer overload taking the partition labels of all 9 participating irreps.

The multiplicity maxima passed to recoupling routines must match the exact `Rho` values for the corresponding channels; a mismatch triggers an internal assertion. A forbidden or empty coupling returns an empty container or buffer. You can query its shape, but must not index it.

Each database singleton provides a `clear()` member function (and `u3::clear()` clears all five) to remove cached coefficient blocks while retaining allocated thread workspaces.

H. Conventions and normalization

Phase and normalization conventions follow Ref. [1]; documented values and the generation procedure are in `test/reference/`.

- **Multiplicity indexing:** Counts come from `Kappa`/`Rho`; container accessors use zero-based indices.
- **Canonical-coupling phase:** For each ρ block the phase and normalization are fixed by the highest-weight reduced coefficients—the orthonormal null space of the system imposed by E_{23} and E_{12} annihilating the highest-weight coupled state. For $\rho_{\max} > 1$ (and self-conjugate irreps) the solver fixes a valid orthonormal ρ basis (Row Reduced Echelon Form + Gram-Schmidt + positive leading sign); any alternative must be applied consistently to every ρ-carrying coefficient. Conjugation of a Gelfand state satisfies $C\,|g\rangle = (-1)^{n_{23}-n_{12}-n_{22}-n_{11}}\,|g.\text{conjugate}()\rangle$ (`gelfand::conjugate_phase()`).
- **Physical-to-canonical transformation:** A physical SO(3) state $|(\lambda\mu)\,\kappa\,L\,M\rangle$ is expanded in canonical Gelfand states $|g\rangle$ with $M = e_{11} - e_{22} = 2n_{11} - (n_{12} + n_{22})$. The highest-weight state $M = L$ belongs to the null space of L_+; L_- fixes the lower-M phases within a κ multiplet. Distinct conjugate irreps obey $C\,|(\lambda\mu)\,\kappa\,L\,M\rangle = (-1)^{\lambda+\mu-L+M}\,|(\mu\lambda)\,\kappa\,L,-M\rangle$ (for $\lambda > \mu$ the transformation is built from the conjugate irrep with the phase factor $(-1)^{n_{22}+n_{11}-L}$). For a self-conjugate irrep (λ,λ) conjugation acts as an M-independent symmetric matrix D ($D^2 = \mathbb{1}$) that may mix κ.
- **Generator reduced matrix element factor:** `RME_L`/`RME_Q` carry an extra $\sqrt{2L'+1}$, so $\langle\kappa L\|\mathbf{Q}\|\kappa' L'\rangle = (-1)^{L-L'}\langle\kappa' L'\|\mathbf{Q}\|\kappa L\rangle$.
- **Input checks.** Invalid labels or out-of-range indices abort via assertion.

IV. PYTHON BINDINGS

The `u3lib` module mirrors the C++ API. Build it with `make python`, using an interpreter whose ABI matches the extension. Irreps (`su3.irrep`, `irrep`), `gelfand`, `physicalvector`, the free functions `Rho`, `Kappa`, `decomposition`, `gelfandBasis`, `physicalBasis`, the physical-generator reduced matrix elements `RME_L`/`RME_Q`, the transformation `phys_to_cano`/`phys_to_cano_allM`, and the five database accessors are all exposed with the same names and semantics as Sec. III. The structured containers provide `__call__` (multiplicity-indexed), flat `data()` as a NumPy array, and formatted printing via `print()` and `__str__`; database access before `initialize()` or after `finalize()` raises `RuntimeError` rather than aborting. The `U3U9` container, a scratch-buffer view in C++, is copied into owning storage automatically in Python. For worked examples of irreps, bases, coupling, and recoupling, see the Jupyter notebook `python/tutorial.ipynb`. Build and toolchain details are in the top-level `README.md`.

V. AI-READY INTERFACE

A. Skill catalog and orchestration

The `agents/` directory contains skills that guide an AI assistant through `u3lib` calculations. The catalog includes an index (`agents/AGENTS.md`), shared runtime instructions, and a skill for each capability. Each skill specifies the inputs, library call, conventions, and a numerical check. An assistant with code-execution access uses these instructions to run the configured local library. The library calculations do not require a network service.

The `install` skill (`agents/install/SKILL.md`) covers setup on a fresh checkout. The agent first runs `make config` to resolve or download dependencies, followed by `make` and `make test`. It can then run the calculation skills. Table II gives a sample request for each skill, with expected results checked against the library.

Skill	Example prompt (expected result)
`dim`	"Dimension of the SU(3) irrep $(2,1)$ – equivalently the U(3) irrep $[3,1,0]$ or any shift such as $[4,2,1]$?" (15; shift-invariant)
`casimirs`	"C_2, C_3 for the SU(3) irrep $(3,1)$, and C_1, C_2, C_3 for the U(3) irrep $[4,2,1]$." (SU(3): $50/3, 160/9$; U(3): C_1=7, 32/3, 56/9, U(3)-only)
`cano-list`	"Enumerate the canonical Gelfand basis of $(2,0)$ with each state's $\epsilon, 2\Lambda, 2M$ labels." (6 states)
`phys-list`	"List the physical SO(3) basis (κ, L) of $(4,2)$ and verify the dimension sum rule." (L=6, 5, 4, 3, 2, 0 with κ=1, 1, 2, 1, 2, 1; $\sum \kappa(2L{+}1)$=60)
`cano-generators`	"Act with E_{12} on the Gelfand state $\lvert 2,0,0;2,0;1\rangle$ of $(2,0)$; give the resulting state and matrix element." ($\lvert 2,0,0;2,0;2\rangle$ with $\sqrt{2}$)
`phys-generators`	"Compute $\langle (2,0)\,\kappa{=}1\,L{=}2 \Vert \mathbf{Q} \Vert (2,0)\,\kappa{=}1\,L{=}2\rangle$ and $\langle \Vert \mathbf{L} \Vert \rangle$." ($\sqrt{35/3} \approx 3.416$ and $\sqrt{30} \approx 5.477$)
`phys-to-cano`	"Express $\lvert (2,0)\,\kappa{=}1\,L{=}2\,M{=}0\rangle$ in the canonical Gelfand basis." (amplitudes $\sqrt{1/3}, \sqrt{2/3}$)
`inner-mult`	"SO(3) multiplicity κ of L=2 in $(4,2)$?" (2)
`outer-mult`	"Outer multiplicity ρ of $(1,1)$ in $(1,1) \otimes (1,1)$?" (2)
`decomposition`	"Full decomposition of $(2,0) \otimes (0,2)$ with outer multiplicities; check the dimension sum rule." ($(2,2){+}(1,1){+}(0,0)$)
`cano-coupling`	"U(3)$\supset$U(2) reduced coupling for highest-weight states $[2,0,0]\,[2,0] \times [2,0,0]\,[2,0] \to [4,0,0]\,[4,0]$ of $(2,0) \times (2,0) \to (4,0)$; verify it equals 1."
`phys-coupling`	"SO(3)-reduced coupling $(2,0)\,\kappa{=}1\,L{=}2 \times (2,0)\,\kappa{=}1\,L{=}2 \to (4,0)\,\kappa{=}1\,L{=}2$; check against $-\sqrt{2/9}$."
`u6recoupling`	"U-recoupling for $(1,0) \otimes (1,0) \otimes (1,0) \to (3,0)$ through intermediate irreps $AB = (2,0)$ and $BC = (2,0)$." (1)
`z6recoupling`	"Z-recoupling between $(AB)C$ and $(AC)B$ routes for $(1,0) \otimes (1,0) \otimes (1,0) \to (3,0)$ with intermediate irreps $AB = (2,0)$ and $AC = (2,0)$." (1)
`u9recoupling`	"9-$(\lambda\mu)$ recoupling for four fundamentals $(1,0)$ coupling to $(4,0)$ with intermediate irreps $AB = CD = AC = BD = (2,0)$." (1)
`conjugation`	"Conjugate the SU(3) irrep $(2,1)$ and its Gelfand states, reporting the phases." (irrep $\to (1,2)$, phase power -1)

TABLE II: The `u3lib` agent skills with a representative prompt for each.

The assistant translates a request into library calls: it selects a skill, prepares inputs in the required conventions, runs the compiled C++ or Python routine, and reports the results with their labels and numerical check. The library computes the coefficients, Casimirs, and matrix elements; the assistant does not derive or approximate them. If a request is ambiguous or asks for an undefined quantity, the assistant asks for clarification or explains the problem. For example, a first-order Casimir is not defined for an SU(3) irrep.

B. Evaluation objectives

To evaluate the practical value of the skill catalog, we compare autonomous coding agents with and without access to the skills. We ask whether the skills help agents complete tasks correctly with fewer resources, and whether the benefit depends on the implementation complexity of the task. We examine two hypotheses:

1. Resource efficiency: Domain skills reduce token consumption, execution time, and API cost by supplying conventions and usage patterns that the agent would otherwise need to discover from the source.

2. Dependence on task complexity: The benefit increases as tasks move from Python integration to C++ workspace management and parallel execution, where memory ownership and thread-safety requirements introduce additional constraints.

C. Experimental methodology

1. Benchmark Tasks

We designed three tasks with increasing implementation complexity, covering Python integration, C++ workspace management, and parallel execution:

- Task 1: Python Racah recoupling (`consumer/recouple.py`). A standalone Python utility computing U- and Z-recoupling coefficients through the pybind11 interface and verifying unitary symmetry relations across fundamental representations (e.g. $(1,0)\times(1,0)\times(1,0) \to (3,0)$). *Architectural complexity: low* (declarative bindings, automatic memory, single-threaded).
- Task 2: C++ 9-U(3) batch exporter (`consumer/u9_batch.cpp`). A C++ executable evaluating 9-U(3) blocks via `u3::u3u9db()`: it parses the multi-irrep label specification, computes intermediate outer multiplicities, and serializes block and entry records, copying scratch-buffer views into owning storage to avoid aliasing across iterations. *Architectural complexity: medium* (C++ typing, headers, manual workspace lifecycle).
- Task 3: C++ multi-threaded coupling (`consumer/coupling_batch.cpp`). Parallel canonical- and physical-basis coupling with OpenMP (`#pragma omp parallel for`), requiring per-thread database cache and scratch state (initialized on each thread) to avoid races, correct `-fopenmp` configuration, and deterministic output. *Architectural complexity: high* (shared-memory concurrency, thread-private state isolation).

2. Isolated test environments

Each run used a separate, disposable sandbox directory with its own consumer build environment, so files from one trial were unavailable to subsequent trials. The two conditions differed only in access to the skills:

- Control (zero skills): a clean checkout containing the source, headers, and build scripts, with the skill directory (`agents/`) removed.
- Treatment (domain skills): the same workspace plus the repository-local skill catalog indexed by `agents/AGENTS.md`, which the prompt directed the agent to consult.

3. Hidden tests and grader validation

The agent could inspect files and directories, edit code, and execute shell commands. When it declared completion or reached the limit of 50 steps, the environment was locked and an automated grader tested the resulting program against a hidden validation suite. The tests compared coupling and recoupling values with exact group-theoretical results across a range of irreps, using a 10^{-12} tolerance.

Before the trials, we tested the grader by introducing phase inversions, memory leaks, uninitialized buffer reads, and artificial race conditions into reference implementations. On these validation cases, the grader achieved 100% sensitivity and specificity, with no false positives or false negatives.

4. Model and Statistical Protocol

All experiments used Anthropic's Claude Sonnet 5.5 (`claude-sonnet-5-5`) in an autonomous agent test environment. For each task, $N = 10$ matched Control/Treatment pairs were run (60 runs total), recording four metrics per run: functional pass rate; total token consumption (prompt, cache-read, and completion tokens); wall-clock latency; and API cost at official token pricing. Significance was assessed with two-tailed Welch's t-tests (unequal variance), reporting t, degrees of freedom, and p.

D. Results and discussion

1. Overall results

Table III and Fig. 1 summarize the 60 runs. Across the three tasks:

TABLE III: Empirical performance comparison between Control (Zero Skills) and Treatment (Domain Skills) across $N = 10$ independent matched pairs per task (60 runs total) using Claude Sonnet 5.5. Tokens, execution duration, and costs are reported as Mean ± Standard Deviation. Statistical significance is assessed via two-tailed Welch's t-tests.

Task	Condition	Pass Rate	Total Tokens	Time (s)	Cost ($)	Token Δ	Time Δ
Task 1	Control (N=10)	100% (10/10)	$381,613 \pm 80,140$	58.90 ± 10.51	$0.2780	—	—
	Treatment (N=10)	100% (10/10)	$270,260 \pm 93,890$	46.61 ± 7.88	$0.2280	**−29.2%***	**−20.9%****
Task 2	Control (N=10)	100% (10/10)	$614,511 \pm 144,848$	105.21 ± 20.86	$0.3751	—	—
	Treatment (N=10)	100% (10/10)	$329,829 \pm 62,106$	71.42 ± 7.74	$0.2499	**−46.3%*****	**−32.1%*****
Task 3	Control (N=10)	90.0% (9/10)	$654,185 \pm 163,874$	110.73 ± 18.35	$0.3934	—	—
	Treatment (N=10)	100% (10/10)	$320,246 \pm 141,267$	65.89 ± 15.71	$0.2459	**−51.0%*****	**−40.5%*****
Aggregate	Control (N=30)	96.7% (29/30)	$550,103 \pm 178,309$	91.61 ± 28.85	$0.3488	—	—
	Treatment (N=30)	100% (30/30)	$306,778 \pm 104,076$	61.31 ± 15.21	$0.2413	**−44.2%*****	**−33.1%*****

$^{*}p < 0.05$ ($t = -2.85$); $^{**}p < 0.01$ ($t = -2.96$); $^{***}p < 0.001$ (Task 2: $t_{\text{tok}} = -5.71, t_{\text{time}} = -4.80$; Task 3: $t_{\text{tok}} = -4.88, t_{\text{time}} = -5.87$; Grand Aggregate: $t_{\text{tok}} = -6.46, t_{\text{time}} = -5.09$; Welch's two-tailed t-test).

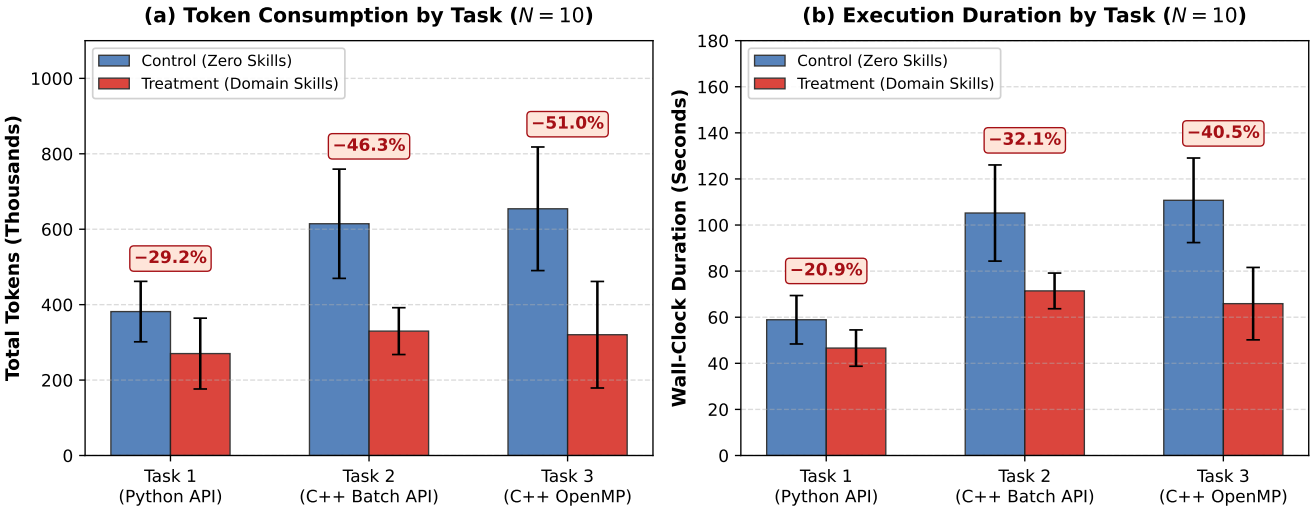


FIG. 1: Benchmark results comparing autonomous agent performance between Control (Zero Skills) and Treatment (Domain Skills) across $N = 10$ independent trials per task. (a) Total token consumption (thousands) by task, showing monotonic savings of 29.2%, 46.3%, and 51.0%. (b) Wall-clock execution duration (seconds) by task, showing speedups of 20.9%, 32.1%, and 40.5%. Error bars indicate ±1 sample standard deviation.

- Token Consumption: Providing domain skills reduced mean token consumption from $550,103 \pm 178,309$ tokens to $306,778 \pm 104,076$ tokens—an aggregate reduction of 44.2% ($t = -6.46$, $p = 6.0 \times 10^{-8}$, Welch's t-test).
- Execution Latency: Wall-clock execution time decreased from 91.61 ± 28.85 s to 61.31 ± 15.21 s, representing an aggregate speedup of 33.1% ($t = -5.09$, $p = 7.2 \times 10^{-6}$).
- API Cost: Direct inference costs fell from an average of $0.3488 to $0.2413 per task (−30.8%, $t = -7.16$, $p = 4.4 \times 10^{-9}$).
- Reliability: Treatment runs achieved a 100% pass rate across all 30 evaluations. One Control run failed Task 3 (9/10), giving an aggregate pass rate of 96.7%.

2. Dependence on task complexity

For the three tasks considered here, the relative reductions in token consumption and execution time increase with the intended progression in task complexity:

$$\Delta_{\text{tokens}} = -29.2\% \text{ (Task 1)} \longrightarrow -46.3\% \text{ (Task 2)} \longrightarrow -51.0\% \text{ (Task 3)},$$
$$\Delta_{\text{time}} = -20.9\% \text{ (Task 1)} \longrightarrow -32.1\% \text{ (Task 2)} \longrightarrow -40.5\% \text{ (Task 3)}.$$

This trend is consistent with the hypothesis that explicit guidance becomes more useful as implementation constraints increase. Three tasks alone do not establish a general scaling law; the following observations describe agent behavior within this benchmark:

1. Task 1 (API discovery): Control agents produced correct recoupling solutions but typically needed 3–6 shell turns (`grep`, `ls`, inspecting pybind11 modules) to discover the exposed bindings (`u3lib.u6recoupling`, `u3lib.z6recoupling`) and their argument order. Treatment agents retrieved the signatures directly from the catalog, finishing in 16.6 steps versus 22.8 for Control.

2. Task 2 (scratch-buffer aliasing): The 9-U(3) database (`u3::u3u9db()`) returns views into internal scratch buffers reused across queries. Control agents encountered aliasing bugs when later queries overwrote stored blocks, spent time locating `U3_U9Coeffs.h`, and misordered the multiplicity strides, inflating consumption to $614,511$ tokens on average (worst case exceeding $853,000$). Treatment agents used the documented copying and indexing procedures and completed the task in $329,829$ tokens with about half the variance.

3. Task 3 (concurrency): The coefficient databases keep per-thread cache and scratch state, so sharing one database across threads without initializing each thread causes races. The second Control run failed the hidden tests after attempts to resolve non-deterministic segmentation faults; even successful Control runs averaged 33.6 steps and 110.7 s to find the need for per-thread state. Treatment agents applied the skill's OpenMP guidance from the first turn, completing the task with deterministic output in 65.89 s and $320,246$ tokens.

VI. BUILD AND INSTALLATION

`u3lib` is header-only; user programs include the headers in `include/` directly. The source is hosted publicly at https://github.com/Deep-SANE/u3lib. The build reads dependency paths from a configuration environment. After sourcing it from the repository root, use the following make targets: `make` (example programs → `bin/`), `make test` (build and run the regression suite), and `make python` (the pybind11 extension). The regression tests check dimensions, Gelfand interlacing and ordering, SO(3) branching, canonical/physical coupling orthonormality across κ and ρ, conjugation identities, and documented analytic values; comparisons use numerical tolerances, since results need not be bitwise identical across backends.

`make config` locates the dependencies and writes `config/config.generated.mk`. It uses existing installations or downloads pinned versions from upstream sources and builds them under `external-libs/`, which is not part of the repository:

- `libwignernj`: https://github.com/susilehtola/libwignernj
- `Eigen3`: https://eigen.tuxfamily.org
- `Boost`: https://www.boost.org
- `OpenMP` (via the compiler runtime, e.g. GCC's libgomp): https://www.openmp.org
- `fmt`: https://github.com/fmtlib/fmt
- `pybind11`: https://github.com/pybind/pybind11

The `KraNS` null-space solver (https://github.com/Deep-SANE/krans) is used as a sibling checkout alongside `u3lib` rather than vendored under `external-libs/`; `make config` clones it from the Deep-SANE organization and builds it in place when it is not already present.

Acknowledgments

This work is dedicated to the memory of J. P. Draayer, who passed on September 13, 2026 and whose pioneering contributions to algebraic computational nuclear physics through Elliott's SU(3) model, the Symplectic Sp(3, R) model, the *ab initio* Symmetry-Adapted No-Core Shell Model, and his recent effort in Wigner's SU(4) model inspired this framework. The authors also gratefully acknowledge F. Pan and D. Kekejian for valuable discussions. PD was supported by the Institute of International Education through the Quad Fellowship, Louisiana State University, and Quantum CodeX. DL and TD were funded by Quantum CodeX.

---

[1] P. Dang, J. P. Draayer, F. Pan, and K. S. Becker, "New Procedure for Evaluation of U(3) Coupling and Recoupling Coefficients," Eur. Phys. J. Plus **139**, 704 (2024), doi:10.1140/epjp/s13360-024-05501-8.
[2] J. P. Draayer and Y. Akiyama, "Wigner and Racah coefficients for SU_3," J. Math. Phys. **14**, 1904 (1973), doi:10.1063/1.1666267.
[3] Y. Akiyama and J. P. Draayer, "A User's Guide to Fortran Programs for Wigner and Racah Coefficients of SU_3," Comp. Phys. Comms. **5**, 405 (1973), doi:10.1016/0010-4655(73)90077-5.
[4] J. P. Elliott, "Collective motion in the nuclear shell model. I. Classification schemes for states of mixed configurations," Proc. R. Soc. Lond. A **245**, 128–145 (1958), doi:10.1098/rspa.1958.0072.
[5] J. P. Elliott, "Collective motion in the nuclear shell model II. The introduction of intrinsic wave-functions," Proc. R. Soc. Lond. A **245**, 562–581 (1958), doi:10.1098/rspa.1958.0101.
[6] T. Dytrych, K. D. Launey, J. P. Draayer, P. Maris, J. P. Vary, *et al.*, "Collective Modes in Light Nuclei from First Principles," Phys. Rev. Lett. **111**, 252501 (2013), doi:10.1103/PhysRevLett.111.252501.
[7] T. Dytrych, K. D. Launey, J. P. Draayer, D. J. Rowe, J. L. Wood, *et al.*, "Physics of Nuclei: Key Role of an Emergent Symmetry," Phys. Rev. Lett. **124**, 042501 (2020), doi:10.1103/PhysRevLett.124.042501.
[8] A. E. McCoy, M. A. Caprio, T. Dytrych, and P. J. Fasano, "Emergent Sp(3, $\mathbb{R}$) Dynamical Symmetry in the Nuclear Many-Body System from an Ab Initio Description," Phys. Rev. Lett. **125**, 102505 (2020), doi:10.1103/PhysRevLett.125.102505.
[9] V. K. B. Kota, "SU(3) Symmetry in Atomic Nuclei," Springer, Singapore (2020), doi:10.1007/978-981-15-3603-8.
[10] D. Bonatsos, A. Martinou, S. K. Peroulis, T. J. Mertzimekis, and N. Minkov, "The Proxy-SU(3) Symmetry in Atomic Nuclei," Symmetry **15**, 169 (2023), doi:10.3390/sym15010169.
[11] J. Cseh, "Shell-like quarteting in heavy nuclei: Algebraic approaches based on the pseudo- and proxy-SU(3) schemes," Phys. Rev. C **101**, 054306 (2020), doi:10.1103/PhysRevC.101.054306.
[12] A. P. Zuker, J. Retamosa, A. Poves, and E. Caurier, "Spherical shell model description of rotational motion," Phys. Rev. C **52**, R1741 (1995), doi:10.1103/PhysRevC.52.R1741.
[13] A. Gade, B. Longfellow, R. V. F. Janssens, D. D. Dao, F. Nowacki, *et al.*, "In-beam spectroscopy reveals competing nuclear shapes in the rare isotope ^{62}Cr," Nat. Phys. **21**, 37–42 (2025), doi:10.1038/s41567-024-02680-0.
[14] J. Cseh, "Microscopic structure and mathematical background of the multiconfigurational dynamical symmetry," Phys. Rev. C **103**, 064322 (2021), doi:10.1103/PhysRevC.103.064322.
[15] C. Bahri, D. J. Rowe and J. P. Draayer, "Programs for generating Clebsch–Gordan coefficients of SU(3) in SU(2) and SO(3) bases," Comp. Phys. Comms. **159**, 121 (2004), doi:10.1016/j.cpc.2004.01.005.
[16] T. Dytrych, D. Langr, J. P. Draayer, K. D. Launey and D. Gazda, "SU3lib: A C++ library for accurate computation of Wigner and Racah coefficients of SU(3)," Comp. Phys. Comms **269**, 108137 (2021), doi:10.1016/j.cpc.2021.108137.
[17] J. Herko, M. A. Caprio, A. E. McCoy and P. J. Fasano, "A modern Fortran library for SU(3) coupling and recoupling coefficients," Eur. Phys. J. A **62**, 95 (2026), doi:10.1140/epja/s10050-026-01845-0.
[18] A. Alex, M. Kalus, A. Huckleberry, and J. von Delft, "A numerical algorithm for the explicit calculation of SU(N) and SL(N, $\mathbb{C}$) Clebsch–Gordan coefficients," J. Math. Phys. **52**, 023507 (2011), doi:10.1063/1.3521562.
[19] A. Krastenov, "Sparse Null Space Solver in C++," Master's thesis, Faculty of Information Technology, Czech Technical University in Prague (2026).
[20] M. Gell-Mann, "Symmetries of Baryons and Mesons," Phys. Rev. **125**, 1067 (1962), doi:10.1103/PhysRev.125.1067.
[21] Y. Ne'eman, "Derivation of strong interactions from a gauge invariance," Nucl. Phys. **26**, 222 (1961), doi:10.1016/0029-5582(61)90134-1.
[22] D. Bacon, I. L. Chuang, and A. W. Harrow, "Efficient Quantum Circuits for Schur and Clebsch–Gordan Transforms," Phys. Rev. Lett. **97**, 170502 (2006), doi:10.1103/PhysRevLett.97.170502.
[23] C. P. Moca, A. Alex, J. von Delft, and G. Zaránd, "SU(3) Anderson impurity model: A numerical renormalization group approach exploiting non-Abelian symmetries," Phys. Rev. B **86**, 195128 (2012), doi:10.1103/PhysRevB.86.195128.
[24] S.-H. Tan, Y. Y. Gao, H. de Guise, and B. C. Sanders, "SU(3) Quantum Interferometry with Single-Photon Input Pulses," Phys. Rev. Lett. **110**, 113603 (2013), doi:10.1103/PhysRevLett.110.113603.
[25] P. Draper and D. Gupta, "Impact of Irrep Truncations on Quantum Simulations of QCD," arXiv:2610.23681 (2026), doi:10.48550/arXiv.2610.23681.