A Computer-Assisted Proof of Sun's Identity (3)

Nikita Kalinin

Problem. Prove the identity

$$\sum_{n=1}^{\infty} \frac{41 + 38nH_{n-1} + 9n^2H_{n-1}^2 - 16n^3H_{n-1}^3 - 6n^4H_{n-1}^4}{n^6\binom{2n}{n}} = 4\zeta(6) + 11\zeta(3)^2.$$

This is conjecture (3) in the MathOverflow post by Zhi-Wei Sun [Cf. Z.-W. Sun, "Few conjectural series for $\zeta(5)$ and $\zeta(6)$," MathOverflow, 2026][1].

Theorem 1. *We have*

$$\sum_{n=1}^{\infty} \frac{41 + 38nH_{n-1} + 9n^2H_{n-1}^2 - 16n^3H_{n-1}^3 - 6n^4H_{n-1}^4}{n^6\binom{2n}{n}} = 4\zeta(6) + 11\zeta(3)^2.$$

Proof. Set

$$A_n := nH_n.$$

Since $H_{n-1} = H_n - \frac{1}{n}$, a direct expansion gives

$$41 + 38nH_{n-1} + 9n^2H_{n-1}^2 - 16n^3H_{n-1}^3 - 6n^4H_{n-1}^4 = 22 - 4A_n + 21A_n^2 + 8A_n^3 - 6A_n^4.$$

Therefore it is enough to prove

$$\sum_{n=1}^{\infty} \frac{22 - 4A_n + 21A_n^2 + 8A_n^3 - 6A_n^4}{n^6\binom{2n}{n}} = 4\zeta(6) + 11\zeta(3)^2.$$

We now use the two exact coefficient-extraction identities in [C. Li and W. Chu, "Infinite series about harmonic numbers inspired by Zeilberger's algorithm," *Electronic Research Archive* **31** (2023), 4611–4636][2], namely Proposition 10 and Proposition 11. For brevity, write

$$\Gamma\left[\frac{\alpha_1, \dots, \alpha_r}{\beta_1, \dots, \beta_s}\right] := \frac{\Gamma(\alpha_1)\cdots\Gamma(\alpha_r)}{\Gamma(\beta_1)\cdots\Gamma(\beta_s)}, \qquad \left[\frac{u_1, \dots, u_r}{v_1, \dots, v_s}\right]_n := \frac{(u_1)_n \cdots (u_r)_n}{(v_1)_n \cdots (v_s)_n}.$$

The two parent identities are

$$\begin{aligned}\mathcal{A}(a,b,d;x) &:= \Gamma\left[\frac{1+(a-b)x,\ \frac{1}{2}+(a+b)x,\ 1+dx,\ \frac{1}{2}+dx}{1+ax,\ \frac{1}{2}+ax,\ \frac{1}{2}+(b+d)x,\ 1+(-b+d)x}\right] \\ &= \sum_{n=0}^{\infty} \frac{2ax+3n}{2ax} \left[\frac{(2ax),\ (2bx),\ (1-2bx),\ ((a-d)x)}{1,\ 1+(a-b)x,\ \frac{1}{2}+(a+b)x,\ 1+2dx}\right]_n \frac{1}{4^n},\end{aligned}$$

$$\begin{aligned}\mathcal{B}(a,b,d;x) &:= \Gamma\left[\frac{\frac{1}{2},\ \frac{1}{2}+(a-b)x,\ 1+(a-d)x,\ 1+(b+d)x}{1+ax,\ 1+bx,\ \frac{1}{2}+dx,\ \frac{1}{2}+(a-b-d)x}\right] \\ &= \sum_{n=0}^{\infty} \frac{2ax+3n}{2ax} \left[\frac{(ax),\ (1+2bx),\ (2dx),\ ((2a-2b-2d)x)}{1,\ 1+(a-d)x,\ 1+(b+d)x,\ \frac{1}{2}+(a-b)x}\right]_n \frac{1}{4^n}.\end{aligned}$$

[1] https://mathoverflow.net/questions/508663/few-conjectural-series-for-zeta5-and-zeta6

[2] https://www.aimspress.com/aimspress-data/era/2023/8/PDF/era-31-08-236.pdf

Expand both sides in powers of x:

$$\mathcal{A}(a,b,d;x)=\sum_{m\ge 0}x^m A_m(a,b,d),\qquad \mathcal{B}(a,b,d;x)=\sum_{m\ge 0}x^m B_m(a,b,d).$$

For nonnegative integers i,j,k, let

$$R^{(10)}_{i,j,k}:=[a^ib^jd^k]A_6(a,b,d),\qquad R^{(11)}_{i,j,k}:=[a^ib^jd^k]B_6(a,b,d).$$

Each such coefficient extraction is an exact identity between a denominator-binomial series on the hypergeometric side and a closed form in the $\mathbb{Q}$-span of $\zeta(6)$ and $\zeta(3)^2$ on the gamma-quotient side.

Using exact symbolic expansion of the two propositions to order x^6, followed by exact coefficient extraction in a,b,d, and then exact rational linear algebra over $\mathbb{Q}$, one obtains the following explicit linear combination:

$$\begin{aligned}T:=&\frac{35500}{22347}R^{(10)}_{0,1,5}-\frac{420035}{178776}R^{(10)}_{0,2,4}+\frac{973549}{357552}R^{(10)}_{0,3,3}-\frac{806983}{357552}R^{(10)}_{0,4,2}+\frac{97379}{59592}R^{(10)}_{0,5,1}\\&+\frac{93505}{59592}R^{(10)}_{1,1,4}-\frac{68141}{29796}R^{(10)}_{1,2,3}+\frac{242933}{119184}R^{(10)}_{1,3,2}-\frac{175459}{119184}R^{(10)}_{1,4,1}\\&+\frac{89299}{39728}R^{(10)}_{2,1,3}-\frac{100879}{39728}R^{(10)}_{2,2,2}+\frac{4614}{2483}R^{(10)}_{2,3,1}\\&+\frac{119463}{39728}R^{(10)}_{3,1,2}-\frac{88503}{39728}R^{(10)}_{3,2,1}+\frac{40047}{19864}R^{(10)}_{4,1,1}\\&-\frac{58174}{22347}R^{(11)}_{0,0,6}-\frac{1535}{1719}R^{(11)}_{0,2,4}-\frac{296}{573}R^{(11)}_{1,1,4}.\end{aligned}$$

This linear combination has the following exact effect:

- on the series side,
$$T=\sum_{n=1}^{\infty}\frac{22-4A_n+21A_n^2+8A_n^3-6A_n^4}{n^6\binom{2n}{n}};$$
- on the closed-form side,
$$T=4\zeta(6)+11\zeta(3)^2.$$

Hence

$$\sum_{n=1}^{\infty}\frac{22-4A_n+21A_n^2+8A_n^3-6A_n^4}{n^6\binom{2n}{n}}=4\zeta(6)+11\zeta(3)^2.$$

Substituting back $A_n=nH_n$ and then replacing H_n by $H_{n-1}+\frac{1}{n}$ yields

$$\sum_{n=1}^{\infty}\frac{41+38nH_{n-1}+9n^2H_{n-1}^2-16n^3H_{n-1}^3-6n^4H_{n-1}^4}{n^6\binom{2n}{n}}=4\zeta(6)+11\zeta(3)^2,$$

as required.

Finally, the attached verification script carries out the above extraction and linear solve in exact rational arithmetic and checks the explicit certificate by computing the residuals on both sides. The output is

`lhs residual = 0`, `rhs residual = 0`,

so the certificate is exact. □

Remark 1. *This is a computer-assisted proof, but every step is exact: the two parent identities are exact analytic identities, the coefficient extraction is symbolic, the linear algebra is over $\mathbb{Q}$, and the final residual check is exact. No floating-point numerics are used.*

Appendix: Python code used for the exact certificate search

The proof above is implemented by the accompanying Python script `identity3_prop10_prop11_extractor_v3.`
The script uses `sympy` only, and every step is exact.

The main helper functions have the following roles.

- `clean_series`, `add_series`, `mul_series`, `scale_series`, `shift_series`: represent and manipulate truncated Laurent series in the variable x. A series is stored as a dictionary mapping each power of x to its exact coefficient.
- `reciprocal_series`: computes the reciprocal of a power series with constant term 1, truncated to the required order.
- `exp_series_from_log`: reconstructs $\exp(L(x))$ from the coefficients of $L(x)$, using the standard recurrence for truncated exponential series.
- `coeff_param`, `exact_degree_monomials`, `poly_support`, `expr_vec`: perform exact coefficient extraction in the parameters a, b, d, then convert identities into vectors for exact linear algebra over $\mathbb{Q}$.
- `one_shift_series`, `half_shift_series`, `zero_shift_series`: build the formal expansions of the Pochhammer-type factors appearing in Proposition 10 and Proposition 11.
- `gamma_log_series`: expands the logarithm of the gamma-quotient factors into zeta-values, again in exact symbolic form.
- `build_shift`, `build_series_kernel`, `build_gamma_kernel`: assemble the full series-side and closed-form-side kernels attached to Proposition 10 and Proposition 11.
- `extract_rows`: extracts all rows $[a^i b^j d^k][x^6]$ from a chosen proposition.
- `solve_exact_relation`: builds the exact matrix of coefficients and solves for a rational linear combination matching the target identity.
- `main`: runs the whole pipeline, prints the particular 18-term certificate, and performs the final exact residual check.

For convenience, we include the full script below.

Listing 1: Exact SymPy extractor used in the proof

```
#!/usr/bin/env python3
"""
Exact SymPy extractor for Li--Chu (ERA 2023), Proposition 10 and Proposition 11.

This version uses custom truncated Laurent-series arithmetic in x, which is much
faster than repeated calls to SymPy's general-purpose series engine.

It extracts the [x^6][a^i b^j d^k] rows from the normalized series-side kernels
and from the matching gamma-quotient side, then optionally tries an exact
rational relation search for identity (3).
"""

import sympy as sp
from pathlib import Path

# ------------------------------------------------------------------------------
# Global symbols
# ------------------------------------------------------------------------------

x, a, b, d, n = sp.symbols("x a b d n")

h1, h2, h3, h4, h5, h6 = sp.symbols("h1 h2 h3 h4 h5 h6")
o1, o2, o3, o4, o5, o6 = sp.symbols("o1 o2 o3 o4 o5 o6")
H_RAW = [h1, h2, h3, h4, h5, h6]
O_RAW = [o1, o2, o3, o4, o5, o6]
SERIES_GENS = H_RAW + O_RAW

Z2, Z3, Z4, Z5, Z6 = sp.symbols("Z2 Z3 Z4 Z5 Z6")
ZETA = {2: Z2, 3: Z3, 4: Z4, 5: Z5, 6: Z6}
ZETA_GENS = [Z2, Z3, Z4, Z5, Z6]
```

```
MAX_X_ORDER = 6
PRINT_MAX_ROWS_PER_RECIPE = 10
WRITE_ALL_ROWS = True
TRY_SOLVE = True
ROWS_DUMP = Path(__file__).with_name("identity3_rows_dump.txt")

TARGET_LHS = sp.expand(
    sp.Rational(22, 1) / n**6
    - sp.Rational(4, 1) * h1 / n**5
    + sp.Rational(21, 1) * h1**2 / n**4
    + sp.Rational(8, 1) * h1**3 / n**3
    - sp.Rational(6, 1) * h1**4 / n**2
)
TARGET_RHS = 4 * Z6 + 11 * Z3**2

# ---------------------------------------------------------------------------
# Truncated Laurent-series arithmetic in x
# ---------------------------------------------------------------------------
# A series is represented as a dict {degree: coefficient}.


def clean_series(s):
    out = {}
    for k, v in s.items():
        vv = sp.expand(v)
        if vv != 0:
            out[k] = vv
    return out



def add_series(s, t, min_deg=-1, max_deg=MAX_X_ORDER):
    out = dict(s)
    for k, v in t.items():
        if min_deg <= k <= max_deg:
            out[k] = sp.expand(out.get(k, 0) + v)
    return clean_series(out)



def mul_series(s, t, min_deg=-1, max_deg=MAX_X_ORDER):
    out = {}
    for i, ci in s.items():
        for j, cj in t.items():
            k = i + j
            if min_deg <= k <= max_deg:
                out[k] = sp.expand(out.get(k, 0) + ci * cj)
    return clean_series(out)



def scale_series(s, c, min_deg=-1, max_deg=MAX_X_ORDER):
    return clean_series({k: sp.expand(c * v) for k, v in s.items() if min_deg <= k <= max_deg})



def shift_series(s, shift, min_deg=-1, max_deg=MAX_X_ORDER):
    return clean_series({k + shift: v for k, v in s.items() if min_deg <= k + shift <= max_deg})



def reciprocal_series(f, max_deg=MAX_X_ORDER):
    """Reciprocal of a power series with constant term 1."""
    if sp.expand(f.get(0, 0) - 1) != 0:
        raise ValueError("reciprocal_series requires constant term 1")
    g = {0: sp.Integer(1)}
    for m in range(1, max_deg + 1):
        acc = sp.Integer(0)
        for k in range(1, m + 1):
            acc += f.get(k, 0) * g.get(m - k, 0)
        g[m] = sp.expand(-acc)
    return clean_series(g)



def exp_series_from_log(log_terms, max_deg=MAX_X_ORDER):
    """
    If E = exp(L) and L = sum_{m>=1} l_m x^m, then
    m E_m = sum_{k=1}^m k l_k E_{m-k}.
    """
    e = {0: sp.Integer(1)}
    for m in range(1, max_deg + 1):
        acc = sp.Integer(0)
        for k in range(1, m + 1):
            acc += k * log_terms.get(k, 0) * e.get(m - k, 0)
        e[m] = sp.expand(acc / m)
    return clean_series(e)



def series_coeff(s, deg):
    return sp.expand(s.get(deg, 0))



def series_to_expr(s):
    return sp.expand(sum(v * x**k for k, v in s.items()))
```

```
# ------------------------------------------------------------------------------
# Symbolic coefficient extraction in parameters / formal generators
# ------------------------------------------------------------------------------


def coeff_param(expr, i, j, k):
    poly = sp.Poly(sp.expand(expr), a, b, d)
    return sp.expand(poly.coeff_monomial(a**i * b**j * d**k))



def exact_degree_monomials(total_deg):
    out = []
    for i in range(total_deg + 1):
        for j in range(total_deg - i + 1):
            out.append((i, j, total_deg - i - j))
    return out



def poly_support(exprs, gens):
    monoms = set()
    for expr in exprs:
        poly = sp.Poly(sp.expand(expr), *gens)
        monoms.update(poly.monoms())
    return sorted(monoms)



def expr_vec(expr, gens, monoms):
    poly = sp.Poly(sp.expand(expr), *gens)
    out = []
    for mono in monoms:
        mon = sp.Integer(1)
        for g, e in zip(gens, mono):
            mon *= g**e
        out.append(sp.expand(poly.coeff_monomial(mon)))
    return out


# ------------------------------------------------------------------------------
# Formal factor builders
# ------------------------------------------------------------------------------


def one_shift_series(u, order=MAX_X_ORDER):
    log_terms = {}
    for m in range(1, order + 1):
        log_terms[m] = sp.expand(((-1) ** (m + 1)) * u**m * H_RAW[m - 1] / m)
    return exp_series_from_log(log_terms, order)



def half_shift_series(u, order=MAX_X_ORDER):
    log_terms = {}
    for m in range(1, order + 1):
        log_terms[m] = sp.expand(((-1) ** (m + 1)) * (2 * u) ** m * O_RAW[m - 1] / m)
    return exp_series_from_log(log_terms, order)



def zero_shift_series(u, order=MAX_X_ORDER):
    log_terms = {}
    for m in range(1, order + 1):
        log_terms[m] = sp.expand(((-1) ** (m + 1)) * u**m * (H_RAW[m - 1] - n ** (-m)) / m)
    e = exp_series_from_log(log_terms, order)
    return shift_series(scale_series(e, u / n), 1, min_deg=0, max_deg=order)



def gamma_log_series(kind, slope, order=MAX_X_ORDER):
    out = {}
    if slope == 0:
        return out
    if kind == "one":
        for m in range(2, order + 1):
            out[m] = sp.expand(((-1) ** m) * slope**m * ZETA[m] / m)
        return out
    if kind == "half":
        for m in range(2, order + 1):
            out[m] = sp.expand(((-1) ** m) * (2**m - 1) * slope**m * ZETA[m] / m)
        return out
    raise ValueError(f"Unknown gamma kind: {kind}")


# ------------------------------------------------------------------------------
# Proposition 10 and 11 recipes
# ------------------------------------------------------------------------------

PROP10_RECIPE = {
    "name": "Proposition10",
    "numerator": [
        {"kind": "zero", "slope": 2 * a},
        {"kind": "zero", "slope": 2 * b},
        {"kind": "one",  "slope": -2 * b},
        {"kind": "zero", "slope": a - d},
    ],
    "denominator": [
        {"kind": "one",  "slope": a - b},
```

```
        {"kind": "half", "slope": a + b},
        {"kind": "one",  "slope": 2 * d},
    ],
    "gamma_num": [
        {"kind": "one",  "slope": a - b},
        {"kind": "half", "slope": a + b},
        {"kind": "one",  "slope": d},
        {"kind": "half", "slope": d},
    ],
    "gamma_den": [
        {"kind": "one",  "slope": a},
        {"kind": "half", "slope": a},
        {"kind": "half", "slope": b + d},
        {"kind": "one",  "slope": -b + d},
    ],
    "regularizer": {-1: sp.Rational(3, 2) * n / a, 0: sp.Integer(1)},
    "order": 6,
}

PROP11_RECIPE = {
    "name": "Proposition11",
    "numerator": [
        {"kind": "zero", "slope": a},
        {"kind": "one",  "slope": 2 * b},
        {"kind": "zero", "slope": 2 * d},
        {"kind": "zero", "slope": 2 * a - 2 * b - 2 * d},
    ],
    "denominator": [
        {"kind": "one",  "slope": a - d},
        {"kind": "one",  "slope": b + d},
        {"kind": "half", "slope": a - b},
    ],
    "gamma_num": [
        {"kind": "half", "slope": 0},
        {"kind": "half", "slope": a - b},
        {"kind": "one",  "slope": a - d},
        {"kind": "one",  "slope": b + d},
    ],
    "gamma_den": [
        {"kind": "one",  "slope": a},
        {"kind": "one",  "slope": b},
        {"kind": "half", "slope": d},
        {"kind": "half", "slope": a - b - d},
    ],
    "regularizer": {-1: sp.Rational(3, 2) * n / a, 0: sp.Integer(1)},
    "order": 6,
}

# ---------------------------------------------------------------------------
# Kernel builders
# ---------------------------------------------------------------------------


def build_shift(spec, order):
    kind = spec["kind"]
    slope = spec["slope"]
    if kind == "one":
        return one_shift_series(slope, order)
    if kind == "half":
        return half_shift_series(slope, order)
    if kind == "zero":
        return zero_shift_series(slope, order)
    raise ValueError(f"Unknown shift kind: {kind}")



def build_series_kernel(recipe):
    order = recipe["order"]
    s = dict(recipe["regularizer"])
    for spec in recipe["numerator"]:
        s = mul_series(s, build_shift(spec, order), min_deg=-1, max_deg=order)
    for spec in recipe["denominator"]:
        inv = reciprocal_series(build_shift(spec, order), order)
        s = mul_series(s, inv, min_deg=-1, max_deg=order)
    return clean_series(s)



def build_gamma_kernel(recipe):
    order = recipe["order"]

    # Balance checks for the omitted linear terms.
    one_num = sum((g["slope"] for g in recipe["gamma_num"] if g["kind"] == "one"), sp.Integer(0))
    one_den = sum((g["slope"] for g in recipe["gamma_den"] if g["kind"] == "one"), sp.Integer(0))
    half_num = sum((g["slope"] for g in recipe["gamma_num"] if g["kind"] == "half"), sp.Integer(0))
    half_den = sum((g["slope"] for g in recipe["gamma_den"] if g["kind"] == "half"), sp.Integer(0))
    if sp.simplify(one_num - one_den) != 0:
        print(f"[warning] {recipe['name']}: one-base balance failed: {sp.simplify(one_num - one_den)}")
    if sp.simplify(half_num - half_den) != 0:
        print(f"[warning] {recipe['name']}: half-base balance failed: {sp.simplify(half_num - half_den)}")

    log_s = {}
    for g in recipe["gamma_num"]:
        log_s = add_series(log_s, gamma_log_series(g["kind"], g["slope"], order), min_deg=1, max_deg=order)
    for g in recipe["gamma_den"]:
        log_s = add_series(log_s, scale_series(gamma_log_series(g["kind"], g["slope"], order), -1, 1, order), min_deg=1
    return exp_series_from_log(log_s, order)
```

```
# ------------------------------------------------------------------------------
# Row extraction and relation solving
# ------------------------------------------------------------------------------


def extract_rows(recipe, x_degree=6, total_param_deg=6):
    s_coeff = series_coeff(build_series_kernel(recipe), x_degree)
    g_coeff = series_coeff(build_gamma_kernel(recipe), x_degree)
    rows = []
    for i, j, k in exact_degree_monomials(total_param_deg):
        lhs = coeff_param(s_coeff, i, j, k)
        rhs = coeff_param(g_coeff, i, j, k)
        if lhs == 0 and rhs == 0:
            continue
        rows.append((f"{recipe['name']} : [a^{i} b^{j} d^{k}] [x^{x_degree}]", sp.expand(lhs), sp.expand(rhs)))
    return rows



def solve_exact_relation(rows):
    if not rows:
        return False, None

    lhs_support = poly_support([r[1] for r in rows] + [TARGET_LHS], SERIES_GENS)
    rhs_support = poly_support([r[2] for r in rows] + [TARGET_RHS], ZETA_GENS)

    mat_rows = []
    rhs_vec = []

    for mono in lhs_support:
        mat_rows.append([expr_vec(r[1], SERIES_GENS, [mono])[0] for r in rows])
        rhs_vec.append(expr_vec(TARGET_LHS, SERIES_GENS, [mono])[0])

    for mono in rhs_support:
        mat_rows.append([expr_vec(r[2], ZETA_GENS, [mono])[0] for r in rows])
        rhs_vec.append(expr_vec(TARGET_RHS, ZETA_GENS, [mono])[0])

    M = sp.Matrix(mat_rows)
    v = sp.Matrix(rhs_vec)
    sol = sp.linsolve((M, v))
    if not sol:
        return False, None
    return True, list(next(iter(sol)))


# ------------------------------------------------------------------------------
# Main
# ------------------------------------------------------------------------------


def main():
    print("Proposition 10 / 11 exact row extractor (fast SymPy version)")
    print("=" * 70)
    print(f"SymPy version: {sp.__version__}")
    print(f"x-order: {MAX_X_ORDER}")
    print()

    all_rows = []

    for recipe in (PROP10_RECIPE, PROP11_RECIPE):
        print(f"--- {recipe['name']} ---")
        s6 = series_coeff(build_series_kernel(recipe), 6)
        g6 = series_coeff(build_gamma_kernel(recipe), 6)
        print("Computed [x^6] on both sides.")
        print()

        rows = extract_rows(recipe, 6, 6)
        print(f"Extracted {len(rows)} nonzero degree-6 rows.")
        for label, lhs, rhs in rows[:PRINT_MAX_ROWS_PER_RECIPE]:
            lhs_mon = len(sp.Poly(sp.expand(lhs), a, b, d).terms()) if lhs != 0 else 0
            rhs_mon = len(sp.Poly(sp.expand(rhs), a, b, d).terms()) if rhs != 0 else 0
            print(label)
            print(f"  lhs monomials in (a,b,d): {lhs_mon}")
            print(f"  rhs monomials in (a,b,d): {rhs_mon}")
            print()
        if len(rows) > PRINT_MAX_ROWS_PER_RECIPE:
            print(f"... {len(rows) - PRINT_MAX_ROWS_PER_RECIPE} more rows omitted on screen.\n")
        all_rows.extend(rows)

    if WRITE_ALL_ROWS:
        lines = []
        for label, lhs, rhs in all_rows:
            lines.append(label)
            lines.append(f"lhs = {sp.srepr(lhs)}")
            lines.append(f"rhs = {sp.srepr(rhs)}")
            lines.append("")
        ROWS_DUMP.write_text("\n".join(lines), encoding="utf-8")
        print(f"Wrote full row dump to {ROWS_DUMP.resolve()}")
        print()

    print("Target relation:")
    print(f"  lhs = {TARGET_LHS}")
    print(f"  rhs = {TARGET_RHS}")
    print()

    if TRY_SOLVE:
        print("Attempting exact relation search...")
        ok, coeffs = solve_exact_relation(all_rows)
        if ok and coeffs is not None:
```

```
            nz = [(all_rows[i][0], sp.simplify(c)) for i, c in enumerate(coeffs) if sp.simplify(c) != 0]
            print("Found an exact relation.")
            print(f"Nonzero coefficients: {len(nz)}")
            for label, c in nz:
                print(f"  ({c}) * [{label}]")

            tau_syms = sorted(
                {s for c in coeffs for s in getattr(c, "free_symbols", set()) if s.name.startswith("tau")},
                key=lambda s: s.name,
            )
            if tau_syms:
                particular_subs = {t: sp.Integer(0) for t in tau_syms}
                particular = [sp.simplify(c.subs(particular_subs)) for c in coeffs]
                nz_part = [
                    (all_rows[i][0], sp.simplify(c))
                    for i, c in enumerate(particular)
                    if sp.simplify(c) != 0
                ]
                print()
                print(f"Particular certificate (set all tau_i = 0): {len(nz_part)} nonzero coefficients")
                for label, c in nz_part:
                    print(f"  ({c}) * [{label}]")
            else:
                particular = [sp.simplify(c) for c in coeffs]
                nz_part = [
                    (all_rows[i][0], sp.simplify(c))
                    for i, c in enumerate(particular)
                    if sp.simplify(c) != 0
                ]
                print()
                print(f"Particular certificate (unique solution): {len(nz_part)} nonzero coefficients")
                for label, c in nz_part:
                    print(f"  ({c}) * [{label}]")

            # Verify the chosen particular certificate explicitly.
            res_lhs = sp.Integer(0)
            res_rhs = sp.Integer(0)
            for (_, row_lhs, row_rhs), c in zip(all_rows, particular):
                c = sp.simplify(c)
                if c != 0:
                    res_lhs += sp.expand(c * row_lhs)
                    res_rhs += sp.expand(c * row_rhs)

            res_lhs = sp.simplify(sp.expand(res_lhs - TARGET_LHS))
            res_rhs = sp.simplify(sp.expand(res_rhs - TARGET_RHS))

            print()
            print("Residual check:")
            print(f"  lhs residual = {res_lhs}")
            print(f"  rhs residual = {res_rhs}")
        else:
            print("No exact relation found with the currently extracted rows.")
            print("The current row set / basis is still insufficient.")
    else:
        print("Skipping exact solve.")


if __name__ == "__main__":
    main()
```