Challenge 05: Positive Geometry for Gravity

Pure Thought AI Challenge 05

Pure Thought AI Challenges Project

January 18, 2026

Abstract

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

Contents

Domain: Quantum Gravity Particle Physics
Difficulty: High
Timeline: 9-12 months
Prerequisites: Scattering amplitudes, algebraic geometry, on-shell methods

---

0.1 Problem Statement

0.1.1 Scientific Context

The amplituhedron program revealed that planar N=4 super-Yang-Mills scattering amplitudes can be computed as canonical differential forms on positive geometries—polytopes in kinematic space where all physical quantities are manifestly positive. This geometric reformulation exposes hidden structures invisible in Feynman diagrams.

0.1.2 The Core Question

Do analogous positive-geometry structures exist for (super)gravity amplitudes, or are there fundamental obstructions unique to gravity?

Specifically:

- Can gravity loop integrands be expressed as canonical forms on positive geometries?
- What is the correct geometric object (if any): Grassmannian, polytope, tropical variety?
- Does the double-copy structure of gravity → YM × YM have a geometric interpretation?

0.1.3 Why This Matters

- **Hidden mathematical structure:** Would reveal deeper organization of quantum gravity
- **Computational power:** Positive geometries bypass traditional integral reduction
- **UV properties:** Geometric constraints might explain gravity's UV behavior
- **Impossibility is also progress:** Rigorous no-go theorems constrain what structures quantum gravity can have

---

0.2 Mathematical Formulation

0.2.1 Problem Definition

Input: Graviton scattering amplitude at L loops, n external particles, specific helicity configuration

Goal: Find a positive geometry G and canonical form such that:

```
Integrand_L(      , ...,   _L  ;  k   , ..., k_n) =   _canonical  (G) /
   vol(redundancies)
```

where:

- G is a geometric object (polytope, Grassmannian variety, etc.)

- ¿$anonicalisauniquetop - formonGdeterminedbyresiduetheorems$

- Integration over G via residues recovers the amplitude

Requirements for positive geometry:

- **Positivity:** All physical quantities (momentum invariants) are positive in the interior of G
- **Boundary structure:** Codimension-1 boundaries factorization channels
- **Canonical form:** is uniquely determined by:

```
    R e s _ G      =    _boundary
```

(recursive definition from lower-dimensional boundaries)

- **Symbol structure:** Amplitude should have d log form:

```
   Symbol(A) =     c_i d log            d log            ...      d
      log  _n
```

where ¡$arethealphabetletters(coordinatesonG)$

0.2.2 Specific Test Cases

1-loop 4-graviton MHV amplitude:

```
    M          =         d     /(2  )     N(    , k_i) /
    [      (    - k  )  (    - k  - k  )  (    + k  )  ]
```

Questions:

- Does the integrand have pure d log form?
- What is the symbol alphabet?
- Can it be written as canonical form on a geometry?

2-loop test:

- N=8 supergravity 4-point amplitude
- Known to be UV finite at 2 loops
- Does geometric structure explain cancellations?

0.2.3 Certificate of Correctness

If positive geometry exists:

- Explicit description of geometry G (inequalities defining polytope, or Grassmannian parametrization)
- Canonical form written explicitly
- **Verification:** Compute residues on all boundaries, verify they match factorization
- **Verification:** Integrate and recover known amplitude
- Symbol integrability: verify d(Symbol) = 0

If no positive geometry exists:

- Obstruction certificate: show symbol alphabet violates positive geometry requirements
- E.g., letters that cannot come from momentum invariants
- Or: integrability violations
- Or: branch cut structure incompatible with boundaries

0.3 Implementation Approach

0.3.1 Phase 1: Amplitude Computation via Unitarity (Months 1-3)

Build on-shell infrastructure:

```
import sympy as sp
from sympy import symbols, expand, simplify

def spinor_helicity_formalism(momenta, helicities):
    """
    Represent momenta as spinor products    ij    , [ij]
    """
    angle_brackets = {}  #    ij    =  _i ^    _ {j  }
    square_brackets = {}  # [ij] =   _  {i,    }     ^{    }_j

    for i, j in combinations(range(len(momenta)), 2):
        angle_brackets[i,j] = compute_angle_bracket(momenta[i],
            momenta[j])
        square_brackets[i,j] = compute_square_bracket(momenta[i],
            momenta[j])

    return angle_brackets, square_brackets

def generalized_unitarity_cuts(tree_amplitudes, loop_order):
    """
    Compute loop integrand by gluing tree amplitudes

    For 1-loop: cut 4 propagators, solve for loop momentum
    """
```

```
    cuts = []
    for cut_configuration in generate_maximal_cuts(loop_order):
        # Solve cut conditions:    _i    = 0 for cut propagators
        loop_solution = solve_cut_equations(cut_configuration)

        # Evaluate product of tree amplitudes
        cut_amplitude = evaluate_tree_product(tree_amplitudes,
           loop_solution)

        cuts.append((cut_configuration, cut_amplitude))

    # Reconstruct integrand from cuts
    integrand = reconstruct_from_cuts(cuts)
    return integrand
```

Validate with known results:

- **1-loop 4-gluon YM:** Verify reproduces known integral basis
- **1-loop 4-graviton:** Compute via double copy

```
   M_gravity = M_YM^left    M_YM^right  # Schematic
```

0.3.2 Phase 2: Symbol Extraction (Months 3-5)

Compute symbol alphabet:
The symbol of a polylogarithmic function is:

```
Symbol(Li_n(z)) = z     Symbol(Li_{n-1}(z))
Symbol(log(z)) = z
```

```
def extract_symbol(amplitude, loop_order):
    """
    Compute symbol: multi-linear map
    A                          ...       _n
    """
    # Express amplitude in terms of classical polylogarithms
    poly_expansion = expand_in_polylogs(amplitude)

    # Extract symbol recursively
    symbol_entries = []
    for term in poly_expansion:
        symbol_entries.append(compute_symbol_recursive(term))

    # Collect all letters that appear
    alphabet = set()
    for entry in symbol_entries:
        alphabet.update(extract_letters(entry))

    return alphabet, symbol_entries

def check_integrability(symbol):
    """
    Verify d(Symbol) = 0 (integrability condition)

```

```
    d maps (n-1)-forms to n-forms:
    d(           ...       _n ) =  _i  (-1)^{i-1}
                ...     d_i     ...   _n
    """
    d_symbol = apply_differential(symbol)
    return simplify(d_symbol) == 0
```

Test symbol properties:

- **First-entry condition:** First entry should be related to leading singularities
- **Adjacency:** Adjacent entries should satisfy certain restrictions
- **Integrability:** d(Symbol) = 0
- **Branch cuts:** Symbol must encode correct analytic structure

0.3.3 Phase 3: Geometry Hunting (Months 5-8)

Approach 1: Momentum twistors (for amplituhedron-like geometries)

```
def momentum_twistor_transform(external_momenta):
    """
    Map momenta to momentum twistor space

    Z_i^A = ( _i ^  ,  _ {i,    })
    where  _ {i+1} =  _i  +  _i    _i
    """
    Z = []
    mu = [0, 0]  #  _0  = 0

    for i, p in enumerate(external_momenta):
        lambda_i, lambda_tilde_i = spinor_decomposition(p)
        mu_next = mu + lambda_i @ lambda_tilde_i
        Z.append(concatenate([lambda_i, mu_next]))
        mu = mu_next

    return Z

def test_grassmannian_geometry(Z_twistors, loop_momenta):
    """
    Check if integrand is canonical form on Gr(k, n)
    """
    # Parametrize loop momentum in twistor space
    # Check if integrand =  _can  / vol(redundancy)
    pass
```

Approach 2: Polytope from symbol alphabet

```
def construct_polytope_from_alphabet(alphabet):
    """
    If alphabet = {     , ...,  _m }, try to identify polytope
    where  _i  > 0 defines interior
    """
    # Define polytope P = {x :  _i (x) > 0 for all i}

    # Check if this is a valid positive geometry:
```

```
    # 1. Boundaries at  _i  = 0 correspond to factorization
    # 2. Vertices/edges have physical interpretation

    inequalities = [alpha_i > 0 for alpha_i in alphabet]
    polytope = solve_polytope_vertices(inequalities)

    return polytope

def verify_factorization_at_boundaries(polytope, amplitude):
    """
    Check codim-1 boundaries     physical factorization channels
    """
    boundaries = enumerate_facets(polytope)

    for facet in boundaries:
        # Approach boundary: one  _i      0
        residue = compute_residue_at_facet(amplitude, facet)

        # Should factor into lower-point amplitudes
        expected_factorization = compute_factorization_limit(amplitude,
            facet)

        assert is_equivalent(residue, expected_factorization)
```

Approach 3: Tropical geometry

```
def tropical_limit(symbol_alphabet):
    """
    Take tropical (log) limit of kinematic space

     _i      e^{t x_i} as t
    Scattering equations     tropical curves
    """
    tropical_variety = []
    for alpha in symbol_alphabet:
        tropical_variety.append(take_log(alpha))

    # Tropical variety = combinatorial shadow of positive geometry
    return tropical_variety
```

0.3.4 Phase 4: Canonical Form Construction (Months 8-10)

If geometry G is identified:

```
def construct_canonical_form(geometry):
    """
       is unique form determined by:
    - Top-dimensional on G
    - Satisfies  R e s _ G     =   _boundary   (recursive)
    """
    # Start from top dimension
    dim = geometry.dimension

    # Impose boundary conditions
    boundaries = geometry.get_boundaries(codim=1)
    boundary_forms = [construct_canonical_form(B) for B in boundaries]
```

```
    # Solve for    such that residues match boundary forms
    Omega = solve_recursive_residue_equations(geometry, boundary_forms)

    return Omega

def verify_canonical_form(Omega, geometry, integrand):
    """
    Check:
    1.    has correct singularities
    2.   _G      reproduces amplitude (via residue theorem)
    """
    # Compute integral via sum of residues
    residue_sum = sum(compute_all_residues(Omega, geometry))

    # Compare to direct integration of integrand
    direct_integral = integrate_amplitude(integrand)

    assert is_close(residue_sum, direct_integral, rtol=1e-10)
```

0.3.5 Phase 5: No-Go Theorems (if no geometry found) (Months 10-12)

Prove obstructions:

```
def prove_alphabet_obstruction(symbol_alphabet):
    """
    Show alphabet cannot come from positive kinematics

    E.g., if alphabet contains (s+t), this is NOT positive
    in physical region where s,t,u < 0 with s+t+u=0
    """
    # Check each letter for positivity in physical region
    for letter in symbol_alphabet:
        if not is_always_positive_in_physical_region(letter):
            return f"Obstruction: {letter} changes sign"

    # Check for branch cut incompatibilities
    branch_cuts = extract_branch_cut_structure(symbol_alphabet)
    if not compatible_with_boundary_structure(branch_cuts):
        return "Branch cut obstruction"

    return "No obstruction found (yet)"

def check_cluster_algebra_structure(alphabet):
    """
    Positive geometries often have cluster algebra structure
    Test if alphabet closes under mutations
    """
    from cluster_algebra import ClusterAlgebra

    cluster = ClusterAlgebra(alphabet)
    if not cluster.is_finite_type():
        return "Infinite cluster algebrano finite positive geometry"

    return cluster
```

0.4 Example Starting Prompt

```
I need you to investigate whether 1-loop graviton scattering amplitudes
have a positive-geometry structure.

GOAL: Determine if the 1-loop 4-graviton MHV amplitude can be expressed
as a canonical form on a positive geometry.

PHASE 1 - Compute the integrand:
1. Use generalized unitarity to compute the 1-loop 4-graviton MHV
   integrand.
   You'll need to:
   - Implement 3-point and 4-point tree-level graviton amplitudes
   - Use BCFW recursion or direct Feynman rules
   - Compute maximal cuts (cut 4 propagators)

2. Express the result in terms of loop momentum     and external
   momenta k_i.

3. Verify gauge invariance and correct dimensions.

PHASE 2 - Symbol alphabet:
4. Integrate the loop integrand (you can use known results or numerical
   integration for specific kinematics).

5. Express the result in terms of classical polylogarithms Li_n.

6. Extract the symbol: iteratively peel off d log layers.

7. Collect the symbol alphabetall letters that appear.

PHASE 3 - Test positive geometry:
8. Check if all alphabet letters can be expressed as ratios of
   momentum invariants that are positive in some physical region.

9. Test integrability: compute d(Symbol) and verify it equals zero.

10. Check adjacency conditions on symbol entries.

PHASE 4 - Construct geometry (if possible):
11. If letters look promising, try to identify a polytope where
    letter_i > 0 defines the interior.

12. Check if boundaries correspond to factorization channels.

13. Construct the canonical form and verify residue theorems.

PHASE 5 - Or prove obstruction:
14. If letters cannot be made simultaneously positive, document the
    obstruction.

15. Check if the obstruction is fundamental to gravity or specific to
    this amplitude.
```

```
Please proceed step-by-step and verify each calculation independently.
Use high-precision arithmetic and cross-check against known results
in the literature.
```

0.5 Success Criteria

0.5.1 Minimum Viable Result (9 months)

1-loop amplitude computed:

- 4-graviton MHV integrand via generalized unitarity
- Symbol extracted and alphabet documented
- Integrability verified

Geometry test completed:

- Positive geometry either found or
- Clear obstruction identified and proven

One definitive result:

- Either: canonical form on explicit geometry
- Or: rigorous no-go theorem with certificate

0.5.2 Strong Result (12 months)

Multiple amplitudes analyzed:

- 1-loop 4-graviton in multiple helicity configurations
- 1-loop 5-graviton
- One 2-loop amplitude

Pattern identified:

- If geometries exist: unified description across helicities
- If obstructed: systematic classification of obstructions

Double-copy structure:

- Geometric interpretation of gravity = YM $\times$ YM

0.5.3 Publication-Quality Result (12+ months)

Comprehensive theory:

- Complete characterization of when positive geometries exist
- If yes: construction algorithm for arbitrary amplitudes
- If no: fundamental theorem explaining obstruction

Formal verification:

- All symbol integrability checks mechanized
- Residue calculations verified symbolically
- Lean formalization of key theorems

Novel insights:

- New computational methods, or
- Deep structural understanding of gravity's UV properties

0.6 Verification Protocol

0.6.1 Automated Checks

```python
def verify_positive_geometry_claim(integrand, geometry, canonical_form):
    """
    Comprehensive verification suite
    """
    # 1. Verify integrand is computed correctly
    assert check_unitarity_cuts(integrand)
    assert check_gauge_invariance(integrand)

    # 2. Verify symbol extraction
    symbol = extract_symbol(integrand)
    assert check_integrability(symbol)

    # 3. Verify geometry is positive
    alphabet = symbol.letters()
    for x in sample_physical_region(n=1000):
        for letter in alphabet:
            assert letter.evaluate(x) > 0  # Must be positive

    # 4. Verify canonical form
    boundaries = geometry.boundaries()
    for boundary in boundaries:
        residue_calculated = compute_residue(canonical_form, boundary)
        residue_expected = boundary.canonical_form()
        assert is_close(residue_calculated, residue_expected)

```

```
    # 5. Verify integration
    integral_via_residues = sum_all_residues(canonical_form,
        geometry)
    integral_direct = integrate(integrand)
    assert is_close(integral_via_residues, integral_direct,
        rtol=1e-8)

    return "GEOMETRY VERIFIED"

def verify_obstruction_claim(symbol_alphabet,
    obstruction_certificate):
    """
    Verify that obstruction proof is valid
    """
    if obstruction_certificate.type == "sign_change":
        # Certificate shows letter changes sign in physical region
        letter = obstruction_certificate.letter
        points = obstruction_certificate.points

        assert letter.evaluate(points[0]) > 0
        assert letter.evaluate(points[1]) < 0
        assert both_in_physical_region(points[0], points[1])

    elif obstruction_certificate.type == "branch_cut":
        # Verify branch cut incompatibility
        assert not is_compatible_with_boundaries(
            obstruction_certificate.branch_structure,
            geometry_boundaries
        )

    return "OBSTRUCTION VERIFIED"
```

0.6.2 Exported Artifacts

- **Integrand file:** `integrand`$1loop$`4grav`$_M HV.m (Mathematica format)$
- Explicit expression in terms of , k_i

- Human-readable and machine-parseable
- **Symbol data:** `symbol`$1loop$`4grav.json`

```
{
  "alphabet": ["  s    ", "  s    ", "  s    ", ...],
  "entries": [
    ["  s    ", "  s    "],
    ["  s    ", "  s    "],
    ...
  ],
  "integrability_check": "passed"
}
```

- **Geometry description:** `geometry`$_4 grav.poly (if found)$
- Inequalities defining polytope, or

- Grassmannian parametrization
- **Obstruction certificate:** `obstruction.proof` (if no geometry)
- Formal proof that no positive geometry exists
- Verifiable independently

0.7 Common Pitfalls How to Avoid Them

0.7.1 Incomplete Symbol Extraction

Problem: Missing transcendental weight contributions
Solution: Cross-check against known analytic results; verify weight consistency

0.7.2 False Positive Geometries

Problem: Geometry works for special kinematics but fails generically
Solution: Test on dense grid in kinematic space; verify for multiple helicity configurations

0.7.3 Branch Cut Misidentification

Problem: Confusing logarithmic branch cuts with physical discontinuities
Solution: Carefully track i prescription; verify unitarity cuts independently

0.7.4 Numerical Precision Loss

Problem: Claiming obstruction due to numerical errors
Solution: Use exact arithmetic (sympy) for symbol; arbitrary precision for integrals

---

0.8 Resources References

0.8.1 Essential Papers

- Arkani-Hamed et al. (2016): *Grassmannian Geometry of Scattering Amplitudes*
- Bern, Dixon, Kosower (1994): *One-Loop Amplitudes for ee to Four Partons*
- Carrasco, Johansson (2011): *Generic Multiloop Methods and Application to N=4 SYM*
- Hodges (2013): *Eliminating Spurious Poles from Gauge-Theoretic Amplitudes*

0.8.2 Code Libraries

- **FiniteFlow:** For numerical evaluations with finite field methods
- **Mathematica:** For symbolic manipulations and polylog functions
- **GiNaC/sympy:** For symbol extraction and integrability checks

0.8.3 Key Concepts

- Spinor-helicity formalism
- Generalized unitarity and on-shell recursion (BCFW)
- Symbols and coproducts of polylogarithms
- Cluster algebras and positive geometries
- Double-copy construction (gravity = YM YM)

0.9 Milestone Checklist

Spinor-helicity formalism implemented and tested

Tree-level graviton amplitudes (3-pt, 4-pt) verified

Generalized unitarity code working

1-loop 4-gluon YM integrand reproduced

1-loop 4-graviton integrand computed

Loop integral evaluated (numerically or analytically)

Symbol extracted and alphabet documented

Integrability verified

Positive geometry identified OR obstruction proven

Canonical form constructed (if geometry found)

Residue theorems verified

Publication draft with proof repository

Next Steps: Begin with implementing tree-level graviton amplitudes using spinor-helicity formalism. Verify all helicity configurations before attempting loop calculations. Build robust unitarity cut infrastructure before computing integrands.