Challenge 05:
Positive Geometry for Gravity

Comprehensive Technical Report

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

Pure Thought AI Challenges
January 19, 2026

Contents

1 Executive Summary

The **amplituhedron program** revealed that scattering amplitudes in planar $\mathcal{N} = 4$ super-Yang-Mills 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 traditional Feynman diagram calculations.

Analysis Note

This challenge investigates whether analogous positive-geometry structures exist for **gravity amplitudes**. A positive answer would revolutionize our understanding of quantum gravity by revealing deep geometric structures. A negative answer—a rigorous no-go theorem—would be equally valuable, establishing fundamental differences between gauge theory and gravity at the structural level.

2 Scientific Context

2.1 The Amplituhedron Revolution

For planar $\mathcal{N} = 4$ super-Yang-Mills (SYM), the amplituhedron provides a revolutionary reformulation of scattering amplitudes:

Physical Insight

Key Features of the Amplituhedron:

1. A geometric object $\mathcal{A}_{n,k,L}$ in momentum-twistor space
2. A unique **canonical form** Ω determined by boundary structure
3. Amplitude $= \int \Omega$ (integration via residues)
4. Locality and unitarity **emerge** from geometry—they are not assumed
5. No reference to spacetime or Lagrangian needed
6. Symmetries (dual conformal, Yangian) are manifest

Definition 2.1 (Positive Geometry)**.** A **positive geometry** (G, Ω) consists of:

1. A geometric object G (polytope, Grassmannian variety, etc.)
2. A canonical form Ω uniquely determined by:
   - Ω is a top-dimensional form on G
   - $\text{Res}_{\partial G}\Omega = \Omega_{\text{boundary}}$ (recursive definition)

2.2 The Central Question

Central Research 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: Grassmannian, polytope, tropical variety?
- Does the double-copy structure (gravity = YM × YM) have a geometric interpretation?

2.3 Why This Matters

(1) Hidden Mathematical Structure: Would reveal deeper organization of quantum gravity beyond traditional perturbation theory

(2) Computational Power: Positive geometries bypass traditional integral reduction—amplitudes computed by counting faces of polytopes

(3) UV Properties: Geometric constraints might explain gravity's surprising UV behavior (cancellations invisible in Feynman diagrams)

(4) No-Go Theorems as Progress: Rigorous obstructions constrain what structures quantum gravity *can* have, guiding future research

2.4 The Double-Copy Structure

A key feature of gravity amplitudes is the **BCJ double-copy** relation:

$$M_{\text{gravity}} = A_{\text{YM}}^{\text{left}} \otimes A_{\text{YM}}^{\text{right}} \tag{1}$$

Physical Insight

Geometric Question: If Yang-Mills has the amplituhedron $\mathcal{A}_{\text{YM}}$, does gravity have:

$$\mathcal{A}_{\text{grav}} = \mathcal{A}_{\text{YM}} \times \mathcal{A}_{\text{YM}} \quad ? \tag{2}$$

This would provide a construction algorithm for gravity positive geometries.

3 Mathematical Formulation

3.1 Positive Geometry Requirements

Definition 3.1 (Positive Geometry Axioms). A positive geometry G with canonical form Ω must satisfy:

1. **Positivity:** All physical quantities are positive in the interior of G
2. **Boundary Structure:** Codimension-1 boundaries $\leftrightarrow$ factorization channels
3. **Recursive Form:** $\text{Res}_{\partial G}\Omega = \Omega_{\text{boundary}}$
4. $d\log$ **Structure:** $\Omega = \sum c_i \, d\log \alpha_1 \wedge \cdots \wedge d\log \alpha_n$

3.2 Gravity Amplitude Structure

The **1-loop 4-graviton MHV amplitude**:

$$M_4^{(1)} = \int \frac{d^4\ell}{(2\pi)^4} \frac{N(\ell, k_i)}{\ell^2(\ell - k_1)^2(\ell - k_1 - k_2)^2(\ell + k_4)^2} \tag{3}$$

The key questions are:

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

3.3 Spinor-Helicity Formalism

External momenta are decomposed using spinor-helicity variables:

$$p_i^{\alpha\dot{\alpha}} = \lambda_i^\alpha \tilde{\lambda}_i^{\dot{\alpha}} \tag{4}$$

Define spinor brackets:

$$\langle ij \rangle = \epsilon_{\alpha\beta} \lambda_i^\alpha \lambda_j^\beta \tag{5}$$

$$[ij] = \epsilon_{\dot{\alpha}\dot{\beta}} \tilde{\lambda}_i^{\dot{\alpha}} \tilde{\lambda}_j^{\dot{\beta}} \tag{6}$$

Mandelstam invariants:

$$s_{ij} = (p_i + p_j)^2 = \langle ij \rangle [ji] \tag{7}$$

3.4 Momentum Twistors

For the amplituhedron construction, we use **momentum twistors**:

$$Z_i^A = (\lambda_i^\alpha, \mu_{i,\dot{\alpha}}) \tag{8}$$

where $\mu_{i+1} = \mu_i + \lambda_i \tilde{\lambda}_i$.

Physical Insight

In momentum-twistor space, the amplituhedron for planar $\mathcal{N} = 4$ SYM is defined by positivity conditions on determinants:

$$\langle Z_{i_1} Z_{i_2} Z_{i_3} Z_{i_4} \rangle > 0 \quad \text{for appropriate sequences} \tag{9}$$

3.5 Symbol of Polylogarithmic Functions

The **symbol** is a linear map from transcendental functions to tensor products:

Definition 3.2 (Symbol)**.**

$$\text{Symbol}(\log z) = z \tag{10}$$

$$\text{Symbol}(\text{Li}_n(z)) = z \otimes \text{Symbol}(\text{Li}_{n-1}(z)) \tag{11}$$

For an amplitude A with polylogarithmic structure:

$$\text{Symbol}(A) = \sum_i c_i \, \alpha_{i_1} \otimes \alpha_{i_2} \otimes \cdots \otimes \alpha_{i_n} \tag{12}$$

where $\{\alpha_i\}$ is the **symbol alphabet**.

Theorem 3.1 (Integrability)**.** A valid symbol satisfies $d(\text{Symbol}) = 0$, which translates to specific constraints on adjacent entries.

3.6 Certificate Specification

If positive geometry exists:

- Explicit description of G (inequalities or Grassmannian parametrization)
- Canonical form Ω written explicitly
- **Verification:** Residues on all boundaries match factorization
- **Verification:** Integration of Ω recovers the amplitude
- Symbol integrability: $d(\text{Symbol}) = 0$

If no positive geometry exists:

- Obstruction certificate showing symbol alphabet violates requirements
- Example: Letters that change sign in physical region
- Example: Integrability violations
- Example: Branch cut structure incompatible with boundaries

4 Implementation Approach

4.1 Phase 1: Amplitude Computation via Unitarity (Months 1–3)

Listing 1: Spinor-helicity infrastructure

```python
import numpy as np
from itertools import combinations
import sympy as sp

class SpinorHelicity:
    """Spinor-helicity formalism for scattering amplitudes."""

    def __init__(self, momenta):
        self.n = len(momenta)
        self.momenta = momenta
        self.lambdas = []
        self.lambda_tildes = []

        for p in momenta:
            lam, lam_tilde = self.spinor_decomposition(p)
            self.lambdas.append(lam)
            self.lambda_tildes.append(lam_tilde)

    def spinor_decomposition(self, p):
        """Decompose null momentum into spinors."""
        # p^{alpha dot{alpha}} = lambda^alpha tilde{lambda}^{dot{alpha}}
        # For massless: p^2 = det(p) = 0
        p_matrix = np.array([[p[0] + p[3], p[1] - 1j*p[2]],
                             [p[1] + 1j*p[2], p[0] - p[3]]])

        # SVD to extract spinors
        U, S, Vh = np.linalg.svd(p_matrix)
        lambda_alpha = np.sqrt(S[0]) * U[:, 0]
        lambda_tilde = np.sqrt(S[0]) * Vh[0, :]

        return lambda_alpha, lambda_tilde

    def angle_bracket(self, i, j):
        """Compute <ij> = epsilon_{alpha beta} lambda_i^alpha lambda_j^beta"""
        return (self.lambdas[i][0] * self.lambdas[j][1] -
                self.lambdas[i][1] * self.lambdas[j][0])

    def square_bracket(self, i, j):
        """Compute [ij] = epsilon_{dot{alpha} dot{beta}} tilde{lambda}_i tilde{
            lambda}_j"""
        return (self.lambda_tildes[i][0] * self.lambda_tildes[j][1] -
                self.lambda_tildes[i][1] * self.lambda_tildes[j][0])

    def mandelstam(self, i, j):
        """Compute s_{ij} = <ij>[ji]"""
        return self.angle_bracket(i, j) * self.square_bracket(j, i)


def three_graviton_amplitude(sh, helicities):
    """
    3-graviton amplitude in spinor-helicity formalism.
    M_3(1^{h1}, 2^{h2}, 3^{h3})
    """
    h1, h2, h3 = helicities

    # All-plus or all-minus vanish
    if h1 == h2 == h3:
        return 0

    # MHV: two minus, one plus
    if h1 == -2 and h2 == -2 and h3 == +2:
        return (sh.angle_bracket(0, 1) ** 6 /
                (sh.angle_bracket(1, 2) ** 2 * sh.angle_bracket(2, 0) ** 2))

    # Other configurations by permutation
    # ...

    return 0


def four_graviton_amplitude_tree(sh, helicities):
    """
    4-graviton tree amplitude.
    Uses BCFW recursion or direct formula.
    """
    # MHV amplitude: M_4(1^-, 2^-, 3^+, 4^+)
    # M_4 = <12>^8 / (<12><23><34><41> * s_{12} * s_{14})

    s12 = sh.mandelstam(0, 1)
    s14 = sh.mandelstam(0, 3)

    numerator = sh.angle_bracket(0, 1) ** 8
    denominator = (sh.angle_bracket(0, 1) * sh.angle_bracket(1, 2) *
                   sh.angle_bracket(2, 3) * sh.angle_bracket(3, 0) *
                   s12 * s14)

    return numerator / denominator
```

Listing 2: Generalized unitarity for loop amplitudes

```python
def generalized_unitarity_cuts(tree_amplitudes, loop_order, external_momenta):
    """
    Compute loop integrand by gluing tree amplitudes on cuts.

    For 1-loop: cut 4 propagators (maximal cut), solve for loop momentum.
    """
    cuts = []

    for cut_config in generate_maximal_cuts(loop_order, len(external_momenta)):
        # Cut conditions: ell_i^2 = 0 for cut propagators
        cut_propagators = cut_config['propagators']

        # Solve cut equations for loop momentum
        loop_solutions = solve_cut_equations(cut_propagators, external_momenta)

        for ell_solution in loop_solutions:
            # Evaluate product of tree amplitudes at cut
            cut_value = 1.0
            for tree_config in cut_config['trees']:
                tree_amp = evaluate_tree_amplitude(
                    tree_amplitudes,
                    tree_config['momenta'],
                    tree_config['helicities'],
                    ell_solution
                )
                cut_value *= tree_amp

            cuts.append({
                'configuration': cut_config,
                'loop_momentum': ell_solution,
                'value': cut_value
            })

    # Reconstruct full integrand from cuts
    integrand = reconstruct_integrand_from_cuts(cuts, external_momenta)

    return integrand


def solve_cut_equations(propagators, external_momenta):
    """
    Solve the on-shell conditions for cut propagators.

    For maximal cut: 4 conditions in 4D -> discrete solutions.
    """
    # Propagators: (ell - K_i)^2 = 0 for each cut
    # K_i = sum of external momenta flowing into vertex

    # Parametrize loop momentum
    ell = sp.symbols('ell_0:4')

    equations = []
    for prop in propagators:
        K = prop['momentum_sum']
        eq = sum((ell[mu] - K[mu])**2 for mu in range(4))
        equations.append(eq)

    # Solve system
    solutions = sp.solve(equations, ell)

    return solutions


def reconstruct_integrand_from_cuts(cuts, external_momenta):
    """
    Reconstruct the full loop integrand from unitarity cuts.

    Uses ansatz with master integrals and matches on cuts.
    """
    # Master integral basis for 1-loop 4-point
    # Box, triangles, bubbles

    # Ansatz: I = c_box * I_box + sum c_tri * I_tri + sum c_bub * I_bub
    coefficients = {}

    # Match coefficients by evaluating ansatz on cuts
    for cut in cuts:
        # Each maximal cut isolates one master integral
        master = identify_master_integral(cut['configuration'])
        coefficients[master] = cut['value']

    return build_integrand_from_coefficients(coefficients)
```

4.2 Phase 2: Symbol Extraction (Months 3–5)

Listing 3: Symbol computation and alphabet extraction

```
from sympy import log, polylog, symbols, expand, simplify
from sympy import tensorproduct

def extract_symbol(amplitude, loop_order):
    """
    Compute symbol: A -> alpha_1 (x) alpha_2 (x) ... (x) alpha_n

    Symbol is a multilinear map extracting logarithmic structure.
    """
    # Express amplitude in terms of classical polylogarithms
    poly_expansion = expand_in_polylogs(amplitude)

    symbol_entries = []
    for term in poly_expansion:
        entry = compute_symbol_recursive(term)
        if entry is not None:
            symbol_entries.append(entry)

    # Collect all letters that appear
    alphabet = extract_alphabet(symbol_entries)

    return alphabet, symbol_entries


def compute_symbol_recursive(expr):
    """
    Recursively compute symbol of polylogarithmic expression.

    Symbol(log(z)) = z
    Symbol(Li_n(z)) = z (x) Symbol(Li_{n-1}(z))
    Symbol(f * g) = Symbol(f) + Symbol(g)  (for products)
    """
    if expr.is_number:
        return None  # Rational numbers have trivial symbol

    if expr.func == log:
        arg = expr.args[0]
        return TensorEntry([arg])

    if expr.func == polylog:
        n, z = expr.args
        if n == 1:
            # Li_1(z) = -log(1-z)
```

```python
            return TensorEntry([1 - z])
        else:
            # Li_n(z) = z (x) Symbol(Li_{n-1}(z))
            lower_symbol = compute_symbol_recursive(polylog(n-1, z))
            return TensorEntry([z] + lower_symbol.entries)

    # Handle sums and products
    if expr.is_Add:
        return sum_symbols([compute_symbol_recursive(arg) for arg in expr.args])

    if expr.is_Mul:
        # For products of transcendentals, need shuffle product
        return shuffle_product([compute_symbol_recursive(arg) for arg in expr.args])

    return None


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

    For tensor a1 (x) a2 (x) ... (x) an:
    d(a1 (x) ... (x) an) = sum_i (-1)^{i-1} a1 (x)...(x) d(ai) (x)...(x) an

    Integrability: d log a_i ^ d log a_{i+1} must be consistent.
    """
    for entry in symbol.entries:
        for i in range(len(entry) - 1):
            # Check adjacency condition
            a_i = entry[i]
            a_i1 = entry[i + 1]

            # d log a_i ^ d log a_{i+1} must satisfy certain relations
            if not check_adjacency_constraint(a_i, a_i1):
                return False, f"Adjacency violation at position {i}"

    return True, "Integrability verified"


def check_adjacency_constraint(a, b):
    """
    Check if adjacent symbol entries satisfy integrability.

    Specifically: sum over residues of d log a ^ d log b must vanish.
    """
    # Compute d log a ^ d log b
    # Check residue conditions

    # Simplified: check that a and b are algebraically independent
    # or satisfy known relations
    return True  # Placeholder


class TensorEntry:
    """Represents a tensor product entry in the symbol."""

    def __init__(self, entries):
        self.entries = entries
        self.weight = len(entries)

    def __add__(self, other):
        if other is None:
            return self
        # Formal sum of tensor entries
        return SymbolSum([self, other])

    def tensor(self, other):
        """Tensor product: self (x) other"""
        return TensorEntry(self.entries + other.entries)


def extract_alphabet(symbol_entries):
    """
    Extract all distinct letters appearing in the symbol.
    """
    alphabet = set()

    for entry in symbol_entries:
        if isinstance(entry, TensorEntry):
            for letter in entry.entries:
                alphabet.add(simplify(letter))

    return sorted(alphabet, key=str)
```

4.3 Phase 3: Geometry Search (Months 5–8)

Listing 4: Polytope construction from symbol alphabet

```python
import numpy as np
from scipy.spatial import ConvexHull
from sympy import symbols, solve, Poly

def construct_polytope_from_alphabet(alphabet, kinematic_vars):
    """
    If alphabet = {alpha_1, ..., alpha_m}, try to identify polytope
    where alpha_i > 0 defines the interior.

    The polytope P = {x : alpha_i(x) > 0 for all i}
    """
    # Express each letter as function of kinematic variables
    letter_functions = []
    for alpha in alphabet:
        func = express_as_kinematic_function(alpha, kinematic_vars)
        letter_functions.append(func)

    # Define polytope by inequalities
    inequalities = [(f, '>') for f in letter_functions]

    # Solve for vertices (where d-1 inequalities are equalities)
    vertices = find_polytope_vertices(inequalities, kinematic_vars)

    if vertices is None:
        return None, "No␣bounded␣polytope␣exists"

    # Construct polytope object
    polytope = Polytope(vertices, inequalities)

    return polytope, "Polytope␣constructed"


def verify_positivity_in_physical_region(alphabet, n_samples=1000):
    """
    Check if all alphabet letters can be simultaneously positive
    in the physical scattering region.
    """
    # Physical region for 2->2 scattering:
    # s > 0, t < 0, u < 0 with s + t + u = 0

    violations = []

    for _ in range(n_samples):
        # Sample physical kinematics
        s = np.random.uniform(1, 100)
        t = -np.random.uniform(0.1, s/2)
        u = -s - t

        kinematics = {'s': s, 't': t, 'u': u}

        # Evaluate each letter
        for letter in alphabet:
            value = evaluate_letter(letter, kinematics)

            if value <= 0:
                violations.append({
                    'letter': letter,
                    'kinematics': kinematics,
                    'value': value
                })

    if violations:
        return False, violations
    return True, "All␣letters␣positive␣in␣physical␣region"


def verify_factorization_at_boundaries(polytope, amplitude):
    """
    Check that codimension-1 boundaries correspond to
    physical factorization channels.
    """
    boundaries = polytope.get_facets()

    for facet in boundaries:
        # Identify which letter vanishes at this boundary
        vanishing_letter = facet.defining_inequality

        # Compute residue of amplitude at this boundary
        residue = compute_residue_at_facet(amplitude, facet)

        # Check if residue factorizes as lower-point amplitudes
        expected = compute_factorization_limit(amplitude, vanishing_letter)

        if not is_equivalent(residue, expected):
            return False, f"Factorization␣fails␣at␣{vanishing_letter}"

    return True, "All␣boundaries␣match␣factorization␣channels"


class Polytope:
    """Represents a convex polytope in kinematic space."""

    def __init__(self, vertices, inequalities):
        self.vertices = vertices
        self.inequalities = inequalities
        self.dimension = len(vertices[0]) if vertices else 0

    def get_facets(self):
```

```python
        """Return codimension-1 faces (facets)."""
        facets = []
        for ineq in self.inequalities:
            facet = Facet(ineq, self)
            facets.append(facet)
        return facets

    def contains(self, point):
        """Check if point is in interior of polytope."""
        for func, direction in self.inequalities:
            value = evaluate_letter(func, point)
            if direction == '>' and value <= 0:
                return False
            if direction == '<' and value >= 0:
                return False
        return True

    def canonical_form(self):
        """
        Construct the canonical form Omega on this polytope.
        Omega is unique form with logarithmic singularities on boundaries.
        """
        return construct_canonical_form(self)
```

Listing 5: Momentum twistor geometry

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

    Z_i^A = (lambda_i^alpha, mu_{i,dot{alpha}})
    where mu_{i+1} = mu_i + lambda_i * tilde{lambda}_i
    """
    Z = []
    mu = np.zeros(2, dtype=complex)

    for i, p in enumerate(external_momenta):
        sh = SpinorHelicity([p])
        lambda_i = sh.lambdas[0]
        lambda_tilde_i = sh.lambda_tildes[0]

        # Update mu via incidence relation
        mu_next = mu + np.outer(lambda_i, lambda_tilde_i).flatten()[:2]

        # Momentum twistor
        Z_i = np.concatenate([lambda_i, mu_next])
        Z.append(Z_i)

        mu = mu_next

    return np.array(Z)


def test_grassmannian_geometry(Z_twistors, loop_momenta, k):
    """
    Check if integrand is canonical form on Gr(k, n).

    The Grassmannian Gr(k,n) parametrizes k-planes in C^n.
    Positive Grassmannian: all ordered minors positive.
    """
    n = len(Z_twistors)

    # Parametrize Gr(k, n) by k x n matrix C
    C = construct_grassmannian_parametrization(k, n)

    # Positive Grassmannian conditions
    positivity_conditions = []
    for indices in combinations(range(n), k):
        minor = compute_minor(C, indices)
        positivity_conditions.append(minor > 0)

    # Check if loop integrand matches canonical form
    canonical = grassmannian_canonical_form(C, positivity_conditions)

    return canonical


def compute_minor(matrix, indices):
    """Compute the minor of matrix using specified columns."""
    submatrix = matrix[:, list(indices)]
    return np.linalg.det(submatrix)
```

4.4 Phase 4: Canonical Form Construction (Months 8–10)

Listing 6: Canonical form construction

```python
def construct_canonical_form(geometry):
    """
    Omega is unique form determined by:
    - Top-dimensional on G
    - Satisfies Res_{boundary} Omega = Omega_boundary (recursive)
    """
    dim = geometry.dimension

    if dim == 0:
        # Point: canonical form is 1
        return 1

    # Get codimension-1 boundaries
    boundaries = geometry.get_facets()

    # Recursively construct canonical forms on boundaries
    boundary_forms = []
    for B in boundaries:
        omega_B = construct_canonical_form(B)
        boundary_forms.append((B, omega_B))

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

    return Omega


def solve_recursive_residue_equations(geometry, boundary_forms):
    """
    Find form Omega such that Res_{B} Omega = Omega_B for all boundaries B.
    """
    # Ansatz: Omega = sum_i c_i * d log alpha_i
    # where alpha_i are the defining inequalities

    inequalities = geometry.inequalities
    n = len(inequalities)

    # Build d log form
    dlog_terms = []
    for alpha, _ in inequalities:
        dlog_terms.append(f"d log({alpha})")

    # The canonical form is the wedge product
    # Omega = d log alpha_1 ^ d log alpha_2 ^ ... ^ d log alpha_n
    # with appropriate normalization

    # For a simplex: Omega = d log(alpha_1/alpha_0) ^ ... ^ d log(alpha_n/
        alpha_0)

    Omega = wedge_product(dlog_terms)

    # Verify residue conditions
    for B, omega_B in boundary_forms:
        res = compute_residue(Omega, B)
        if not is_equivalent(res, omega_B):
            raise ValueError(f"Residue mismatch at boundary {B}")

    return Omega


def verify_canonical_form(Omega, geometry, integrand):
    """
    Check:
    1. Omega has correct singularities (only on boundaries)
    2. Integration of Omega reproduces amplitude
    """
    # Check singularity structure
    singularities = find_singularities(Omega)
    boundaries = geometry.get_facets()

    for sing in singularities:
        if not any(sing.on_boundary(B) for B in boundaries):
            return False, f"Spurious singularity at {sing}"

    # Compute integral via sum of residues
    residue_sum = sum_all_residues(Omega, geometry)

    # Compare to direct integration
    direct_integral = integrate_amplitude(integrand)

    if not np.isclose(residue_sum, direct_integral, rtol=1e-8):
        return False, f"Integration mismatch: {residue_sum} vs {direct_integral
            }"

    return True, "Canonical form verified"


def sum_all_residues(Omega, geometry):
    """
    Compute amplitude by summing residues at all vertices.
    """
    vertices = geometry.get_vertices()
    total = 0

    for vertex in vertices:
        res = compute_residue_at_vertex(Omega, vertex)
        total += res

    return total
```

4.5 Phase 5: Obstruction Detection (Months 10–12)

Listing 7: Proving obstructions to positive geometry

```python
def prove_alphabet_obstruction(symbol_alphabet, kinematic_space):
    """
    Show that symbol alphabet cannot come from positive geometry.

    Obstructions:
    1. Letters that change sign in physical region
    2. Branch cut structure incompatible with boundaries
    3. Integrability violations
    """
    obstructions = []

    # Check 1: Sign changes in physical region
    for letter in symbol_alphabet:
        sign_change = find_sign_change(letter, kinematic_space)
        if sign_change:
            obstructions.append({
                'type': 'sign_change',
                'letter': letter,
                'points': sign_change
            })

    # Check 2: Branch cut compatibility
    branch_cuts = extract_branch_cut_structure(symbol_alphabet)
    if not compatible_with_positive_geometry(branch_cuts):
        obstructions.append({
            'type': 'branch_cut',
            'structure': branch_cuts
        })

    # Check 3: Cluster algebra structure
    cluster_check = check_cluster_algebra_structure(symbol_alphabet)
    if cluster_check['type'] == 'infinite':
        obstructions.append({
            'type': 'infinite_cluster',
            'details': cluster_check
        })

    if obstructions:
        return ObstructionCertificate(obstructions)
    return None


def find_sign_change(letter, kinematic_space):
    """
    Find two points in physical region where letter changes sign.
    """
    physical_region = kinematic_space.physical_region()

    # Sample points in physical region
    positive_points = []
    negative_points = []

    for point in physical_region.sample(1000):
        value = letter.evaluate(point)
        if value > 0:
            positive_points.append(point)
        elif value < 0:
            negative_points.append(point)

    if positive_points and negative_points:
        return (positive_points[0], negative_points[0])
    return None


def check_cluster_algebra_structure(alphabet):
    """
    Positive geometries often have finite cluster algebra structure.
    Test if alphabet closes under mutations.
    """
    # Initialize cluster algebra with alphabet as cluster variables
    initial_cluster = list(alphabet)

    # Perform mutations and check if new variables appear
    visited = set(initial_cluster)
    to_explore = list(initial_cluster)

    while to_explore:
        current = to_explore.pop()
        for mutation in generate_mutations(current, alphabet):
            new_var = apply_mutation(mutation)
            if new_var not in visited:
                visited.add(new_var)
                to_explore.append(new_var)

                # Check if algebra is becoming infinite
                if len(visited) > 1000:
                    return {
                        'type': 'infinite',
                        'message': 'Cluster␣algebra␣appears␣infinite'
                    }

    return {
        'type': 'finite',
        'size': len(visited),
        'variables': visited
    }


class ObstructionCertificate:
    """Machine-verifiable proof that no positive geometry exists."""

    def __init__(self, obstructions):
        self.obstructions = obstructions

    def verify(self):
        """Independently verify the obstruction claims."""
        for obs in self.obstructions:
            if obs['type'] == 'sign_change':
                # Verify that points are in physical region
                p1, p2 = obs['points']
                assert is_physical(p1), f"{p1}␣not␣in␣physical␣region"
                assert is_physical(p2), f"{p2}␣not␣in␣physical␣region"

                # Verify sign change
                letter = obs['letter']
                v1 = letter.evaluate(p1)
                v2 = letter.evaluate(p2)
                assert v1 > 0 and v2 < 0, "Sign␣change␣not␣verified"

            elif obs['type'] == 'branch_cut':
                # Verify branch cut incompatibility
                structure = obs['structure']
                assert not compatible_with_positive_geometry(structure)

        return True

    def export(self, filename):
        """Export certificate to JSON for external verification."""
        import json
        with open(filename, 'w') as f:
            json.dump({
                'type': 'obstruction_certificate',
                'obstructions': [self._serialize_obs(o) for o in self.
                    obstructions]
            }, f, indent=2)
```

5 Research Directions

5.1 Direction 1: MHV Sector Analysis

Research Direction

Focus on MHV (maximally helicity violating) amplitudes where graviton expressions are simplest:

1. Start with 4-graviton MHV amplitude
2. Extract symbol and analyze alphabet
3. Test positive geometry existence
4. Extend to 5-graviton MHV

5.2 Direction 2: Double-Copy Geometry

Research Direction

Question: If YM has amplituhedron $\mathcal{A}_{\text{YM}}$, does gravity have $\mathcal{A}_{\text{grav}} = \mathcal{A}_{\text{YM}} \times \mathcal{A}_{\text{YM}}$?
Approach:

- Study how color-kinematics duality acts geometrically
- Understand fiber product vs direct product of geometries
- Construct explicit examples at 4- and 5-point

5.3 Direction 3: Tropical Geometry

Research Direction

Take the **tropical (log) limit** of kinematic space:

- $\alpha_i \to e^{tx_i}$ as $t \to \infty$
- Scattering equations $\to$ tropical curves
- Tropical varieties are combinatorial shadows of positive geometries
- May reveal structure even when full geometry is obstructed

5.4 Direction 4: $\mathcal{N} = 8$ Supergravity

Research Direction

$\mathcal{N} = 8$ supergravity is the most supersymmetric (and best-behaved) gravity theory:

- Known to be UV finite through at least 4 loops
- Maximum supersymmetry may enable positive geometry
- Test whether UV finiteness has geometric origin

6 Success Criteria

6.1 Minimum Viable Result (9 months)

✓ 4-graviton MHV integrand computed via generalized unitarity

✓ Symbol extracted and alphabet documented

✓ Integrability verified

✓ Geometry found OR obstruction proven with certificate

6.2 Strong Result (12 months)

✓ Multiple helicity configurations analyzed

✓ Pattern identified (geometries exist OR systematic obstruction)

✓ Double-copy interpretation explored

✓ One 2-loop amplitude studied

6.3 Publication Quality (12+ months)

✓ Complete characterization of when geometries exist

✓ Formal verification of all symbol calculations

✓ Novel computational methods or structural insights

✓ Lean/Isabelle formalization of key theorems (stretch goal)

7 Verification Protocol

Listing 8: Comprehensive verification suite

```python
def verify_positive_geometry_claim(integrand, geometry, canonical_form):
    """
    Comprehensive verification for positive geometry claim.
    """
    results = {}

    # 1. Verify integrand correctness
    print("1. Verifying integrand...")
    results['unitarity'] = check_unitarity_cuts(integrand)
    results['gauge_invariance'] = check_gauge_invariance(integrand)
    assert results['unitarity'], "Unitarity␣cuts␣failed"
    assert results['gauge_invariance'], "Gauge␣invariance␣failed"

    # 2. Verify symbol extraction
    print("2.␣Verifying␣symbol...")
    symbol = extract_symbol(integrand)
    results['integrability'] = check_integrability(symbol)
    assert results['integrability'][0], results['integrability'][1]

    # 3. Verify positivity
    print("3.␣Verifying␣positivity...")
    alphabet = symbol.letters()
    for x in sample_physical_region(n=1000):
        for letter in alphabet:
            value = letter.evaluate(x)
            assert value > 0, f"Letter␣{letter}␣not␣positive␣at␣{x}"
    results['positivity'] = True

    # 4. Verify canonical form residues
    print("4.␣Verifying␣residues...")
    for boundary in geometry.get_facets():
        res_calc = compute_residue(canonical_form, boundary)
        res_exp = boundary.canonical_form()
        assert is_close(res_calc, res_exp), f"Residue␣mismatch␣at␣{boundary}"
    results['residues'] = True

    # 5. Verify integration
    print("5.␣Verifying␣integration...")
    integral_residues = sum_all_residues(canonical_form, geometry)
    integral_direct = integrate_amplitude(integrand)
    assert is_close(integral_residues, integral_direct, rtol=1e-8)
    results['integration'] = True

    print("\n===␣GEOMETRY␣VERIFIED␣===")
    return results


def verify_obstruction_claim(symbol_alphabet, certificate):
    """
    Verify that obstruction proof is valid.
    """
    print("Verifying␣obstruction␣certificate...")

    if certificate.obstructions[0]['type'] == 'sign_change':
        obs = certificate.obstructions[0]
        letter = obs['letter']
        p1, p2 = obs['points']

        # Verify both points in physical region
        assert is_in_physical_region(p1), f"{p1}␣not␣physical"
        assert is_in_physical_region(p2), f"{p2}␣not␣physical"

        # Verify sign change
        v1 = letter.evaluate(p1)
        v2 = letter.evaluate(p2)
        assert v1 > 0 and v2 < 0, "Sign␣change␣not␣verified"

        print(f"Letter␣{letter}␣changes␣sign:")
        print(f"␣␣At␣{p1}:␣value␣=␣{v1}␣>␣0")
        print(f"␣␣At␣{p2}:␣value␣=␣{v2}␣<␣0")

    print("\n===␣OBSTRUCTION␣VERIFIED␣===")
    return True
```

8 Common Pitfalls

Critical Consideration

Incomplete Symbol Extraction: Missing transcendental weight contributions invalidate the analysis. Cross-check against known analytic results; verify weight consistency at each step.

Critical Consideration

False Positive Geometries: A geometry that works for special kinematics may fail generically. Test on a dense grid throughout kinematic space; verify for multiple helicity configurations.

Critical Consideration

Branch Cut Misidentification: Confusing logarithmic branch cuts with physical discontinuities leads to incorrect conclusions. Carefully track $i\epsilon$ prescription; verify unitarity cuts independently.

Critical Consideration

Numerical Precision Loss: Claiming obstruction due to numerical errors is a common failure mode. Use exact arithmetic (SymPy) for symbol computation; arbitrary precision for numerical integrals.

Critical Consideration

Coordinate Dependence: Positivity may hold in one parametrization but not another. Test multiple coordinate systems; identify coordinate-independent obstructions.

9 Computational Resources

9.1 Software Stack

Component	Tool	Purpose
Symbolic computation	SymPy, Mathematica	Amplitude expressions
Numerical evaluation	FiniteFlow	Finite field methods
Symbol extraction	GiNaC, PolyLogTools	Polylogarithm algebra
Geometry	SageMath, polymake	Polytope computation
Verification	pytest, hypothesis	Property-based testing

9.2 Essential References

- Arkani-Hamed et al. (2016): "Grassmannian Geometry of Scattering Amplitudes"
- Bern, Dixon, Kosower (1994): "One-Loop Amplitudes for e^+e^- to Four Partons"
- Carrasco, Johansson (2011): "Generic Multiloop Methods and Application to $\mathcal{N} = 4$ SYM"
- Hodges (2013): "Eliminating Spurious Poles from Gauge-Theoretic Amplitudes"
- Arkani-Hamed, Trnka (2014): "The Amplituhedron"

10 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 (benchmark)
- ☐ 1-loop 4-graviton integrand computed
- ☐ Loop integral evaluated (numerically or analytically)
- ☐ Symbol extracted from amplitude
- ☐ Alphabet documented
- ☐ Integrability condition verified
- ☐ Positivity tested in physical region
- ☐ Positive geometry identified OR obstruction proven
- ☐ Canonical form constructed (if geometry found)
- ☐ Residue theorems verified
- ☐ Certificate exported (geometry or obstruction)
- ☐ Independent verification passed
- ☐ Publication draft with proof repository

11 Conclusion

The search for positive geometries in gravity represents one of the most tantalizing open problems at the intersection of physics and mathematics. The amplituhedron's success for $\mathcal{N} = 4$ super-Yang-Mills suggests that scattering amplitudes may have deep geometric origins—but whether this extends to gravity remains unknown.

Analysis Note

Two Possible Outcomes:

1. **Positive geometry exists:** Would reveal that quantum gravity has hidden geometric structure, potentially explaining UV properties and providing new computational methods.

2. **Fundamental obstruction:** Would establish a sharp structural difference between gauge theory and gravity, guiding future theoretical developments.

Either outcome represents significant progress in our understanding of quantum gravity.

The methodology developed here—combining amplitude computation via unitarity, symbol extraction, geometry construction, and rigorous verification—provides a systematic approach to answering this question. The machine-checkable certificates ensure that any claimed result can be independently verified, meeting the highest standards of mathematical rigor.