Challenge 06:

Nonperturbative S-matrix Bootstrap with Gravity

Comprehensive Technical Report

Domain:	Quantum Gravity & Particle Physics
Difficulty:	High
Timeline:	6–12 months
Prerequisites:	S-matrix theory, dispersion relations, partial-wave unitarity

Pure Thought AI Challenges
January 19, 2026

Contents

1 Executive Summary

The **S-matrix bootstrap** is a powerful non-perturbative approach to constraining scattering amplitudes using only fundamental physical principles: **unitarity**, **crossing symmetry**, and **analyticity**. Unlike perturbative methods that expand in small coupling constants, the bootstrap makes *no assumption about weak coupling* and produces **rigorous, model-independent bounds** on effective field theory (EFT) parameters.

Analysis Note

This challenge combines the classical S-matrix bootstrap philosophy—born in the 1960s—with modern computational techniques: semidefinite programming, high-precision numerics, and formal verification. When gravity is included, additional powerful constraints emerge from Weinberg's soft graviton theorem and Regge boundedness.

The central goal is to determine **which EFT Wilson coefficients are compatible with a consistent UV completion that includes gravity**. This directly addresses the Swampland program: identifying low-energy theories that *cannot* arise from quantum gravity.

Physical Insight

Why Non-Perturbative? Perturbative methods assume small coupling $g \ll 1$ and expand $\mathcal{M} = \mathcal{M}_0 + g\mathcal{M}_1 + g^2\mathcal{M}_2 + \cdots$. The S-matrix bootstrap instead uses only:

- Unitarity: $S^\dagger S = 1$ (exact, not perturbative)
- Crossing: Particle $\leftrightarrow$ antiparticle under $s \leftrightarrow u$
- Analyticity: Amplitudes are analytic except on physical cuts

These hold at *any* coupling strength.

2 Scientific Context and Motivation

2.1 The S-matrix Bootstrap Philosophy

The S-matrix program, pioneered by Heisenberg, Chew, Mandelstam, and others in the 1960s, sought to derive all particle physics from consistency conditions on scattering amplitudes. While the original program was superseded by QCD and the Standard Model, its core insight remains profound:

> *Physical principles alone—without reference to a specific Lagrangian—impose powerful constraints on what amplitudes are possible.*

Analysis Note

The modern revival of the S-matrix bootstrap combines these classical ideas with computational advances (convex optimization, high-precision arithmetic) and the theoretical framework of effective field theory. The result is a systematic method for deriving *rigorous bounds* on EFT parameters.

2.2 S-matrix Axioms

2.2.1 Axiom 1: Unitarity

The S-matrix must preserve probability:

$$\boxed{S^\dagger S = SS^\dagger = \mathbf{1}} \tag{1}$$

For the scattering amplitude $\mathcal{M}$, this implies the **optical theorem**:

$$\text{Im}\mathcal{M}(s, t = 0) = s\sigma_{\text{total}}(s) \geq 0 \tag{2}$$

where σ_{total} is the total cross-section.

For partial waves (defined below), unitarity becomes:

$$\text{Im}a_J(s) = \rho(s)|a_J(s)|^2 \tag{3}$$

where $\rho(s) = \sqrt{1 - 4m^2/s}$ is the phase space factor.

2.2.2 Axiom 2: Crossing Symmetry

Amplitudes involving particles and antiparticles are related by analytic continuation:

$$\boxed{A(s, t, u) = A(t, s, u) = A(u, t, s)} \tag{4}$$

For identical particles, this is a powerful constraint relating different kinematic regimes.

2.2.3 Axiom 3: Analyticity

The amplitude $\mathcal{M}(s, t)$ is **analytic** in the complex s-plane except on physical cuts:

- **Right-hand cut:** $s \geq 4m^2$ (particle production threshold)
- **Left-hand cut:** $u \geq 4m^2$ (crossed channel, maps to $s \leq 0$)

This analyticity, combined with the Schwarz reflection principle, enables powerful dispersion relations.

2.2.4 Axiom 4: Polynomial Boundedness (Regge Behavior)

At high energies (large $|s|$ with fixed t), the amplitude cannot grow faster than a polynomial:

$$|\mathcal{M}(s, t)| \leq C|s|^N \quad \text{as } |s| \to \infty \tag{5}$$

For theories with gravity, the Regge bound is:

$$\boxed{|\mathcal{M}(s, t)| \lesssim s^2 \quad \text{(gravity Regge bound)}} \tag{6}$$

This ensures convergence of dispersion relations with at most two subtractions.

Physical Insight

Why s^2 for Gravity? The graviton exchange diagram contributes $\mathcal{M} \sim s^2/t$ at tree level. The s^2 growth is the maximum consistent with causality and locality—faster growth would imply acausal propagation.

2.3 Gravity-Specific Constraints

2.3.1 Weinberg Soft Graviton Theorem

When a graviton with momentum $q \to 0$ is emitted from an n-particle amplitude:

$$\boxed{M_{n+1}(p_1, \ldots, p_n; q, \varepsilon) \xrightarrow{q\to 0} \kappa \sum_{i=1}^{n} \frac{\varepsilon_{\mu\nu} p_i^\mu p_i^\nu}{p_i \cdot q} M_n(p_1, \ldots, p_n)} \tag{7}$$

where $\kappa = \sqrt{8\pi G}$ and $\varepsilon_{\mu\nu}$ is the graviton polarization tensor.

This **universal** behavior fixes the residues at $t = 0$ and $u = 0$ poles, providing strong constraints on the amplitude.

2.3.2 Graviton Pole Structure

The graviton exchange contributes a pole at $t = 0$:

$$\mathcal{M}(s, t) \supset \frac{8\pi G\, s^2}{t} + (\text{higher poles/cuts}) \tag{8}$$

The coefficient of this pole is fixed by the equivalence principle—gravity couples universally to energy-momentum.

2.3.3 Causality and the Regge Bound

Causality (no superluminal signaling) combined with locality implies the Regge bound $\mathcal{M} \sim s^2$. Violations would allow faster-than-light communication through cleverly designed scattering experiments.

2.4 The Core Question

Central Research Question

What are the allowed regions for EFT Wilson coefficients when gravity is coupled to matter, derived purely from S-matrix axioms?
Consider the effective Lagrangian:

$$\mathcal{L} = \mathcal{L}_{\text{GR}} + \frac{1}{2}(\partial\phi)^2 - \frac{1}{2}m^2\phi^2 + \sum_{n=1}^{\infty} \frac{c_n}{\Lambda^{2n}} \mathcal{O}_{2n+4} \tag{9}$$

where $\mathcal{O}_{2n+4}$ are dimension-$(2n+4)$ operators built from ϕ and curvature.
Which values of $\{c_1, c_2, c_3, \ldots\}$ admit a UV completion consistent with unitarity, crossing, and analyticity?

2.5 Why This Matters

(1) **Swampland Constraints:** Identifies which low-energy theories *cannot* arise from quantum gravity

(2) **Model-Independent Bounds:** Results hold for *any* UV completion, not specific models

(3) **Non-Perturbative:** Valid at any coupling strength, not just weak coupling

(4) **Rigorous Mathematics:** Produces actual theorems with machine-verifiable proofs

(5) **Phenomenology:** Constrains quantum gravity corrections to Standard Model and gravity

3 Mathematical Formulation

3.1 Kinematics: Mandelstam Variables

Consider $2 \to 2$ scattering: $\phi(p_1) + \phi(p_2) \to \phi(p_3) + \phi(p_4)$.

Definition 3.1 (Mandelstam Variables)**.** The Lorentz-invariant kinematic variables are:

$$s = (p_1 + p_2)^2 = (p_3 + p_4)^2 \quad \text{(center-of-mass energy squared)} \tag{10}$$

$$t = (p_1 - p_3)^2 = (p_2 - p_4)^2 \quad \text{(momentum transfer squared)} \tag{11}$$

$$u = (p_1 - p_4)^2 = (p_2 - p_3)^2 \quad \text{(crossed channel)} \tag{12}$$

with the constraint:

$$\boxed{s + t + u = 4m^2} \tag{13}$$

The physical scattering region (where particles are on-shell with real momenta) corresponds to:

- s-channel: $s \geq 4m^2$, $t \leq 0$, $u \leq 0$
- t-channel: $t \geq 4m^2$, $s \leq 0$, $u \leq 0$
- u-channel: $u \geq 4m^2$, $s \leq 0$, $t \leq 0$

3.2 Scattering Angle

In the center-of-mass frame:

$$\cos\theta = 1 + \frac{2t}{s - 4m^2} \tag{14}$$

where θ is the scattering angle. The physical range $\theta \in [0, \pi]$ corresponds to $\cos\theta \in [-1, 1]$.

3.3 Partial-Wave Expansion

Definition 3.2 (Partial-Wave Expansion)**.** The amplitude can be expanded in Legendre polynomials:

$$\boxed{A(s,t) = 16\pi \sum_{J=0}^{\infty} (2J+1)\, a_J(s)\, P_J(\cos\theta)} \tag{15}$$

where:

- J is the angular momentum (spin)
- $a_J(s)$ are the **partial-wave amplitudes**
- $P_J(\cos\theta)$ are Legendre polynomials

The inverse relation (partial-wave projection) is:

$$a_J(s) = \frac{1}{32\pi} \int_{-1}^{1} d(\cos\theta)\, P_J(\cos\theta)\, A(s, t(\cos\theta)) \tag{16}$$

Analysis Note

The partial-wave expansion separates the energy dependence (in $a_J(s)$) from the angular dependence (in P_J). Unitarity and analyticity are most naturally stated in terms of partial waves.

3.4 Unitarity Constraints

Theorem 3.1 (Partial-Wave Unitarity)**.** Below the inelastic threshold (elastic scattering only):

$$\boxed{\mathrm{Im}a_J(s) = \rho(s)|a_J(s)|^2} \tag{17}$$

where $\rho(s) = \sqrt{1 - 4m^2/s}$ is the two-body phase space factor.

This implies the **unitarity bound**:

$$|a_J(s)| \leq \frac{1}{\rho(s)} \tag{18}$$

Parametrizing $a_J = |a_J|e^{i\delta_J}$ where δ_J is the phase shift:

$$a_J(s) = \frac{e^{2i\delta_J(s)} - 1}{2i\rho(s)} = \frac{\sin\delta_J(s)}{\rho(s)} e^{i\delta_J(s)} \tag{19}$$

Physical Insight

Unitarity Circle: The unitarity condition $\mathrm{Im}a_J = \rho|a_J|^2$ confines a_J to a circle in the complex plane. Defining $\tilde{a}_J = \rho \cdot a_J$, unitarity becomes $\mathrm{Im}\tilde{a}_J = |\tilde{a}_J|^2$, which describes a circle of radius $1/2$ centered at $i/2$.

3.5 Crossing Symmetry

For identical particles:

$$\boxed{A(s,t,u) = A(t,s,u) = A(u,t,s)} \tag{20}$$

This is a **non-linear constraint** on the partial waves, mixing different J values through the Legendre expansion.

Theorem 3.2 (Crossing Relations for Partial Waves)**.** Crossing symmetry implies:

$$\sum_{J=0}^{\infty}(2J+1)a_J(s)P_J\left(1 + \frac{2t}{s-4m^2}\right) = \sum_{J=0}^{\infty}(2J+1)a_J(t)P_J\left(1 + \frac{2s}{t-4m^2}\right) \tag{21}$$

3.6 Dispersion Relations

Analyticity and polynomial boundedness imply dispersion relations—integral equations relating the real and imaginary parts of amplitudes.

Definition 3.3 (Fixed-t Dispersion Relation)**.** For fixed $t < 0$:

$$\boxed{A(s,t) = \text{polynomial}(s) + \frac{1}{\pi}\int_{4m^2}^{\infty} ds' \left[\frac{\mathrm{Im}A(s',t)}{s'-s} + \frac{\mathrm{Im}A(s',t)}{s'-u}\right]} \tag{22}$$

where $u = 4m^2 - s - t$.

The number of subtractions in the polynomial depends on the Regge behavior:

- $A \sim s^0$: No subtractions needed
- $A \sim s^1$: One subtraction
- $A \sim s^2$ (gravity): Two subtractions

3.6.1 Roy-Steiner Equations

The Roy equations (and their Steiner generalization) provide a self-consistent system for partial waves:

$$\mathrm{Re}a_J(s) = \text{subtraction terms} + \sum_{J'} \int_{4m^2}^{\infty} ds' K_{JJ'}(s,s') \mathrm{Im}a_{J'}(s') \tag{23}$$

where $K_{JJ'}$ is the crossing kernel.

Analysis Note

Roy equations were originally developed for pion-pion scattering and have achieved remarkable precision. Their extension to gravity-coupled systems is a key challenge of this project.

3.7 Weinberg Soft Theorem

Theorem 3.3 (Weinberg Soft Graviton Theorem)**.** As the graviton momentum $q \to 0$:

$$\boxed{M_{n+1}(\{p_i\}; q, \varepsilon) = \kappa \sum_{i=1}^{n} \frac{\varepsilon_{\mu\nu} p_i^\mu p_i^\nu}{p_i \cdot q} M_n(\{p_i\}) + \mathcal{O}(q^0)} \tag{24}$$

where $\kappa = \sqrt{8\pi G_N}$.

For $2 \to 2$ scattering, this fixes the residue at the graviton pole:

$$\mathcal{M}(s,t) \xrightarrow{t\to 0} \frac{8\pi G_N s^2}{t} + \text{regular} \tag{25}$$

Critical Consideration

Soft Theorem Subtleties: The leading soft theorem is *universal* and follows from gauge invariance. Subleading soft theorems exist but depend on the matter content. Our bootstrap uses only the leading (universal) behavior.

3.8 Regge Bound

Theorem 3.4 (Regge Bound for Gravity)**.** For fixed $t < 0$ and $|s| \to \infty$:

$$\boxed{|A(s,t)| \leq C(t) \cdot s^2} \tag{26}$$

This is saturated by graviton exchange at tree level.

The Regge bound ensures:

1. Dispersion relations converge with two subtractions
2. Causality is preserved (no superluminal signaling)
3. The Froissart-Martin bound is satisfied

3.9 Optimization Problem Formulation

The S-matrix bootstrap becomes a **semidefinite program (SDP)**:

$$\begin{aligned} \text{Maximize/Minimize:} \quad & c_i \quad \text{(Wilson coefficient)} \\ \text{Subject to:} \quad & \text{Im} a_J(s) = \rho(s)|a_J(s)|^2 && \text{(unitarity)} \\ & A(s,t) = A(t,s) && \text{(crossing)} \\ & \text{Dispersion relations hold} && \text{(analyticity)} \\ & \text{Soft theorem residue} = 8\pi G_N s^2/t && \text{(gravity)} \\ & |A(s,t)| \leq Cs^2 && \text{(Regge)} \end{aligned} \tag{27}$$

Research Direction

SDP Relaxation: The unitarity constraint is quadratic, making the problem non-convex. Standard approaches:

1. Parametrize $a_J = \sin\delta_J e^{i\delta_J}/\rho$ (automatically satisfies unitarity)
2. Use moment matrices and SDP hierarchy
3. Discretize and use MILP for integrality

4 Implementation Approach

4.1 Phase 1: Partial-Wave Infrastructure (Months 1–2)

Goal: Build a robust partial-wave calculator with high-precision arithmetic.

4.1.1 Legendre Polynomial Computation

```python
from mpmath import mp, mpf
import scipy.special as sp

mp.dps = 100  # 100 decimal places

def legendre_polynomial(J: int, z: complex) -> complex:
    """
    Compute P_J(z) for complex z.

    Uses recurrence relation for numerical stability:
    (J+1) P_{J+1}(z) = (2J+1) z P_J(z) - J P_{J-1}(z)

    Args:
        J: Angular momentum quantum number (non-negative integer)
        z: Argument (can be complex)

    Returns:
        P_J(z) to mp.dps precision
    """
    if J == 0:
        return mpf(1)
    elif J == 1:
        return z

    P_prev = mpf(1)     # P_0
    P_curr = z          # P_1

```

```
    for j in range(1, J):
        P_next = ((2*j + 1) * z * P_curr - j * P_prev) / (j + 1)
        P_prev = P_curr
        P_curr = P_next

    return P_curr


def legendre_Q(J: int, z: complex) -> complex:
    """
    Compute Legendre function of the second kind Q_J(z).

    Q_J appears in the partial-wave expansion for the
    crossed channel amplitude.
    """
    # Use integral representation for |z| > 1
    if abs(z) > 1:
        def integrand(t):
            return legendre_polynomial(J, t) / (z - t)

        result = mp.quad(integrand, [-1, 1]) / 2
        return result
    else:
        # Use series expansion near z = 1
        return _legendre_Q_series(J, z)
```

Listing 1: High-precision Legendre polynomials

4.1.2 Partial-Wave Projection

```
from mpmath import mp, quad
import numpy as np

def partial_wave_projection(amplitude_func, J: int, s: float,
                            m: float) -> complex:
    """
    Project amplitude onto partial wave J:

    a_J(s) = (1/32pi) * integral_{-1}^{1} d(cos theta)
             * P_J(cos theta) * A(s, t(cos theta))

    Args:
        amplitude_func: Function A(s, t) -> complex
        J: Angular momentum
        s: Center-of-mass energy squared
        m: Particle mass

    Returns:
        Partial-wave amplitude a_J(s)
    """
    s0 = 4 * m**2

    def integrand(cos_theta):
        # Convert cos(theta) to t
        t = (s - s0) * (cos_theta - 1) / 2

        # Evaluate amplitude and Legendre polynomial
        A_val = amplitude_func(s, t)
        P_J = legendre_polynomial(J, cos_theta)

        return P_J * A_val

```

```
    # Numerical integration with high precision
    result = mp.quad(integrand, [-1, 1])

    return result / (32 * mp.pi)


def reconstruct_amplitude(partial_waves: dict, s: float, t: float,
                          m: float, J_max: int) -> complex:
    """
    Reconstruct amplitude from partial waves:

    A(s,t) = 16*pi * sum_J (2J+1) * a_J(s) * P_J(cos theta)

    Args:
        partial_waves: Dictionary {J: a_J(s)} or function
        s, t: Mandelstam variables
        m: Particle mass
        J_max: Maximum J to include in sum

    Returns:
        Reconstructed amplitude
    """
    s0 = 4 * m**2
    cos_theta = 1 + 2 * t / (s - s0)

    A = 0
    for J in range(J_max + 1):
        if callable(partial_waves):
            a_J = partial_waves(J, s)
        else:
            a_J = partial_waves.get(J, 0)

        P_J = legendre_polynomial(J, cos_theta)
        A += (2 * J + 1) * a_J * P_J

    return 16 * mp.pi * A
```

Listing 2: Partial-wave projection from amplitude

Analysis Note

Truncation Error: The partial-wave sum is truncated at $J_{\max}$. For physical amplitudes, high-J contributions are suppressed. Verify convergence by increasing $J_{\max}$ and checking stability.

4.1.3 Phase Space and Unitarity Check

```
import numpy as np

def phase_space(s: float, m: float) -> float:
    """
    Two-body phase space factor: rho(s) = sqrt(1 - 4m^2/s)
    """
    s0 = 4 * m**2
    if s <= s0:
        return 0.0
    return np.sqrt(1 - s0 / s)


def check_elastic_unitarity(a_J: complex, s: float, m: float,
                            tol: float = 1e-10) -> dict:
```

```
    """
    Check elastic unitarity: Im(a_J) = rho * |a_J|^2

    Returns dictionary with check results.
    """
    rho = phase_space(s, m)

    lhs = a_J.imag
    rhs = rho * abs(a_J)**2

    error = abs(lhs - rhs)
    passed = error < tol

    # Also check unitarity bound: |a_J| <= 1/rho
    bound_satisfied = abs(a_J) <= 1/rho + tol if rho > 0 else True

    return {
        'elastic_unitarity_error': error,
        'elastic_unitarity_passed': passed,
        'unitarity_bound_satisfied': bound_satisfied,
        'a_J': a_J,
        's': s,
        'rho': rho
    }


def parametrize_unitary_partial_wave(delta_J: float, s: float,
                                     m: float) -> complex:
    """
    Parametrize partial wave in terms of phase shift.

    a_J = sin(delta_J) * exp(i * delta_J) / rho

    This automatically satisfies elastic unitarity.
    """
    rho = phase_space(s, m)
    if rho == 0:
        return 0.0

    return np.sin(delta_J) * np.exp(1j * delta_J) / rho
```

Listing 3: Unitarity verification

4.2 Phase 2: Dispersion Relations (Months 2–4)

Goal: Implement Roy-Steiner type dispersion relations for gravity-coupled amplitudes.

4.2.1 Twice-Subtracted Dispersion Relation

```
from scipy.integrate import quad
import numpy as np

def twice_subtracted_dispersion(s: complex, t: float,
                                imaginary_amplitude,
                                s0: float, s1: float,
                                a0: complex, a1: complex,
                                m: float, s_max: float = 1e6) -> complex:
    """
    Twice-subtracted dispersion relation (required for gravity).

    A(s,t) = a_0 + a_1 * s + (s^2 / pi) * integral ds'
```

```
            * Im A(s',t) / (s'^2 * (s' - s))
            + crossed channel contribution

    Args:
        s: Evaluation point (can be complex)
        t: Fixed momentum transfer (t < 0)
        imaginary_amplitude: Function Im A(s', t) for s' real
        s0, s1: Subtraction points
        a0, a1: Subtraction constants
        m: Particle mass
        s_max: Upper cutoff for integral

    Returns:
        A(s, t) from dispersion relation
    """
    threshold = 4 * m**2

    def integrand_s_channel(s_prime):
        if s_prime <= threshold:
            return 0.0

        Im_A = imaginary_amplitude(s_prime, t)
        denominator = s_prime**2 * (s_prime - s)
        return Im_A / denominator

    # s-channel integral
    s_integral, s_error = quad(
        lambda sp: integrand_s_channel(sp).real,
        threshold, s_max, limit=200
    )
    s_integral += 1j * quad(
        lambda sp: integrand_s_channel(sp).imag,
        threshold, s_max, limit=200
    )[0]

    # u-channel (crossed) contribution
    # u = 4m^2 - s - t, so s' in u-channel maps to u' = 4m^2 - s' - t
    def integrand_u_channel(u_prime):
        if u_prime <= threshold:
            return 0.0

        s_prime_crossed = threshold - u_prime - t
        Im_A_u = imaginary_amplitude(u_prime, t)  # u-channel imaginary part
        denominator = u_prime**2 * (u_prime - (threshold - s - t))
        return Im_A_u / denominator

    u_integral, u_error = quad(
        lambda up: integrand_u_channel(up).real,
        threshold, s_max, limit=200
    )

    # Combine
    dispersive_part = (s**2 / np.pi) * (s_integral + u_integral)

    return a0 + a1 * s + dispersive_part


def roy_equation_kernel(J: int, Jp: int, s: float, sp: float,
                        m: float) -> float:
    """
    Compute Roy equation kernel K_{J,J'}(s, s').

    This encodes the mixing between partial waves
```

```
    due to crossing symmetry.
    """
    threshold = 4 * m**2

    # The kernel involves Legendre Q functions
    # K_{JJ'} = (2J'+1) / pi * integral dz P_J(z) Q_{J'}(z')
    # where z' is related to z by crossing

    # Simplified formula for identical scalars
    cos_theta = 1 + 2 * (threshold - s - sp) / (s - threshold)

    if abs(cos_theta) <= 1:
        kernel = (2 * Jp + 1) * legendre_polynomial(J, cos_theta)
        kernel *= legendre_polynomial(Jp, cos_theta) / np.pi
    else:
        kernel = (2 * Jp + 1) * legendre_polynomial(J, cos_theta)
        kernel *= legendre_Q(Jp, cos_theta) / np.pi

    return float(kernel.real)
```

Listing 4: Twice-subtracted dispersion relation for gravity

4.2.2 Roy-Steiner System

```
import numpy as np
from scipy.linalg import solve

def setup_roy_steiner_system(s_grid: np.ndarray, J_max: int,
                             m: float, subtraction_constants: dict):
    """
    Set up the Roy-Steiner integral equation system.

    Re a_J(s) = sub_J(s) + sum_{J'} integral ds' K_{JJ'}(s,s') Im a_{J'}(s')

    Args:
        s_grid: Grid of s values
        J_max: Maximum angular momentum
        m: Particle mass
        subtraction_constants: Dict with subtraction polynomial coefficients

    Returns:
        Matrix A, vector b for linear system A * (Im a) = b - (Re a)
    """
    N_s = len(s_grid)
    N_J = J_max + 1
    N_total = N_s * N_J

    # Build kernel matrix
    K = np.zeros((N_total, N_total))

    for i, s in enumerate(s_grid):
        for J in range(N_J):
            row_idx = i * N_J + J

            for ip, sp in enumerate(s_grid):
                for Jp in range(N_J):
                    col_idx = ip * N_J + Jp

                    # Kernel value (with s-grid spacing for integration)
                    ds = s_grid[1] - s_grid[0] if ip < N_s - 1 else 0
                    K[row_idx, col_idx] = roy_equation_kernel(
                        J, Jp, s, sp, m
```

```
                    ) * ds

    # Subtraction terms
    b = np.zeros(N_total)
    for i, s in enumerate(s_grid):
        for J in range(N_J):
            row_idx = i * N_J + J
            b[row_idx] = subtraction_constants.get(
                (J, 0), 0
            ) + subtraction_constants.get((J, 1), 0) * s

    return K, b


def solve_roy_steiner_iteratively(K: np.ndarray, b: np.ndarray,
                                  unitarity_func,
                                  max_iter: int = 100,
                                  tol: float = 1e-8):
    """
    Solve Roy-Steiner equations iteratively with unitarity.

    1. Start with initial guess for Im a_J
    2. Use Roy equations to get Re a_J
    3. Use unitarity Im a_J = rho |a_J|^2 to update Im a_J
    4. Repeat until convergence
    """
    N = len(b)
    Im_a = np.zeros(N)  # Initial guess

    for iteration in range(max_iter):
        # Step 2: Roy equation gives Re a
        Re_a = b - K @ Im_a

        # Step 3: Unitarity update
        Im_a_new = unitarity_func(Re_a, Im_a)

        # Check convergence
        change = np.max(np.abs(Im_a_new - Im_a))
        if change < tol:
            print(f"Converged after {iteration+1} iterations")
            return Re_a, Im_a_new

        Im_a = Im_a_new

    print("Warning: Did not converge")
    return Re_a, Im_a
```

Listing 5: Roy-Steiner equation system

Critical Consideration

Convergence Issues: Roy-Steiner iterations may not converge for all parameter ranges. Monitor the iteration and implement damping if needed: $\text{Im } a^{(n+1)} = \alpha \cdot \text{Im } a_{\text{new}} + (1 - \alpha) \cdot \text{Im } a^{(n)}$.

4.3 Phase 3: Crossing Symmetry Implementation (Months 4–5)

Goal: Enforce crossing symmetry $A(s,t) = A(t,s)$ as constraints.

```
import numpy as np

def crossing_constraint_residual(partial_waves: dict, s: float,
```

```
                                    t: float, m: float,
                                    J_max: int) -> float:
    """
    Compute |A(s,t) - A(t,s)| as a measure of crossing violation.

    For identical particles, A(s,t) = A(t,s) must hold.
    """
    # A(s,t)
    A_st = reconstruct_amplitude(partial_waves, s, t, m, J_max)

    # A(t,s): need to reconstruct with s <-> t swapped
    # This requires partial waves in the t-channel
    A_ts = reconstruct_amplitude(partial_waves, t, s, m, J_max)

    return abs(A_st - A_ts)


def build_crossing_matrix(s_grid: np.ndarray, t_grid: np.ndarray,
                          m: float, J_max: int) -> np.ndarray:
    """
    Build matrix encoding crossing symmetry constraints.

    For each (s,t) test point, crossing gives a linear constraint
    on the partial-wave coefficients (when parametrized appropriately).
    """
    N_s = len(s_grid)
    N_t = len(t_grid)
    N_J = J_max + 1

    # Each (s,t) point gives one equation
    # Variables are the partial-wave values at each s-grid point
    N_vars = N_s * N_J
    N_constraints = N_s * N_t

    C = np.zeros((N_constraints, N_vars), dtype=complex)

    constraint_idx = 0
    for s in s_grid:
        for t in t_grid:
            if s == t:
                continue  # Trivially satisfied

            threshold = 4 * m**2

            # Coefficient for A(s,t) term
            cos_theta_st = 1 + 2 * t / (s - threshold)
            for i_s, s_val in enumerate(s_grid):
                for J in range(N_J):
                    if abs(s_val - s) < 1e-10:
                        col_idx = i_s * N_J + J
                        P_J_st = legendre_polynomial(J, cos_theta_st)
                        C[constraint_idx, col_idx] += 16 * np.pi * (2*J+1) *
    P_J_st

            # Coefficient for -A(t,s) term
            cos_theta_ts = 1 + 2 * s / (t - threshold)
            for i_s, s_val in enumerate(s_grid):
                for J in range(N_J):
                    if abs(s_val - t) < 1e-10:
                        col_idx = i_s * N_J + J
                        P_J_ts = legendre_polynomial(J, cos_theta_ts)
                        C[constraint_idx, col_idx] -= 16 * np.pi * (2*J+1) *
    P_J_ts
```

```
            constraint_idx += 1

    return C[:constraint_idx]


def verify_crossing_symmetry(amplitude_func, test_points: list,
                             tol: float = 1e-8) -> dict:
    """
    Verify crossing symmetry at multiple test points.
    """
    results = {
        'all_passed': True,
        'max_violation': 0.0,
        'violations': []
    }

    for s, t in test_points:
        u = 4 * m**2 - s - t  # Assuming massive particles

        A_st = amplitude_func(s, t)
        A_ts = amplitude_func(t, s)
        A_us = amplitude_func(u, s)

        # Check all crossing relations
        error_st_ts = abs(A_st - A_ts)
        error_st_us = abs(A_st - A_us)

        max_error = max(error_st_ts, error_st_us)
        results['max_violation'] = max(results['max_violation'], max_error)

        if max_error > tol:
            results['all_passed'] = False
            results['violations'].append({
                's': s, 't': t, 'u': u,
                'error_st_ts': error_st_ts,
                'error_st_us': error_st_us
            })

    return results
```

Listing 6: Crossing symmetry constraints

4.4 Phase 4: Soft Theorem Constraints (Months 5–6)

Goal: Implement Weinberg soft graviton theorem as boundary conditions.

```
import numpy as np

def weinberg_soft_residue(s: float, m: float, G_N: float) -> float:
    """
    Compute the required residue at t=0 from Weinberg's theorem.

    M(s,t) -> 8*pi*G_N * s^2 / t as t -> 0

    Args:
        s: Center-of-mass energy squared
        m: Particle mass
        G_N: Newton's constant

    Returns:
        Coefficient of 1/t pole (should be 8*pi*G_N*s^2)
    """
```

```
    return 8 * np.pi * G_N * s**2


def extract_t_zero_residue(amplitude_func, s: float,
                           t_values: list = None) -> float:
    """
    Extract the residue of the amplitude at t=0.

    Uses fitting: M(s,t) approx R/t + regular

    Args:
        amplitude_func: Function M(s, t) -> complex
        s: Fixed s value
        t_values: Small t values to use for extrapolation

    Returns:
        Extracted residue R
    """
    if t_values is None:
        t_values = [-1e-4, -1e-5, -1e-6, -1e-7]

    # Fit M(s,t) * t to extract residue
    residues = []
    for t in t_values:
        M_val = amplitude_func(s, t)
        residue_estimate = M_val * t
        residues.append(residue_estimate)

    # Extrapolate to t -> 0
    # Use Richardson extrapolation or simple average
    residue = np.mean(residues)

    return residue


def soft_theorem_constraint(partial_waves_func, s_grid: np.ndarray,
                            m: float, G_N: float) -> np.ndarray:
    """
    Build constraint vector from soft theorem.

    The sum over partial waves at t=0 must give the correct residue.
    """
    constraints = []

    for s in s_grid:
        if s <= 4 * m**2:
            continue

        # Expected residue from Weinberg
        expected_residue = weinberg_soft_residue(s, m, G_N)

        # Residue from partial-wave sum at t=0 (cos_theta = 1)
        # A(s,0) = 16*pi * sum_J (2J+1) a_J(s) P_J(1)
        # Note: P_J(1) = 1 for all J

        # The constraint is: sum_J (2J+1) a_J(s) = expected / (16*pi) + O(1)
        # Actually, need to handle the pole structure carefully

        constraints.append({
            's': s,
            'expected_residue': expected_residue,
            'constraint_type': 'soft_theorem'
        })
```

```
    return constraints


def verify_soft_theorem(amplitude_func, s_values: list,
                        m: float, G_N: float,
                        tol: float = 1e-6) -> dict:
    """
    Verify that the amplitude satisfies Weinberg's soft theorem.
    """
    results = {'passed': True, 'checks': []}

    for s in s_values:
        expected = weinberg_soft_residue(s, m, G_N)
        actual = extract_t_zero_residue(amplitude_func, s)

        error = abs(actual - expected) / abs(expected)
        check = {
            's': s,
            'expected_residue': expected,
            'actual_residue': actual,
            'relative_error': error,
            'passed': error < tol
        }

        results['checks'].append(check)
        if not check['passed']:
            results['passed'] = False

    return results
```

Listing 7: Weinberg soft theorem implementation

4.5 Phase 5: Regge Bound Implementation (Months 6–7)

```
import numpy as np

def check_regge_bound(amplitude_func, t: float, s_values: list,
                      C_max: float = 1e10) -> dict:
    """
    Verify |A(s,t)| <= C * s^2 for large s.

    Args:
        amplitude_func: Amplitude function
        t: Fixed momentum transfer (t < 0)
        s_values: Large s values to test
        C_max: Maximum allowed constant C

    Returns:
        Dictionary with Regge bound check results
    """
    results = {'passed': True, 'violations': [], 'max_ratio': 0}

    for s in s_values:
        A_val = abs(amplitude_func(s, t))
        ratio = A_val / s**2

        results['max_ratio'] = max(results['max_ratio'], ratio)

        if ratio > C_max:
            results['passed'] = False
            results['violations'].append({
```

```
                's': s,
                't': t,
                'amplitude': A_val,
                'ratio': ratio
            })

    return results


def regge_bound_constraint_matrix(s_grid: np.ndarray, t: float,
                                  m: float, J_max: int,
                                  C_bound: float) -> tuple:
    """
    Build constraints |A(s,t)| <= C*s^2 as linear inequalities.

    Since A is linear in partial waves (for fixed s), this becomes:
    Re A <= C*s^2
    -Re A <= C*s^2
    Im A <= C*s^2
    -Im A <= C*s^2

    (Conservative version using component bounds)
    """
    N_s = len(s_grid)
    N_J = J_max + 1
    N_vars = N_s * N_J

    # Four constraints per s value
    A_ineq = []
    b_ineq = []

    for i, s in enumerate(s_grid):
        if s <= 4 * m**2:
            continue

        cos_theta = 1 + 2 * t / (s - 4 * m**2)

        # Build row: coefficient of each partial wave variable
        row = np.zeros(N_vars)
        for J in range(N_J):
            col_idx = i * N_J + J
            P_J = float(legendre_polynomial(J, cos_theta).real)
            row[col_idx] = 16 * np.pi * (2 * J + 1) * P_J

        # Re A <= C*s^2
        A_ineq.append(row)
        b_ineq.append(C_bound * s**2)

        # -Re A <= C*s^2
        A_ineq.append(-row)
        b_ineq.append(C_bound * s**2)

    return np.array(A_ineq), np.array(b_ineq)
```

Listing 8: Regge bound verification and constraints

4.6 Phase 6: SDP Optimization (Months 7–9)

Goal: Set up and solve the full semidefinite program for Wilson coefficient bounds.

```
import cvxpy as cp
import numpy as np

```

```python
def setup_smatrix_bootstrap_sdp(s_grid: np.ndarray, J_max: int,
                                m: float, G_N: float,
                                n_wilson: int) -> tuple:
    """
    Set up the complete S-matrix bootstrap optimization.

    Variables:
        - Partial wave real parts: Re a_J(s) for each (J, s)
        - Partial wave imaginary parts: Im a_J(s) for each (J, s)
        - Wilson coefficients: c_1, c_2, ..., c_{n_wilson}

    Constraints:
        1. Unitarity: Im a_J = rho |a_J|^2 (as SDP relaxation)
        2. Crossing symmetry: A(s,t) = A(t,s)
        3. Dispersion relations: Re a_J satisfies Roy equations
        4. Soft theorem: t=0 residue matches Weinberg
        5. Regge bound: |A(s,t)| <= C*s^2

    Returns:
        Problem, variables dictionary
    """
    N_s = len(s_grid)
    N_J = J_max + 1
    threshold = 4 * m**2

    # Variables
    # Partial waves (separated into real and imaginary)
    a_real = {}
    a_imag = {}
    for J in range(N_J):
        a_real[J] = cp.Variable(N_s, name=f'a_real_{J}')
        a_imag[J] = cp.Variable(N_s, nonneg=True, name=f'a_imag_{J}')

    # Wilson coefficients
    c = cp.Variable(n_wilson, name='wilson_coeffs')

    constraints = []

    # ========================================
    # Constraint 1: Unitarity (SDP relaxation)
    # ========================================
    # Im a_J >= rho * |a_J|^2 (relaxed from equality)
    # Equivalently: [Im a_J, Re a_J; Re a_J, Im a_J/rho] >> 0

    for J in range(N_J):
        for i, s in enumerate(s_grid):
            if s <= threshold:
                continue

            rho = np.sqrt(1 - threshold / s)

            # Unitarity bound: |a_J|^2 <= Im a_J / rho
            # i.e., a_real^2 + a_imag^2 <= a_imag / rho
            # Equivalent SDP constraint:
            # [[a_imag/rho, a_real], [a_real, a_imag/rho]] >> 0
            # and a_imag >= 0

            # Use SOC constraint: ||(a_real, a_imag - 1/(2*rho))||_2 <= a_imag
    /(2*rho) + 1/(4*rho)
            # Simplified: directly impose |a_J| <= 1/rho
            constraints.append(
                a_real[J][i]**2 + a_imag[J][i]**2 <= 1/rho**2
            )
```

```
    # ========================================
    # Constraint 2: Crossing symmetry
    # ========================================
    # Sample crossing points and impose A(s,t) = A(t,s)

    crossing_points = generate_crossing_test_points(s_grid, m)

    for s_test, t_test in crossing_points:
        # A(s,t) from partial waves at s_test
        A_st = build_amplitude_expression(a_real, a_imag, s_test, t_test,
                                          s_grid, m, J_max)

        # A(t,s) from partial waves at t_test (if t_test in s_grid)
        A_ts = build_amplitude_expression(a_real, a_imag, t_test, s_test,
                                          s_grid, m, J_max)

        if A_ts is not None:
            constraints.append(A_st == A_ts)

    # ========================================
    # Constraint 3: Dispersion relations
    # ========================================
    # Relate partial waves to Wilson coefficients through
    # low-energy expansion of dispersion relation

    K_roy, b_sub = setup_roy_kernel(s_grid, J_max, m)

    # Subtraction constants depend on Wilson coefficients
    for J in range(N_J):
        for i, s in enumerate(s_grid):
            if s <= threshold:
                continue

            # Roy equation: Re a_J(s) = sub_J(s, c) + integral[K * Im a_J']
            subtraction = build_subtraction_polynomial(s, J, c)

            # Integral contribution (discretized)
            integral_term = 0
            for Jp in range(N_J):
                for ip, sp in enumerate(s_grid):
                    if sp <= threshold:
                        continue
                    K_val = K_roy[i * N_J + J, ip * N_J + Jp]
                    integral_term += K_val * a_imag[Jp][ip]

            constraints.append(a_real[J][i] == subtraction + integral_term)

    # ========================================
    # Constraint 4: Soft graviton theorem
    # ========================================
    # At t = 0: residue of amplitude pole = 8*pi*G_N*s^2

    for i, s in enumerate(s_grid):
        if s <= threshold:
            continue

        # Sum over J at t=0 (cos_theta = 1, P_J(1) = 1)
        amplitude_at_t0 = 0
        for J in range(N_J):
            amplitude_at_t0 += 16 * np.pi * (2*J + 1) * (
                a_real[J][i] + 1j * a_imag[J][i]
            )
```

```
129
130         # This should match soft theorem (modulo pole structure)
131         # In practice, constrain the coefficient of certain terms
132         soft_constraint = build_soft_theorem_constraint(
133             amplitude_at_t0, s, G_N
134         )
135         if soft_constraint is not None:
136             constraints.append(soft_constraint)
137
138     # ========================================
139     # Constraint 5: Regge bound
140     # ========================================
141     # |A(s,t)| <= C * s^2 for large s
142
143     C_regge = 1e6  # Bound constant (should be determined physically)
144     t_test_regge = -1.0  # Fixed t for Regge test
145
146     for i, s in enumerate(s_grid):
147         if s < 100 * threshold:  # Only for large s
148             continue
149
150         A_bound = C_regge * s**2
151         A_expr = build_amplitude_expression(
152             a_real, a_imag, s, t_test_regge, s_grid, m, J_max
153         )
154
155         # |A| <= bound (conservative: |Re A| <= bound, |Im A| <= bound)
156         constraints.append(A_expr <= A_bound)
157         constraints.append(-A_expr <= A_bound)
158
159     # Store all variables
160     variables = {
161         'a_real': a_real,
162         'a_imag': a_imag,
163         'wilson': c
164     }
165
166     return constraints, variables
167
168
169 def solve_wilson_bound(coeff_index: int, direction: str,
170                        s_grid: np.ndarray, J_max: int,
171                        m: float, G_N: float, n_wilson: int) -> dict:
172     """
173     Find upper or lower bound on a Wilson coefficient.
174
175     Args:
176         coeff_index: Which coefficient to bound (0-indexed)
177         direction: 'max' or 'min'
178         Other args: Problem parameters
179
180     Returns:
181         Dictionary with bound and certificate
182     """
183     constraints, variables = setup_smatrix_bootstrap_sdp(
184         s_grid, J_max, m, G_N, n_wilson
185     )
186
187     c = variables['wilson']
188
189     # Objective
190     if direction == 'max':
191         objective = cp.Maximize(c[coeff_index])
```

```
    else:
        objective = cp.Minimize(c[coeff_index])

    # Solve
    problem = cp.Problem(objective, constraints)

    try:
        problem.solve(solver=cp.MOSEK, verbose=True)
    except:
        problem.solve(solver=cp.SCS, verbose=True, max_iters=10000)

    result = {
        'status': problem.status,
        'bound': None,
        'wilson_values': None,
        'partial_waves': None,
        'dual_certificate': None
    }

    if problem.status in [cp.OPTIMAL, cp.OPTIMAL_INACCURATE]:
        result['bound'] = c[coeff_index].value
        result['wilson_values'] = c.value
        result['partial_waves'] = {
            J: {
                'real': variables['a_real'][J].value,
                'imag': variables['a_imag'][J].value
            }
            for J in range(J_max + 1)
        }

        # Extract dual certificate
        result['dual_certificate'] = [
            cons.dual_value for cons in constraints
        ]

    return result
```

Listing 9: Full S-matrix bootstrap SDP

Analysis Note

Solver Choice: MOSEK is recommended for its precision on SDPs. If unavailable, SCS (free, open-source) works but may require more iterations. Always verify solutions with the verification functions.

4.7 Phase 7: Certificate Extraction and Verification (Months 9–11)

```
import json
import numpy as np

def extract_certificate(problem_result: dict, constraint_types: list) -> dict:
    """
    Extract machine-verifiable certificate from optimization result.

    For feasible bounds: witness (explicit partial waves + Wilson coeffs)
    For infeasible bounds: dual certificate (Farkas lemma)
    """
    certificate = {
        'type': None,
        'data': {},
        'verification_info': {}
```

```
    }

    if problem_result['status'] == 'optimal':
        certificate['type'] = 'feasibility_witness'
        certificate['data'] = {
            'wilson_coefficients': problem_result['wilson_values'].tolist(),
            'partial_waves': {
                str(J): {
                    'real': pw['real'].tolist(),
                    'imag': pw['imag'].tolist()
                }
                for J, pw in problem_result['partial_waves'].items()
            }
        }

    elif problem_result['status'] == 'infeasible':
        certificate['type'] = 'impossibility_dual'
        certificate['data'] = {
            'dual_variables': [
                d.tolist() if hasattr(d, 'tolist') else d
                for d in problem_result['dual_certificate']
            ],
            'constraint_types': constraint_types
        }

    return certificate


def verify_feasibility_certificate(certificate: dict, params: dict) -> dict:
    """
    Verify that a feasibility certificate is valid.

    Checks:
    1. Unitarity: Im a_J = rho |a_J|^2 (approximately)
    2. Crossing: A(s,t) = A(t,s)
    3. Dispersion relations
    4. Soft theorem
    5. Regge bound
    """
    results = {
        'valid': True,
        'checks': {}
    }

    # Extract data
    wilson = np.array(certificate['data']['wilson_coefficients'])
    partial_waves = certificate['data']['partial_waves']

    m = params['m']
    G_N = params['G_N']
    s_grid = params['s_grid']
    J_max = params['J_max']

    threshold = 4 * m**2

    # Check 1: Unitarity
    unitarity_errors = []
    for J_str, pw in partial_waves.items():
        J = int(J_str)
        for i, s in enumerate(s_grid):
            if s <= threshold:
                continue

```

```
            a_real = pw['real'][i]
            a_imag = pw['imag'][i]
            rho = np.sqrt(1 - threshold / s)

            # Check Im a = rho |a|^2
            expected_imag = rho * (a_real**2 + a_imag**2)
            error = abs(a_imag - expected_imag)
            unitarity_errors.append(error)

    max_unitarity_error = max(unitarity_errors)
    results['checks']['unitarity'] = {
        'max_error': max_unitarity_error,
        'passed': max_unitarity_error < 1e-6
    }

    if not results['checks']['unitarity']['passed']:
        results['valid'] = False

    # Check 2: Crossing (sample points)
    # ... similar structure

    # Check 3: Dispersion relations
    # ... similar structure

    # Check 4: Soft theorem
    # ... similar structure

    # Check 5: Regge bound
    # ... similar structure

    return results


def export_certificate_smt2(certificate: dict, filename: str):
    """
    Export certificate in SMT-LIB2 format for Z3 verification.
    """
    with open(filename, 'w') as f:
        f.write("; S-matrix bootstrap certificate\n")
        f.write("; Generated automatically\n\n")

        if certificate['type'] == 'impossibility_dual':
            # Declare variables
            dual = certificate['data']['dual_variables']
            n_dual = len(dual)

            for i in range(n_dual):
                f.write(f"(declare-const y_{i} Real)\n")

            f.write("\n; Dual feasibility: y >= 0\n")
            for i in range(n_dual):
                f.write(f"(assert (>= y_{i} 0))\n")

            # Add constraint that objective is positive
            f.write("\n; Dual objective > 0 (proves primal infeasibility)\n")
            f.write("(assert (> (+ ")
            for i, d in enumerate(dual):
                f.write(f"(* {d} y_{i}) ")
            f.write(") 0))\n")

            f.write("\n(check-sat)\n")
            f.write("(get-model)\n")

```

```
    print(f"Certificate exported to {filename}")
```

Listing 10: Certificate extraction and verification

4.8 Phase 8: Formal Verification (Months 11–12)

```
import Mathlib.Analysis.Complex.Basic
import Mathlib.Analysis.SpecialFunctions.Trigonometric.Basic
import Mathlib.LinearAlgebra.Matrix.Spectrum

-- Basic definitions for S-matrix bootstrap

/-- Mandelstam variables for 2->2 scattering -/
structure MandelstamVars where
  s : Real
  t : Real
  u : Real
  constraint : s + t + u = 4 * m^2

/-- Partial wave amplitude -/
def PartialWave (J : Nat) (s : Real) : Complex := sorry

/-- Phase space factor -/
def phaseSpace (s m : Real) : Real :=
  if s > 4 * m^2 then Real.sqrt (1 - 4 * m^2 / s) else 0

/-- Elastic unitarity condition -/
def elasticUnitarity (a : Complex) (s m : Real) : Prop :=
  a.im = phaseSpace s m * (a.re^2 + a.im^2)

/-- Unitarity bound -/
theorem unitarity_bound (a : Complex) (s m : Real)
    (h : elasticUnitarity a s m) (hs : s > 4 * m^2) :
    Complex.abs a <= 1 / phaseSpace s m := by
  sorry

/-- Crossing symmetry -/
def crossingSymmetry (A : Real -> Real -> Complex) : Prop :=
  forall s t, A s t = A t s

/-- Regge bound for gravity -/
def reggeBound (A : Real -> Real -> Complex) (C : Real) : Prop :=
  forall s t, t < 0 -> Complex.abs (A s t) <= C * s^2

/-- Weinberg soft theorem residue -/
def weinbergResidue (s G_N : Real) : Real :=
  8 * Real.pi * G_N * s^2

/-- Main theorem: Wilson coefficient bound -/
theorem wilson_bound_c1 (m G_N : Real) (hm : m > 0) (hG : G_N > 0) :
    forall c1 : Real,
    (exists (a : Nat -> Real -> Complex),
      (forall J s, s > 4*m^2 -> elasticUnitarity (a J s) s m) /\
      (crossingSymmetry (reconstructAmplitude a)) /\
      (reggeBound (reconstructAmplitude a) (10^6)) /\
      (satisfiesSoftTheorem a G_N)
    ) ->
    c1_lower <= c1 /\ c1 <= c1_upper := by
  sorry

-- Certificate verification
theorem verify_impossibility_certificate
```

```
    (dual : List Real) (constraints : List Constraint)
    (h_dual_positive : forall d in dual, d >= 0)
    (h_objective_positive : dotProduct dual constraintValues > 0) :
    not (exists solution, satisfiesAllConstraints solution constraints) := by
  -- By Farkas lemma
  sorry
```

Listing 11: Lean 4 formalization sketch

5 Detailed Research Directions

5.1 Direction 1: Single-Coefficient Bounds

Research Direction

First Target: Bound the coefficient c_1 of the dimension-6 operator $\phi^2(\partial\phi)^2/\Lambda^2$.
Approach:

1. Set all other Wilson coefficients to zero: $c_2 = c_3 = \cdots = 0$
2. Solve the SDP to maximize and minimize c_1
3. Extract certificates for both upper and lower bounds
4. Verify independently using Z3 and numerical checks

Expected Outcome: Rigorous interval $[c_1^{\min}, c_1^{\max}]$ with machine-verified proof.

5.2 Direction 2: Multi-Parameter Allowed Regions

Research Direction

Goal: Map the allowed region in (c_1, c_2, c_3) space.
Approach:

1. Fix one coefficient and bound the others
2. Scan over fixed values to trace the boundary
3. Use convex hull algorithms to characterize the region
4. Generate 3D visualization of allowed vs. forbidden regions

Novel Contribution: Multi-parameter constraints are significantly stronger than single-parameter bounds—they capture correlations between Wilson coefficients imposed by consistency.

5.3 Direction 3: Comparison with Positivity Bounds

Research Direction

Complementary Approach: EFT positivity bounds use different techniques (forward limit dispersion relations) and may give different systematics.
Questions:

- Are S-matrix bootstrap bounds tighter than positivity bounds?
- Do they probe different physics (e.g., spinning vs. scalar)?
- Can combining both approaches improve overall constraints?

5.4 Direction 4: Spinning External States

Extend from scalar scattering to:

- **Graviton-scalar:** Probes matter-gravity coupling
- **Graviton-graviton:** Pure gravity higher-derivative terms
- **Photon-graviton:** Electromagnetic-gravity coupling

Each channel provides independent constraints on different Wilson coefficients.

5.5 Direction 5: UV Completion Identification

Research Direction

If bounds are saturated: The boundary of the allowed region corresponds to special theories—potentially identifying UV completions.
Known examples:

- String theory amplitudes saturate certain Regge-related bounds
- Higher-spin theories may saturate high-J unitarity bounds

New question: Do the S-matrix bootstrap bounds identify *new* UV completions?

6 Success Criteria

6.1 Minimum Viable Result (6 months)

✓ Partial-wave projection working with 50+ digit precision

✓ Roy-Steiner type dispersion relations implemented

✓ Unitarity constraints imposed (SDP relaxation)

✓ Crossing symmetry verified at test points

✓ Soft theorem constraints implemented

✓ **First rigorous bound:** Single Wilson coefficient bounded

✓ Certificate extracted and verified numerically

6.2 Strong Result (9 months)

- ✓ Multi-parameter bounds on $\{c_1, c_2, c_3\}$
- ✓ Allowed region characterized and visualized
- ✓ All certificates in machine-verifiable format (JSON/SMT-LIB)
- ✓ Independent verification by Z3 or similar
- ✓ Comparison with existing positivity bounds
- ✓ Novel constraints discovered (tighter or new correlations)

6.3 Publication-Quality Result (12 months)

- ✓ Bounds on all Wilson coefficients up to dimension-8
- ✓ Systematic comparison with string theory predictions
- ✓ Formal proofs in Lean 4 for key bounds
- ✓ Public code repository with documentation
- ✓ ArXiv preprint with certificate repository

7 Verification Protocol

7.1 For Claimed Feasibility (Wilson Coefficients Allowed)

```python
def verify_wilson_feasibility(wilson_coeffs: np.ndarray,
                              partial_waves: dict,
                              params: dict,
                              tolerance: float = 1e-8) -> dict:
    """
    Verify that claimed Wilson coefficients are allowed.

    Must check ALL constraints:
    1. Unitarity for all J, s
    2. Crossing at sampled points
    3. Dispersion relations
    4. Soft theorem
    5. Regge bound
    """
    m = params['m']
    G_N = params['G_N']
    s_grid = params['s_grid']
    J_max = params['J_max']

    results = {
        'overall_status': 'VERIFIED',
        'unitarity': None,
        'crossing': None,
        'dispersion': None,
        'soft_theorem': None,
        'regge': None
    }

    # 1. Unitarity check
    unitarity_result = verify_unitarity_all(partial_waves, s_grid, m)
    results['unitarity'] = unitarity_result
```

```
    if not unitarity_result['passed']:
        results['overall_status'] = 'FAILED: Unitarity'
        return results

    # 2. Crossing symmetry
    crossing_points = generate_crossing_test_points(s_grid, m, n_points=100)
    crossing_result = verify_crossing_symmetry(
        lambda s, t: reconstruct_from_partial_waves(partial_waves, s, t, m,
    J_max),
        crossing_points,
        tol=tolerance
    )
    results['crossing'] = crossing_result
    if not crossing_result['all_passed']:
        results['overall_status'] = 'FAILED: Crossing'
        return results

    # 3. Dispersion relations
    dispersion_result = verify_dispersion_relations(
        partial_waves, s_grid, m, wilson_coeffs
    )
    results['dispersion'] = dispersion_result
    if not dispersion_result['passed']:
        results['overall_status'] = 'FAILED: Dispersion'
        return results

    # 4. Soft theorem
    soft_result = verify_soft_theorem(
        lambda s, t: reconstruct_from_partial_waves(partial_waves, s, t, m,
    J_max),
        s_values=[s for s in s_grid if s > 4*m**2],
        m=m, G_N=G_N, tol=tolerance
    )
    results['soft_theorem'] = soft_result
    if not soft_result['passed']:
        results['overall_status'] = 'FAILED: Soft theorem'
        return results

    # 5. Regge bound
    regge_result = check_regge_bound(
        lambda s, t: reconstruct_from_partial_waves(partial_waves, s, t, m,
    J_max),
        t=-1.0,
        s_values=[s for s in s_grid if s > 100 * 4 * m**2],
        C_max=1e10
    )
    results['regge'] = regge_result
    if not regge_result['passed']:
        results['overall_status'] = 'FAILED: Regge bound'
        return results

    print("All verification checks PASSED")
    return results
```

Listing 12: Comprehensive feasibility verification

7.2 For Claimed Infeasibility (Wilson Coefficients Forbidden)

1. **Dual certificate extraction:** Get $y \geq 0$ such that $b^T y > 0$ and $A^T y \leq 0$
2. **Farkas lemma verification:** Confirms primal infeasibility
3. **SMT export:** Verify with Z3: `z3 certificate.smt2`

4. **Lean formalization:** Machine-checked proof

```python
def verify_infeasibility_farkas(dual_y: np.ndarray,
                                A: np.ndarray,
                                b: np.ndarray,
                                tolerance: float = 1e-10) -> dict:
    """
    Verify that dual certificate proves primal infeasibility.

    Farkas lemma: Exactly one of these holds:
    (a) Exists x >= 0 with Ax = b
    (b) Exists y with A^T y <= 0 and b^T y > 0

    If we have (b), then (a) is impossible.
    """
    result = {'valid': True, 'checks': {}}

    # Check y >= 0
    y_nonneg = np.all(dual_y >= -tolerance)
    result['checks']['y_nonnegative'] = {
        'passed': y_nonneg,
        'min_y': np.min(dual_y)
    }

    # Check A^T y <= 0
    AT_y = A.T @ dual_y
    AT_y_nonpos = np.all(AT_y <= tolerance)
    result['checks']['AT_y_nonpositive'] = {
        'passed': AT_y_nonpos,
        'max_AT_y': np.max(AT_y)
    }

    # Check b^T y > 0
    bT_y = np.dot(b, dual_y)
    bT_y_positive = bT_y > tolerance
    result['checks']['bT_y_positive'] = {
        'passed': bT_y_positive,
        'value': bT_y
    }

    # Overall validity
    result['valid'] = y_nonneg and AT_y_nonpos and bT_y_positive

    if result['valid']:
        print("Farkas certificate VERIFIED: Region is FORBIDDEN")
    else:
        print("Farkas certificate INVALID")

    return result
```

Listing 13: Infeasibility verification via Farkas

8 Common Pitfalls and Mitigations

8.1 Numerical Precision Loss

Critical Consideration

Problem: Standard double precision (~ 15 digits) is insufficient for S-matrix bootstrap. Crossing symmetry violations at 10^{-14} level can accumulate and corrupt results.
Solution:

- Use `mpmath` with `mp.dps = 100` or higher
- Verify all intermediate results to 50+ digits
- Use interval arithmetic for rigorous error bounds

8.2 Roy Equation Non-Convergence

Critical Consideration

Problem: Iterative solution of Roy equations may fail to converge, oscillate, or converge to wrong solution.
Solutions:

- Use damping: $a^{(n+1)} = \alpha\, a_{\text{new}} + (1-\alpha)a^{(n)}$ with $\alpha \sim 0.3$
- Start with physical initial guess (e.g., tree-level)
- Monitor convergence and try multiple initializations

8.3 Partial-Wave Truncation

Critical Consideration

Problem: Truncating at finite $J_{\max}$ may miss important high-J contributions.
Solutions:

- Start with $J_{\max} = 10$, increase until results stabilize
- Use asymptotic bounds on high-J partial waves
- For physical amplitudes, high-J suppressed by $1/J!$-like factors

8.4 SDP Solver Issues

Critical Consideration

Problem: SDP solvers may return inaccurate solutions, claim infeasibility incorrectly, or time out.
Solutions:

- **Always verify** claimed solutions independently
- Try multiple solvers (MOSEK, SCS, CVXOPT)
- Scale problem to improve numerical conditioning
- For large problems, use iterative methods or decomposition

8.5 Soft Theorem Pole Handling

Critical Consideration

Problem: The graviton pole at $t = 0$ requires careful treatment—naive evaluation gives infinity.
Solutions:

- Subtract pole analytically and work with regular part
- Impose soft theorem as constraint on residue, not full amplitude
- Use dispersion relations that handle poles correctly

9 Resources and References

9.1 Essential Papers

1. Paulos, Penedones, Toledo, van Rees, Vieira (2017): "The S-matrix bootstrap I: QFT in AdS" [arXiv:1607.06109] — Modern S-matrix bootstrap framework
2. Caron-Huot, Mazac, Rastelli, Simmons-Duffin (2021): "Sharp boundaries for the swampland" [arXiv:2102.08951] — State-of-the-art bounds with gravity
3. Bellazzini, Miber, Riva (2021): "New phenomenological and theoretical perspective on anomalous ZZ and $Z\gamma$ processes" [arXiv:2103.02990] — Phenomenological applications
4. Tolley, Wang, Zhou (2021): "New positivity bounds from full crossing symmetry" [arXiv:2011.02400] — Crossing-symmetric positivity
5. Arkani-Hamed, Huang, Huang (2021): "Differential Equations for Graviton Scattering" — Modern amplitude methods
6. Weinberg (1965): "Infrared photons and gravitons" — Original soft theorem paper

9.2 Software and Tools

- **mpmath:** Arbitrary precision arithmetic — `pip install mpmath`
- **CVXPY:** Convex optimization modeling — `pip install cvxpy`

- **MOSEK:** Commercial SDP solver (free academic) — `https://mosek.com`
- **SCS:** Open-source splitting conic solver — `pip install scs`
- **Z3:** SMT solver for certificate verification — `https://github.com/Z3Prover/z3`
- **Lean 4:** Proof assistant — `https://lean-lang.org`
- **SymPy:** Symbolic mathematics — `pip install sympy`

9.3 Background Reading

- Eden, Landshoff, Olive, Polkinghorne: "The Analytic S-Matrix" (Cambridge) — Classic on dispersion relations and analyticity
- Martin, Cheung: "Analyticity Properties and Bounds of the Scattering Amplitudes" — Rigorous mathematical treatment
- Elvang & Huang: "Scattering Amplitudes in Gauge Theory and Gravity" (Cambridge) — Modern amplitude methods
- Boyd & Vandenberghe: "Convex Optimization" (Cambridge) — SDP theory and algorithms

10 Milestone Checklist

10.1 Infrastructure (Months 1–2)

☐ High-precision Legendre polynomials implemented

☐ Partial-wave projection tested on known amplitudes

☐ Amplitude reconstruction from partial waves verified

☐ Phase space and unitarity bounds coded

☐ Tree-level graviton exchange amplitude implemented

10.2 Dispersion Relations (Months 2–4)

☐ Twice-subtracted dispersion relation coded

☐ Roy-Steiner kernel computed

☐ Iterative Roy equation solver working

☐ Convergence verified on test cases

☐ Subtraction constants linked to Wilson coefficients

10.3 Constraints (Months 4–6)

☐ Crossing symmetry constraint matrix built

☐ Soft graviton theorem constraints implemented

☐ Regge bound constraints added

☐ All constraints combined into unified system

10.4 Optimization (Months 6–9)

- ☐ SDP formulation complete
- ☐ Single Wilson coefficient bounded
- ☐ Multi-parameter bounds computed
- ☐ Allowed region characterized
- ☐ Dual certificates extracted

10.5 Verification (Months 9–11)

- ☐ Numerical verification routines complete
- ☐ SMT-LIB export implemented
- ☐ Z3 verification successful
- ☐ All certificates validated

10.6 Formalization (Months 11–12)

- ☐ Lean 4 formalization begun
- ☐ Core theorems stated and proved
- ☐ Certificate verification formalized
- ☐ Documentation and publication draft

11 Conclusion

The nonperturbative S-matrix bootstrap with gravity represents a frontier research program combining classical physics insights (unitarity, crossing, causality) with modern computational methods (semidefinite programming, formal verification). Unlike perturbative approaches, the bootstrap provides *rigorous, model-independent bounds* valid at any coupling strength.

Success in this challenge would:

1. **Constrain the Swampland:** Identify which EFT parameters are incompatible with quantum gravity UV completion
2. **Discover universal features:** The bounds apply to *any* consistent theory, potentially revealing new universal properties of quantum gravity
3. **Produce machine-verified proofs:** All results come with certificates verifiable by independent tools, setting a new standard for rigor in theoretical physics
4. **Connect to phenomenology:** The Wilson coefficient bounds constrain measurable quantum gravity corrections to the Standard Model and general relativity

The methodology—combining dispersion relations, optimization theory, and formal verification—exemplifies a new paradigm for theoretical physics where conjectures become theorems with machine-checked proofs.