

Challenge 06: Non-perturbative S-matrix Bootstrap with Gravity

Pure Thought AI Challenge 06

Pure Thought AI Challenges Project

January 18, 2026

Abstract

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

Contents

Domain: Quantum Gravity Particle Physics

Difficulty: High

Timeline: 6-12 months

Prerequisites: S-matrix theory, dispersion relations, partial-wave unitarity

0.1 Problem Statement

0.1.1 Scientific Context

The S-matrix bootstrap program uses only fundamental axioms—unitarity, crossing symmetry, and analyticity—to constrain scattering amplitudes non-perturbatively. Unlike perturbative calculations, this approach makes no assumption about weak coupling and can produce rigorous bounds on effective field theory (EFT) parameters.

When gravity is included (massless graviton exchange), additional constraints emerge:

- Weinberg's soft graviton theorem
- Regge boundedness (polynomial growth at high energies)
- Causality from graviton propagation

0.1.2 The Core Question

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

For example, consider scalar-graviton scattering with effective Lagrangian:

$$1 \quad L = L_{GR} + (1/2)(\quad) - (1/2)m^2 + c_1 \partial_\mu \phi \partial^\mu \phi /$$

What ranges of c_1, c_2, c_3, \dots are consistent with:

- Unitarity
- Crossing symmetry
- Analyticity (dispersion relations)
- Weinberg soft graviton theorem
- Regge bounds

0.1.3 Why This Matters

- **Non-perturbative rigor:** Valid beyond weak coupling
 - **Model-independent:** Applies to any QFT + gravity
 - **Swampland program:** Identifies which EFTs can UV-complete with gravity
 - **Complementary to positivity:** Different systematic from EFT positivity bounds
-

0.2 Mathematical Formulation

0.2.1 Problem Definition

Consider 2→2 scattering: $(p) + (p) \rightarrow (p) + (p)$ mediated by graviton exchange.

Mandelstam variables:

```

1 s = ( p     +   p    )
2 t = ( p     -   p    )
3 u = ( p     -   p    )
4 s + t + u = 4 m

```

Amplitude: $A(s, t)$

Partial-wave expansion:

```

1 A(s, t) = 16      _ {J=0}^{\infty} (2J+1) a_J(s) P_J(\cos \theta)

```

where $\cos \theta = 1 + 2t/(s-4m^2)$

0.2.2 Constraints from Physics

1. Unitarity:

For elastic scattering ($s <$ first inelastic threshold):

```

1 Im a_J(s) = (s) |a_J(s)|

```

where $(s) = (1 - 4m^2/s)$ is phase space.

This implies:

```

1 |a_J(s)| ≤ 1/(s) (unitarity bound)

```

2. Crossing Symmetry:

```

1 A(s, t) = A(t, s) = A(u, t)

```

3. Analyticity Dispersion Relations:

Roy-Steiner equations (fixed-t dispersion relation):

```

1 Re a_J(s) = polynomial(s) + (s - s') / - {4m}^2 ds' Im
      a_J(s') / (s' - s) / ((s' - s)(s' - s))

```

4. Weinberg Soft Graviton Theorem:

As graviton momentum $q \rightarrow 0$:

```

1 M(..., q^2, ...{...}) = (p^2_i p^2_{-i}) / (p_i p_{-i} - q^2)
      M_n(... without graviton)

```

This constrains residues at $t=0, u=0$ poles.

5. Regge Bound:

For fixed $t, s \rightarrow \infty$:

```

1 |A(s, t)| ≤ C s (gravity Regge bound)

```

This bounds the number of subtractions in dispersion relations.

0.2.3 Optimization Formulation

Feasibility problem:

Given Wilson coefficients c_1, \dots, c_n , determine if there exist partial waves $a_J(s)$ satisfying all constraints.

Bounding problem:

```

1 Maximise/Minimise: c_i
2 Subject to:
3   - Unitarity: |a_J(s)|      1/ (s)
4   - Crossing: _J (2J+1) a_J(s) P_J(z) = _J (2J+1) a_J(t) P_J(z')
5   - Dispersion relations hold
6   - Soft theorem residues correct
7   - Regge bound satisfied

```

This is a **semi-infinite linear program** (infinite-dimensional due to continuum s , finite J).

0.2.4 Certificate of Correctness

If coefficients are allowed:

- Explicit partial waves $a_J(s)$ satisfying all constraints
- Verification: check unitarity bound pointwise
- Verification: evaluate crossing equation on grid
- Verification: verify dispersion integral converges

If coefficients are forbidden:

- **Dual certificate:** A functional (s, J) such that:

```

1     applied to (constraints) gives contradiction
2     0 on physical region

```

- This proves mathematically that no consistent S-matrix exists

0.3 Implementation Approach

0.3.1 Phase 1: Single-Channel Scalar-Graviton (Months 1-3)

Build partial-wave infrastructure:

```

1 import numpy as np
2 import scipy.special as sp
3 from mpmath import mp
4
5 mp.dps = 100 # High precision
6
7 def legendre_polynomial(J, z):
8     """Compute P_J(z)"""
9     return sp.eval_legendre(J, z)
10

```

```

11 def partial_wave_projection(amplitude_func, J, s_val):
12     """
13     Project amplitude onto J-th partial wave
14
15     a_J(s) = (1/2)           dz P_J(z) A(s, t(z))
16     """
17     def integrand(z):
18         t = compute_t_from_z(s_val, z)
19         return legendre_polynomial(J, z) * amplitude_func(s_val, t)
20
21     result, error = mp.quad(integrand, [-1, 1])
22     return result / 2
23
24 def compute_t_from_z(s, z):
25     """
26     t = (s - 4m)(z-1)/2
27     """
28     return (s - 4*m**2) * (z - 1) / 2

```

Tree-level amplitude (Einstein gravity + scalar):

```

1 def tree_amplitude_scalar_graviton(s, t, m, M_pl):
2     """
3     Leading order: gravitational attraction between scalars
4
5     A ~ G_N s t u / M_pl
6     """
7     u = 4*m**2 - s - t
8     return (8 * np.pi / M_pl**2) * s * t * u / (s * t * u) # Simplified
9
10 def tree_amplitude_with_corrections(s, t, m, M_pl, wilson_coeffs):
11     """
12     Include higher-derivative corrections
13
14     A = A_tree + c_s / + c_t / + ...
15     """
16     A_tree = tree_amplitude_scalar_graviton(s, t, m, M_pl)
17
18     # Higher-derivative corrections
19     corrections = 0
20     corrections += wilson_coeffs['c1'] * s**2 /
21         wilson_coeffs['Lambda']**2
22     corrections += wilson_coeffs['c2'] * t**2 /
23         wilson_coeffs['Lambda']**2
24     corrections += wilson_coeffs['c3'] * s * t /
25         wilson_coeffs['Lambda']**2
26
27     return A_tree + corrections

```

0.3.2 Phase 2: Dispersion Relations (Months 3-5)

Implement Roy equations:

```

1 def dispersion_kernel(s, s_prime, s0, J):
2     """
3     Kernel for partial-wave dispersion relation

```

```

4     K(s, s') such that:
5     Re a_J(s) = poly +      K(s,s') Im a_J(s') ds'
6     """
7
8     # Fixed-t dispersion relation kernel
9     return (s - s0) / ((s_prime - s0) * (s_prime - s))
10
11 def rov_equation(a_J_real, a_J_imag, s_grid, J):
12     """
13     Self-consistency equation for partial waves
14
15     Re a_J(s) must match dispersive integral of Im a_J(s')
16     """
17     s0 = 4 * m**2 # Threshold
18
19     for s in s_grid:
20         # Left-hand side: input real part
21         lhs = a_J_real(s)
22
23         # Right-hand side: dispersive integral
24         def integrand(s_prime):
25             K = dispersion_kernel(s, s_prime, s0, J)
26             return K * a_J_imag(s_prime)
27
28         rhs_integral = np.trapz([integrand(sp) for sp in s_grid],
29                                s_grid) / np.pi
30         rhs_poly = polynomial_subtraction(s, J) # Subtraction
31         polynomial
32         rhs = rhs_poly + rhs_integral
33
34         # Verify self-consistency
35         if abs(lhs - rhs) > 1e-6:
36             return False, s, lhs, rhs
37
38     return True, None, None, None

```

Unitarity relation:

```

1 def unitarity_constraint(a_J, s, m):
2     """
3     Below inelastic threshold:
4     Im a_J(s) = (s) |a_J(s)|
5
6     where   (s) = (1 - 4 m / s)
7     """
8     rho = np.sqrt(1 - 4*m**2 / s)
9
10    # Elastic unitarity
11    Im_aJ_expected = rho * abs(a_J)**2
12    Im_aJ_actual = np.imag(a_J)
13
14    return np.isclose(Im_aJ_actual, Im_aJ_expected, rtol=1e-8)

```

0.3.3 Phase 3: Crossing Symmetry (Months 5-7)

Implement crossing equations:

```

1 def crossing_equation(partial_waves, s, t, J_max):
2     """
3         Crossing: A(s,t) = A(t,s) = A(u,t)
4
5             _J (2J+1) a_J(s) P_J(z_s) = _J (2J+1) a_J(t) P_J(z_t)
6     """
7     # Compute scattering angle for s-channel
8     z_s = compute_scattering_angle(s, t)
9     # Compute scattering angle for t-channel
10    z_t = compute_scattering_angle(t, s)
11
12    # s-channel sum
13    A_s = sum((2*J+1) * partial_waves['s'][J](s) *
14               legendre_polynomial(J, z_s)
15               for J in range(J_max))
16
17    # t-channel sum
18    A_t = sum((2*J+1) * partial_waves['t'][J](t) *
19               legendre_polynomial(J, z_t)
20               for J in range(J_max))
21
22    return abs(A_s - A_t) < 1e-8

```

0.3.4 Phase 4: Soft Theorem Constraints (Months 7-8)

Weinberg soft graviton:

```

1 def soft_graviton_residue(amplitude, m):
2     """
3         Extract residue at t=0 (soft graviton exchange)
4
5         A(s,t) ~ R_soft/t as t      0
6
7         Weinberg: R_soft = specific function of s, m
8     """
9
10        # Compute expected soft factor
11        def weinberg_soft_factor(s, m):
12            # Universal gravitational coupling
13            return 8 * np.pi / M_pl**2 * (s - 2*m**2)
14
15        expected_residue = weinberg_soft_factor(s, m)
16
17        # Extract actual residue from amplitude
18        t_small = 1e-6
19        actual_residue = amplitude(s, t_small) * t_small
20
21        # Verify match
22        assert np.isclose(actual_residue, expected_residue, rtol=1e-6), \
               f"Soft theorem violated: {actual_residue} vs {expected_residue}"

```

0.3.5 Phase 5: Optimization via Linear/Semidefinite Programming (Months 8-11)

Formulate as optimization:

```

1 import cvxpy as cp
2
3 def setup_smatrix_bootstrap(s_grid, J_max, wilson_bounds=None):
4     """
5         Set up optimization problem to bound Wilson coefficients
6     """
7     # Discretize: partial waves at grid points
8     # Variables: a_J[s_i] for J=0,...,J_max and s_i in s_grid
9
10    num_s_points = len(s_grid)
11    a_real = {}
12    a_imag = {}
13
14    for J in range(J_max):
15        a_real[J] = cp.Variable(num_s_points)
16        a_imag[J] = cp.Variable(num_s_points)
17
18    # Variables: Wilson coefficients
19    c = cp.Variable(n_wilson_coeffs)
20
21    constraints = []
22
23    # 1. UNITARITY
24    for J in range(J_max):
25        for i, s in enumerate(s_grid):
26            if s < inelastic_threshold:
27                rho_s = np.sqrt(1 - 4*m**2/s)
28                # |a_J|      Im a_J / rho
29                constraints.append(
30                    a_real[J][i]**2 + a_imag[J][i]**2 <= a_imag[J][i] /
31                    rho_s
32                )
33
34    # 2. CROSSING SYMMETRY
35    # Discretize crossing equation at test points
36    for s_test, t_test in crossing_test_points:
37        s_channel_sum = compute_partial_wave_sum(a_real, a_imag,
38            s_test, t_test, 's')
39        t_channel_sum = compute_partial_wave_sum(a_real, a_imag,
40            t_test, s_test, 't')
41        constraints.append(s_channel_sum == t_channel_sum)
42
43    # 3. DISPERSION RELATIONS
44    for J in range(J_max):
45        for i, s in enumerate(s_grid):
46            dispersive_integral = compute_dispersive_integral(
47                a_imag[J], s, s_grid
48            )
49            poly_part = subtraction_polynomial(s, J, c) # Depends on
50            # Wilson coeffs
51            constraints.append(a_real[J][i] == poly_part +
52                dispersive_integral)
53
54    # 4. SOFT THEOREM
55    # Residue at t=0 must match Weinberg

```

```

51     soft_constraint = extract_t_zero_residue(a_real, a_imag, c)
52     weinberg_value = compute_weinberg_residue(s_grid[0], m)
53     constraints.append(soft_constraint == weinberg_value)

54
55     # 5. REGGE BOUND
56     # At large s: A(s,t) ~ s^2
57     # Constrains high partial waves
58     for J in range(J_max):
59         constraints.append(a_real[J][-1] <= regge_bound(s_grid[-1], J))

60
61     # OBJECTIVE: Maximize c[0] (for example)
62     objective = cp.Maximize(c[0])

63
64     problem = cp.Problem(objective, constraints)
65     return problem, c, a_real, a_imag

```

Solve and extract certificate:

```

1 def solve_and_extract_certificate(problem):
2     """
3     Solve optimization and extract dual certificate if infeasible
4     """
5     problem.solve(solver=cp.MOSEK, verbose=True)
6
7     if problem.status == 'optimal':
8         return {
9             'status': 'feasible',
10            'wilson_coeffs': c.value,
11            'partial_waves': {J: a.value for J, a in a_real.items()}
12        }
13    elif problem.status == 'infeasible':
14        # Extract dual variables (certificate of infeasibility)
15        dual_cert = {}
16        for i, constraint in enumerate(problem.constraints):
17            dual_cert[f'constraint_{i}'] = constraint.dual_value
18
19        return {
20            'status': 'infeasible',
21            'certificate': dual_cert
22        }

```

0.3.6 Phase 6: Verification Formal Proofs (Months 11-12)

```

1 def verify_smatrix_solution(partial_waves, wilson_coeffs, s_grid):
2     """
3     Comprehensive verification of solution
4     """
5     print("Verifying S-matrix bootstrap solution...")
6
7     # 1. Unitarity
8     print("  Checking unitarity...")
9     for J in range(J_max):
10        for s in s_grid:
11            assert check_unitarity(partial_waves[J](s), s)

```

```

13 # 2. Crossing
14 print("  Checking crossing symmetry...")
15 for s, t in test_points:
16     assert check_crossing(partial_waves, s, t)
17
18 # 3. Dispersion relations
19 print("  Checking dispersion relations...")
20 for J in range(J_max):
21     assert check_dispersion_relation(partial_waves[J], s_grid)
22
23 # 4. Soft theorem
24 print("  Checking soft graviton theorem...")
25 assert check_soft_theorem(partial_waves, m, M_pl)
26
27 # 5. Regge bound
28 print("  Checking Regge bound...")
29 assert check_regge_bound(partial_waves, s_grid[-1])
30
31 print("All checks passed! Solution verified.")
32 return True

```

0.4 Example Starting Prompt

```

26 8. Compute Weinberg's universal soft factor and verify they match.
27
28 PHASE 4 - Optimization:
29 9. Formulate as LP/SDP: find  $c$  maximizing/minimizing subject to:
30   - Unitarity constraints
31   - Dispersion relations
32   - Crossing symmetry
33   - Soft theorem
34   - Regge bound
35
36 10. Solve using cvxpy + MOSEK.
37
38 PHASE 5 - Extract certificate:
39 11. If feasible: extract explicit partial waves and verify all
40   constraints.
41
42 12. If infeasible: extract dual functional proving impossibility.
43
44 Please implement this step-by-step with exact arithmetic where possible
  and cross-check against known results in the literature.

```

0.5 Success Criteria

0.5.1 Minimum Viable Result (6 months)

Infrastructure complete:

- Partial-wave projection working
- Dispersion relations implemented
- Crossing symmetry verified for tree-level

First bound obtained:

- Rigorous bound on one Wilson coefficient
- Dual certificate extracted (if infeasible)
- Independent verification confirms result

0.5.2 Strong Result (9 months)

Multi-parameter bounds:

- Simultaneous constraints on c, c, c
- Allowed region in parameter space mapped
- Comparison with EFT positivity bounds

Multiple channels:

- Scalar-scalar + graviton
- Scalar-graviton \rightarrow scalar-graviton
- Consistency across channels verified

0.5.3 Publication-Quality Result (12 months)

Comprehensive EFT space:

- All Wilson coefficients to dimension-8 bounded
- Systematic comparison with swampland criteria
- Identification of universal bounds

Formal verification:

- Certificates formalized in Lean/Isabelle
- All proofs machine-checkable
- Publication with proof repository

0.6 Verification Protocol

```

1  def verify_wilson_bound(c_value, certificate_type,
2      certificate_data):
3          """
4              Verify claimed Wilson coefficient bound
5          """
6
7      if certificate_type == 'feasible':
8          # Verify explicit partial waves satisfy all constraints
9          partial_waves = certificate_data['partial_waves']
10
11         assert all(check_unitarity(a_J, s) for a_J in
12             partial_waves for s in s_grid)
13         assert all(check_crossing(partial_waves, s, t) for s, t in
14             test_points)
15         assert check_dispersion_relations(partial_waves, s_grid)
16         assert check_soft_theorem(partial_waves)
17
18
19     return "FEASIBLE VERIFIED"
20
21
22 elif certificate_type == 'infeasible':
23     # Verify dual certificate proves impossibility
24     dual_functional = certificate_data['dual']
25
26     # Dual must be positive on allowed region
27     assert verify_dual_positivity(dual_functional)
28
29     # Dual applied to constraints gives contradiction
30     gap = evaluate_dual_on_constraints(dual_functional)
31     assert gap < -1e-10 # Negative gap proves infeasibility
32
33
34     return "IMPOSSIBILITY PROVEN"

```

0.7 Milestone Checklist

- Partial-wave projection implemented and tested
 - Tree-level amplitude verified against known results
 - Unitarity bounds imposed and checked
 - Roy dispersion relations implemented
 - Crossing symmetry verified numerically
 - Soft graviton theorem constraint added
 - Regge bound implemented
 - LP/SDP solver infrastructure working
 - First Wilson coefficient bound obtained
 - Dual certificate extracted and verified
 - Multi-parameter optimization completed
 - Formal verification initiated
 - Publication draft with certificates
-

Next Steps: Start with implementing partial-wave projection for scalar scattering. Verify against known amplitudes before adding gravity. Build robust dispersion relation infrastructure with high-precision arithmetic.