PRD 27: N-Body Problem and Central Configurations

Pure Thought AI Challenge 27

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: Celestial Mechanics Dynamical Systems
Timeline: 6-9 months
Difficulty: High
Prerequisites: Classical mechanics, algebraic geometry, computational topology, group theory

---

0.1 1. Problem Statement

0.1.1 Scientific Context

The **N-body problem** is one of the oldest and most fundamental problems in mathematical physics: given N point masses interacting via Newtonian gravity, determine their future motion. While the 2-body problem has exact solutions (Kepler ellipses), N 3 is generally non-integrable and chaotic. However, there exist special initial configurations called **central configurations** where all bodies maintain the same geometric shape while the configuration rotates or expands/contracts homothetically.

Central configurations are solutions to the algebraic equations:
U = mr for all i = 1,...,N
where:

- U = - mm/|r - r| is the gravitational potential
- is a scalar Lagrange multiplier (related to angular velocity)
- r are the positions (typically d = 2 or 3)
- m > 0 are the masses

These configurations generate **homothetic solutions**: r(t) = (t)r where the shape r is preserved. Famous examples include:

- **Lagrange equilateral triangle** (3-body, L4/L5 points)
- **Euler collinear solution** (3-body, L1/L2/L3 points)
- **Figure-eight choreography** (3-body with equal masses discovered by Moore 1993, Chenciner-Montgomery 2000)

0.1.2 Core Question

Can we enumerate all central configurations for N bodies with given masses, classify them by symmetry and stability, and certify them using computational algebraic geometry?

Key challenges:

- **Finiteness conjecture**: For planar N-body with generic masses, are there finitely many central configurations? (Known for N 4, open for N 5)
- **Enumeration**: Explicitly find all solutions to the nonlinear system U = mr
- **Stability classification**: Which central configs generate stable periodic orbits?
- **Symmetry analysis**: How do symmetry groups (rotations, reflections) constrain solutions?
- **Certification**: Generate algebraic certificates proving completeness of enumeration

0.1.3 Why This Matters

- **Celestial mechanics**: Trojan asteroids at Jupiter's L4/L5 points, Earth-Moon-Sun Lagrange points (JWST at L2)
- **Spacecraft mission design**: Halo orbits, Lissajous trajectories around Lagrange points
- **Choreography solutions**: Beautiful periodic N-body orbits with remarkable symmetries
- **Pure mathematics**: Connection to algebraic geometry (Groebner bases), Morse theory (topology of configuration space)
- **Computational topology**: Certified enumeration via homotopy continuation and symbolic computation

0.1.4 Pure Thought Advantages

Central configurations are **ideal for pure thought investigation**:

- Based on **algebraic equations** (polynomial system in positions)
- Solutions computable via **Groebner bases** and **homotopy continuation**
- Finiteness provable using **Bezout's theorem** and **dimension theory**
- Stability determined by **Hessian eigenvalues** (symbolic computation)
- All results **certified via interval arithmetic** or **exact algebraic numbers**
- NO numerical integration of trajectories until verification phase
- NO empirical search or Monte Carlo sampling

0.2 2. Mathematical Formulation

0.2.1 N-Body Equations

The Newton equations for N gravitating bodies:
mr = -U(r)
where U(r) = - mm/|r - r| is the gravitational potential.
Homothetic solutions: Seek r(t) = (t)c where c = (c,...,c$_N$)$isafixedshape.Substituting$:
m(c + 2ċ + c) = -^{2}U(c)
For c = 0 (fixed shape in rotating frame) and = -^{2} (harmonic oscillator), we get:
U(c) = mc
where = ^{2} > 0. This is the **central configuration equation**.

0.2.2 Central Configuration Equations

Definition: Positions r = (r,...,r$_N$)$()formacentralconfigurationif$:

```
U   (r) =     mr      for all i = 1,...,N
```

with and mr = 0 (center of mass at origin).
Equivalent formulation (Albouy-Chenciner):

```
m  ( r   - r  )/| r   - r  |  =  r    for all i
```

This is a **polynomial system** after clearing denominators (multiply by |r - r|³).
Degrees of freedom:

- () positions → dN variables
- Subtract d for center of mass → dN - d
- Subtract 1 for scale invariance (r r gives same config) → dN - d - 1
- Subtract SO(d) rotations → dN - d - 1 - d(d-1)/2

For planar (d=2), N=3: 2 · 3 - 2 - 1 - 1 = 2 DOF
For planar N=4: 2 · 4 - 2 - 1 - 1 = 4 DOF
For planar N=5: 2 · 5 - 2 - 1 - 1 = 6 DOF

0.2.3 Key Examples

- **Euler collinear (3-body)**: All bodies on a line r < r < r. 5 solutions for generic masses (3 with m in middle, 2 with m or m in middle).
- **Lagrange equilateral (3-body)**: r, r, r form equilateral triangle. Unique up to rotations.
- **Kite configuration (4-body)**: Diamond shape with symmetry axis. Family parametrized by mass ratios.
- **Square (4-body, equal masses)**: r, r, r, r at vertices of square. Unstable.
- **Figure-eight (3-body, equal masses)**: Choreography where bodies chase each other along figure-eight curve. Period 6.3 time units.

0.2.4 Stability Analysis

Linearization: Perturb r(t) = c + (t) and linearize equations of motion:
m = -U$|_c + \check{s}m$
where U is the Hessian of U.

Stability criterion (Lyapunov): Central configuration is stable if all eigenvalues of the Hessian (in rotating frame) have non-negative real parts, excluding zero modes (translations, rotations).

0.2.5 Certificates

All results must come with **machine-checkable certificates**:

- **Existence certificate**: Interval arithmetic proof that solution lies in certified box
- **Uniqueness certificate**: Homotopy continuation tracking all solution branches
- **Finiteness certificate**: Groebner basis showing ideal is zero-dimensional
- **Stability certificate**: Interval bounds on all Hessian eigenvalues

Export format: JSON with exact algebraic numbers (via SymPy):

```json
{
  "n_bodies": 4,
  "dimension": 2,
  "positions": [
    {"x": "1/2", "y": "sqrt(3)/2"},
    {"x": "-1/2", "y": "sqrt(3)/2"},
    ...
  ],
  "lambda": "3",
  "stability": "stable",
  "symmetry_group": "D3"
}
```

0.3 3. Implementation Approach

0.3.1 Phase 1 (Months 1-2): Lagrange Points (3-Body)

Goal: Find and analyze L1-L5 Lagrange points in circular restricted 3-body problem.

```python
import numpy as np
from scipy.optimize import fsolve
import sympy as sp
from mpmath import mp
mp.dps = 100  # 100-digit precision

def lagrange_points_circular_restricted(m1: float, m2: float) -> dict:
    """
    Find all five Lagrange points for circular restricted 3-body problem.

    Setup: m1 and m2 orbit their barycenter, m3     0 (test particle).
    Rotating frame: m1 at (-  , 0), m2 at (1-  , 0),    = m2/(m1+m2).

    L1, L2, L3: collinear (on x-axis)
    L4, L5: equilateral triangle vertices
    """
    mu = m2 / (m1 + m2)

    # Effective potential in rotating frame
    def U_eff(x, y):
        r1 = np.sqrt((x + mu)**2 + y**2)
        r2 = np.sqrt((x - 1 + mu)**2 + y**2)
        return 0.5 * (x**2 + y**2) + (1 - mu)/r1 + mu/r2

    # Collinear points: solve  U_eff  / x  = 0 with y = 0
    def collinear_equation(x):
        r1 = abs(x + mu)
        r2 = abs(x - 1 + mu)
        return x - (1 - mu) * (x + mu) / r1**3 - mu * (x - 1 + mu) / r2**3

    # L1: between m1 and m2
    x_L1 = fsolve(collinear_equation, -mu + 0.5*(1-mu), full_output=True)[0][0]

    # L2: beyond m2
    x_L2 = fsolve(collinear_equation, 1-mu + 0.1,
        full_output=True)[0][0]

    # L3: beyond m1
    x_L3 = fsolve(collinear_equation, -mu - 0.5, full_output=True)[0][0]

    # L4, L5: equilateral triangles (analytical)
    x_L4 = 0.5 - mu
    y_L4 = np.sqrt(3) / 2

    x_L5 = 0.5 - mu
    y_L5 = -np.sqrt(3) / 2

    return {
        'L1': np.array([x_L1, 0.0]),
        'L2': np.array([x_L2, 0.0]),
        'L3': np.array([x_L3, 0.0]),
        'L4': np.array([x_L4, y_L4]),
        'L5': np.array([x_L5, y_L5]),
        'mu': mu
    }


def lagrange_equilateral_3body_symbolic(m1, m2, m3) -> dict:
    """
    Symbolic computation of equilateral triangle central configuration.

    For arbitrary masses m1, m2, m3.
    """
    # Symbolic variables
    x1, y1, x2, y2, x3, y3 = sp.symbols('x1 y1 x2 y2 x3 y3', real=True)
    lam = sp.symbols('lambda', positive=True)

    # Positions
    r1 = sp.Matrix([x1, y1])
    r2 = sp.Matrix([x2, y2])
    r3 = sp.Matrix([x3, y3])

    # Gravitational potential gradient
    def grad_U_i(ri, rj, mi, mj):
        r_ij = ri - rj
        dist = sp.sqrt(r_ij.dot(r_ij))
        return mi * mj * r_ij / dist**3

    # Central configuration equations
    eqs = []

    # Body 1
    grad_U_1 = grad_U_i(r1, r2, m1, m2) + grad_U_i(r1, r3, m1, m3)
    eqs.extend(grad_U_1 - lam * m1 * r1)

    # Body 2
    grad_U_2 = grad_U_i(r2, r1, m2, m1) + grad_U_i(r2, r3, m2, m3)
    eqs.extend(grad_U_2 - lam * m2 * r2)
```

```python

    # Body 3
    grad_U_3 = grad_U_i(r3, r1, m3, m1) + grad_U_i(r3, r2, m3, m2)
    eqs.extend(grad_U_3 - lam * m3 * r3)

    # Center of mass constraint
    eqs.append(m1*x1 + m2*x2 + m3*x3)
    eqs.append(m1*y1 + m2*y2 + m3*y3)

    # Equilateral constraint
    eqs.append((x2-x1)**2 + (y2-y1)**2 - 1)  # |r2-r1| = 1 (normalize)
    eqs.append((x3-x1)**2 + (y3-y1)**2 - 1)  # |r3-r1| = 1
    eqs.append((x3-x2)**2 + (y3-y2)**2 - 1)  # |r3-r2| = 1

    # Solve using Groebner basis
    solution = sp.solve(eqs, [x1, y1, x2, y2, x3, y3, lam])

    return solution


def verify_central_configuration(positions: np.ndarray,
                                 masses: np.ndarray,
                                 tolerance: float = 1e-10) -> dict:
    """
    Verify that given positions form a central configuration.

    Checks:   U   =    mr    for all i with same   .
    """
    N = len(masses)
    grad_U = np.zeros_like(positions)

    # Compute potential gradient for each body
    for i in range(N):
        for j in range(N):
            if i == j:
                continue
            r_ij = positions[i] - positions[j]
            dist = np.linalg.norm(r_ij)
            grad_U[i] += masses[i] * masses[j] * r_ij / dist**3

    # Extract   from first body (non-zero position)
    i_nonzero = np.argmax(np.linalg.norm(positions, axis=1))
    lambda_computed = np.dot(grad_U[i_nonzero], positions[i_nonzero]) / (
        masses[i_nonzero] * np.dot(positions[i_nonzero], positions[i_nonzero])
    )

    # Check if   U   =    mr    for all i
    residuals = []
    for i in range(N):
        expected = lambda_computed * masses[i] * positions[i]
        residual = np.linalg.norm(grad_U[i] - expected)
        residuals.append(residual)

    max_residual = max(residuals)
    is_central = max_residual < tolerance

    return {
        'is_central_configuration': is_central,
        'lambda': lambda_computed,
        'max_residual': max_residual,
        'residuals': residuals
    }
```

Validation: Verify Lagrange L4/L5 for Earth-Moon system (0.012), check equilateral triangle property.

0.3.2 Phase 2 (Months 2-4): 4-Body Central Configurations

Goal: Find all planar central configurations for 4 bodies using numerical continuation.

```python
from scipy.optimize import fsolve, root
from sympy.solvers import solve
from sympy.polys.groebnertools import groebner

def find_4body_central_configurations(masses: np.ndarray,
                                      initial_guesses: list = None) ->
                                         list:
    """
    Find all planar central configurations for 4 bodies.

    Uses Albouy-Chenciner formulation + homotopy continuation.
    """
    N = 4
    m1, m2, m3, m4 = masses

    def central_config_equations(x):
        """
        System of equations for central configuration.

        x: [x1, y1, x2, y2, x3, y3, x4, y4, lambda]
        (9 variables for 4 bodies in 2D +    )
        """
        # Extract positions and
        positions = x[:8].reshape(4, 2)
        lam = x[8]

        # Center of mass at origin
        com_x = sum(masses[i] * positions[i, 0] for i in range(N))
        com_y = sum(masses[i] * positions[i, 1] for i in range(N))

        # Compute gradient of potential
        equations = []

        for i in range(N):
            grad_U_i = np.zeros(2)
            for j in range(N):
                if i == j:
                    continue
                r_ij = positions[i] - positions[j]
                dist = np.linalg.norm(r_ij)
                if dist < 1e-10:
                    return np.ones(9) * 1e10  # Collision, invalid

                grad_U_i += masses[j] * r_ij / dist**3

            # Central config condition: grad_U_i =    * m_i * r_i
            eqs_i = grad_U_i - lam * positions[i]
            equations.extend(eqs_i)

        # Add center of mass constraints
        equations.append(com_x)
        equations.append(com_y)

        # Normalization: fix scale (e.g., |r1| = 1)
        equations.append(np.linalg.norm(positions[0]) - 1.0)

        return np.array(equations)

    # Generate initial guesses if not provided
    if initial_guesses is None:
        initial_guesses = generate_4body_initial_guesses(masses)

    # Find solutions using multiple initial conditions
    solutions = []

    for guess in initial_guesses:
        try:
            sol = fsolve(central_config_equations, guess,
                full_output=True)
            x_sol = sol[0]
            info = sol[1]

            # Check if solution converged
            if info['fvec'].max() < 1e-8:
                # Verify it's a new solution (not duplicate)
                is_new = True
                for prev_sol in solutions:
                    if np.linalg.norm(x_sol - prev_sol) < 1e-6:
                        is_new = False
                        break

                if is_new:
                    solutions.append(x_sol)
        except:
            continue

    return solutions


def generate_4body_initial_guesses(masses: np.ndarray) -> list:
    """
    Generate initial guesses for 4-body central configurations.

    Known families:
    1. Collinear (all on a line)
    2. Isosceles trapezoid
    3. Kite (diamond with symmetry axis)
    4. Equilateral triangle + 1 body at centroid
    """
    guesses = []

    # Guess 1: Collinear
    positions = np.array([[-1.5, 0], [-0.5, 0], [0.5, 0], [1.5, 0]])
    guesses.append(np.concatenate([positions.flatten(), [1.0]]))

    # Guess 2: Square (equal masses)
    positions = np.array([[-1, -1], [1, -1], [1, 1], [-1, 1]]) /
        np.sqrt(2)
    guesses.append(np.concatenate([positions.flatten(), [1.0]]))

    # Guess 3: Isosceles trapezoid
    positions = np.array([[-1, -0.5], [1, -0.5], [0.5, 0.5], [-0.5,
        0.5]])
    guesses.append(np.concatenate([positions.flatten(), [1.0]]))

    # Guess 4: Kite
    positions = np.array([[0, -1], [-0.5, 0], [0, 1], [0.5, 0]])
    guesses.append(np.concatenate([positions.flatten(), [1.0]]))

    # Guess 5: Equilateral triangle + centroid
    positions = np.array([[0, 0], [1, 0], [0.5, np.sqrt(3)/2], [0.5,
        np.sqrt(3)/6]])
    guesses.append(np.concatenate([positions.flatten(), [1.0]]))

    return guesses


def classify_4body_configuration_symmetry(positions: np.ndarray,
                                          tolerance: float = 1e-6) ->
                                              str:
    """
    Classify 4-body configuration by symmetry group.

    Returns: Symmetry type (C1, C2, D2, D4, etc.)
    """
    N = 4

    # Check for reflection symmetries
    has_reflection_x = check_reflection_symmetry(positions, axis='x',
        tol=tolerance)
    has_reflection_y = check_reflection_symmetry(positions, axis='y',
        tol=tolerance)
    has_rotation_90 = check_rotational_symmetry(positions, angle=90,
        tol=tolerance)

    if has_rotation_90:
        return 'D4'  # Square
    elif has_reflection_x and has_reflection_y:
        return 'D2'  # Rectangle or kite
    elif has_reflection_x or has_reflection_y:
        return 'C2'  # Single axis of symmetry
    else:
        return 'C1'  # No symmetry


def check_reflection_symmetry(positions: np.ndarray, axis: str, tol: float) -> bool:
    """Check if configuration has reflection symmetry about given axis."""
    if axis == 'x':
        reflected = positions.copy()
        reflected[:, 1] *= -1
    elif axis == 'y':
        reflected = positions.copy()
        reflected[:, 0] *= -1
    else:
        raise ValueError(f"Unknown axis {axis}")

    # Check if reflected positions match original (up to permutation)
    for perm in permutations(range(len(positions))):
        if np.allclose(positions, reflected[list(perm), :], atol=tol):
            return True

    return False
```

Validation: Reproduce known 4-body central configs from Albouy Moeckel (2005), verify count matches literature.

0.3.3 Phase 3 (Months 4-5): Stability Analysis

Goal: Compute Hessian eigenvalues and classify stability of central configurations.

```
def stability_analysis_central_config(positions: np.ndarray,
                                      masses: np.ndarray,
                                      lambda_cc: float) -> dict:
    """
    Analyze linear stability of central configuration.

    Computes eigenvalues of Hessian in rotating frame.
    """
    N = len(masses)
    dim = positions.shape[1]  # 2 for planar, 3 for spatial

    # Compute Hessian of gravitational potential
    H = np.zeros((N * dim, N * dim))

    for i in range(N):
        for j in range(N):
            if i == j:
                # Diagonal block: H_ii = - _ { k i } m_k * (I/r_ik  - 3* r _ i k r_ik / r _ i k )
                for k in range(N):
                    if k == i:
                        continue
                    r_ik = positions[i] - positions[k]
                    dist = np.linalg.norm(r_ik)

                    I_block = np.eye(dim)
                    outer_block = np.outer(r_ik, r_ik) / dist**2

                    H_ik_block = masses[k] * (I_block / dist**3 - 3 *
                       outer_block / dist**3)

                    # Add to diagonal block
                    H[i*dim:(i+1)*dim, i*dim:(i+1)*dim] -= H_ik_block

            else:
                # Off-diagonal block: H_ij = m_j * (I/r_ij   -
                   3* r _ i j r_ij / r _ i j )
                r_ij = positions[i] - positions[j]
                dist = np.linalg.norm(r_ij)

                I_block = np.eye(dim)
                outer_block = np.outer(r_ij, r_ij) / dist**2

                H_ij_block = masses[j] * (I_block / dist**3 - 3 *
                   outer_block / dist**3)

                H[i*dim:(i+1)*dim, j*dim:(j+1)*dim] = H_ij_block

    # Add centrifugal term from rotating frame: +    * I
    H += lambda_cc * np.eye(N * dim)

    # Compute eigenvalues
    eigvals = np.linalg.eigvalsh(H)

    # Filter out zero modes (translations, rotations)
    # Expect dim + dim(dim-1)/2 zero eigenvalues
    n_zero_modes = dim + dim * (dim - 1) // 2

    eigvals_nonzero = eigvals[n_zero_modes:]

    # Stability: all non-zero eigenvalues should be non-negative
    is_stable = all(eigvals_nonzero >= -1e-8)

    # Count unstable modes (negative eigenvalues)
    n_unstable = sum(1 for ev in eigvals_nonzero if ev < -1e-8)

    return {
        'eigenvalues_all': eigvals,
        'eigenvalues_nonzero': eigvals_nonzero,
        'is_stable': is_stable,
        'n_unstable_modes': n_unstable,
        'stability_type': 'stable' if is_stable else f'unstable
           ({n_unstable} modes)'
    }


def morse_index_central_config(positions: np.ndarray,
                               masses: np.ndarray) -> int:
    """
    Compute Morse index of central configuration.

    Morse index = number of negative eigenvalues of Hessian of U
    (without centrifugal term).
    """
    N = len(masses)
    dim = positions.shape[1]

    # Compute Hessian of potential U (no    term)
    H_U = compute_potential_hessian(positions, masses)

    # Eigenvalues
    eigvals = np.linalg.eigvalsh(H_U)

    # Morse index: count negative eigenvalues (exclude zero modes)
    n_zero_modes = dim + dim * (dim - 1) // 2
    eigvals_nonzero = eigvals[n_zero_modes:]

    morse_index = sum(1 for ev in eigvals_nonzero if ev < -1e-8)

    return morse_index
```

Validation: Verify Lagrange L4/L5 are stable (all eigenvalues 0), L1/L2/L3 are unstable (at least one negative eigenvalue).

0.3.4 Phase 4 (Months 5-7): 5-Body Enumeration via Groebner Bases

Goal: Enumerate all planar 5-body central configurations using computational algebraic geometry.

```
import sympy as sp
from sympy.polys.groebnertools import groebner

def enumerate_5body_central_configs_symbolic(masses: list) -> list:
    """
    Enumerate all planar 5-body central configurations using Groebner
       bases.

    CAUTION: Computationally expensive for N=5.
    """
    N = 5
    m1, m2, m3, m4, m5 = masses

    # Symbolic variables
    coords = []
    for i in range(N):
        coords.extend([sp.Symbol(f'x{i}'), sp.Symbol(f'y{i}')])

    lam = sp.Symbol('lambda')

    # Build polynomial equations
    equations = []

    for i in range(N):
        x_i, y_i = coords[2*i], coords[2*i+1]
        r_i = sp.Matrix([x_i, y_i])

        # Compute gradient of U for body i
        grad_U_i = sp.Matrix([0, 0])
```

```python

        for j in range(N):
            if i == j:
                continue
            x_j, y_j = coords[2*j], coords[2*j+1]
            r_j = sp.Matrix([x_j, y_j])

            r_ij = r_i - r_j
            dist_sq = r_ij.dot(r_ij)

            # grad_U_i += masses[j] * r_ij / dist^3
            # Clear denominator: multiply by dist^3 = (dist_sq)^(3/2)
            # Use dist_sq instead to keep polynomial
            grad_U_i += masses[j] * r_ij * sp.sqrt(dist_sq)**(-3)

        # Central config condition: grad_U_i =    * masses[i] * r_i
        eqs_i = grad_U_i - lam * masses[i] * r_i
        equations.extend(eqs_i)

    # Center of mass constraints
    equations.append(sum(masses[i] * coords[2*i] for i in range(N)))
    equations.append(sum(masses[i] * coords[2*i+1] for i in range(N)))

    # Normalization constraint (fix scale)
    equations.append(coords[0]**2 + coords[1]**2 - 1)

    # Compute Groebner basis (WARNING: very slow for N=5)
    print("Computing Groebner basis (this may take hours)...")
    gb = groebner(equations, coords + [lam], order='lex')

    # Extract solutions from univariate polynomial in gb
    solutions = sp.solve(gb, coords + [lam])

    return solutions


def finiteness_certificate_central_configs(masses: np.ndarray, N: int) -> dict:
    """
    Certify finiteness of central configurations for N bodies with given masses.

    Uses Bezout's theorem and dimension analysis.
    """
    # Dimension of configuration space (after removing symmetries)
    dim_config_space = 2 * N - 4  # Planar, remove translations (2), rotations (1), scale (1)

    # Number of equations in central config system
    n_equations = 2 * N + 2  # 2N for central config, 2 for center of mass

    # Bezout bound (very loose upper bound on number of solutions)
    # For generic masses, expect finite number of solutions
    # if system is square (n_equations = dim_config_space)

    is_square = (n_equations == dim_config_space)

    return {
        'dimension': dim_config_space,
        'n_equations': n_equations,
        'is_square_system': is_square,
        'expected_finite': is_square,
        'bezout_bound_estimate': 'exponential in N (not computed)'
    }
```

Validation: Compare to literature counts for specific mass ratios (Hampton Moeckel 2006).

0.3.5 Phase 5 (Months 7-8): Symmetry Classification

Goal: Classify all central configurations by their symmetry groups.

```
from itertools import permutations
import networkx as nx

def classify_symmetry_group(positions: np.ndarray,
                            masses: np.ndarray,
                            tolerance: float = 1e-6) -> dict:
    """
    Determine the symmetry group of a central configuration.

    Returns: Group type (Cn, Dn, etc.) and generators.
    """
    N = len(masses)

    # Find all symmetries (isometries preserving configuration)
    symmetries = []

    # Check rotations
    for k in range(1, N):
        angle = 2 * np.pi * k / N
        rot_matrix = np.array([[np.cos(angle), -np.sin(angle)],
                               [np.sin(angle), np.cos(angle)]])

        is_symmetry = check_isometry_symmetry(positions, masses,
           rot_matrix, tolerance)
        if is_symmetry:
            symmetries.append(('rotation', angle))

    # Check reflections
    for angle in np.linspace(0, np.pi, 20):
        # Reflection about line through origin with angle
        refl_matrix = np.array([[np.cos(2*angle), np.sin(2*angle)],
                                [np.sin(2*angle), -np.cos(2*angle)]])

        is_symmetry = check_isometry_symmetry(positions, masses,
           refl_matrix, tolerance)
        if is_symmetry:
            symmetries.append(('reflection', angle))

    # Determine group type from symmetries
    n_rotations = sum(1 for s in symmetries if s[0] == 'rotation') + 1
       # Include identity
```

```python
    n_reflections = sum(1 for s in symmetries if s[0] == 'reflection')

    if n_reflections > 0 and n_rotations > 1:
        group_type = f'D{n_rotations}'  # Dihedral group
    elif n_rotations > 1:
        group_type = f'C{n_rotations}'  # Cyclic group
    elif n_reflections > 0:
        group_type = 'C_s'  # Single reflection
    else:
        group_type = 'C_1'  # No symmetry

    return {
        'symmetry_group': group_type,
        'n_rotational_symmetries': n_rotations,
        'n_reflection_symmetries': n_reflections,
        'all_symmetries': symmetries
    }


def check_isometry_symmetry(positions: np.ndarray,
                            masses: np.ndarray,
                            transformation: np.ndarray,
                            tolerance: float) -> bool:
    """
    Check if isometry (rotation/reflection) is a symmetry.

    A symmetry must preserve both positions AND masses.
    """
    N = len(masses)

    # Apply transformation
    transformed_positions = (transformation @ positions.T).T

    # Check if transformed positions match original under some permutation
    for perm in permutations(range(N)):
        # Check positions
        pos_match = np.allclose(positions, transformed_positions[list(perm), :],
                               atol=tolerance)

        # Check masses
        mass_match = np.allclose(masses, masses[list(perm)], atol=tolerance)

        if pos_match and mass_match:
            return True

    return False
```

Validation: Verify Lagrange equilateral triangle has D3 symmetry, square configuration has D4.

0.3.6 Phase 6 (Months 8-9): Certificate Generation and Database

Goal: Generate complete database of certified central configurations.

```python
import json
from dataclasses import dataclass, asdict
from typing import List

@dataclass
class CentralConfigurationCertificate:
    """Complete certificate for a central configuration."""

    # System parameters
    n_bodies: int
    dimension: int
    masses: List[float]

    # Configuration data
    positions: List[List[float]]  # N x d array
    lambda_value: float

    # Verification
    is_verified: bool
    max_residual: float
    verification_method: str  # 'numerical', 'symbolic',
        'interval_arithmetic'

    # Stability
    is_stable: bool
    eigenvalues: List[float]
    morse_index: int

    # Symmetry
    symmetry_group: str
    n_rotational_symmetries: int
    n_reflection_symmetries: int

    # Classification
    configuration_type: str  # 'collinear', 'planar', 'spatial',
        'equilateral', etc.

    # Metadata
    computation_date: str
    precision_digits: int

    def export_json(self, filename: str):
        """Export certificate to JSON."""
        with open(filename, 'w') as f:
            json.dump(asdict(self), f, indent=2)

    def verify(self) -> bool:
        """Self-check certificate validity."""
        checks = [
            self.n_bodies > 0,
            len(self.masses) == self.n_bodies,
            len(self.positions) == self.n_bodies,
            self.max_residual < 1e-6,
            len(self.eigenvalues) > 0
        ]
        return all(checks)


def generate_central_config_database(N_max: int = 5) -> list:
    """
    Generate database of all known central configurations for N
       N_max.
    """
    database = []

    # N=3 configurations
    for mass_ratio in [0.5, 1.0, 2.0, 5.0, 10.0]:
        masses = np.array([1.0, mass_ratio, mass_ratio])

        # Equilateral
        config_eq = compute_lagrange_equilateral(masses)
        cert_eq = create_certificate(config_eq, masses)
        database.append(cert_eq)

        # Collinear (multiple solutions)
        configs_col = compute_euler_collinear(masses)
        for config in configs_col:
            cert = create_certificate(config, masses)
            database.append(cert)

    # N=4 configurations
    if N_max >= 4:
        for mass_config in generate_4body_mass_configurations():
            configs = find_4body_central_configurations(mass_config)
            for config_array in configs:
                positions = config_array[:8].reshape(4, 2)
                cert = create_certificate(positions, mass_config)
                database.append(cert)

    return database
```

Validation: Export database to JSON, verify all certificates pass self-check.

0.4 4. Example Starting Prompt

Prompt for AI System:

You are tasked with enumerating and classifying central configurations for the N-body problem. Your goal is to:

- **Lagrange Points (Months 1-2)**:
- Implement circular restricted 3-body problem solver
- Find all five Lagrange points (L1-L5) for Earth-Moon system
- Verify L4/L5 form equilateral triangles
- Analyze stability: compute Hessian eigenvalues in rotating frame
- **4-Body Configurations (Months 2-4)**:

- Implement Albouy-Chenciner equations: U = mr
- Generate initial guesses: collinear, square, kite, trapezoid
- Use numerical continuation to find all solutions for given masses
- Verify each solution: check residuals < 10
- **Stability Analysis (Months 4-5)**:
- Compute Hessian of gravitational potential + centrifugal term
- Extract eigenvalues, filter zero modes (translations, rotations)
- Classify: stable (all eigenvalues 0) vs unstable
- Compute Morse index for each configuration
- **Symmetry Classification (Months 5-7)**:
- Check rotational symmetries: C2, C3, C4, ...
- Check reflection symmetries: , , $_d$

- Determine symmetry group: C1, Cn, Dn
- Verify: Lagrange equilateral has D3, square has D4
- **5-Body Enumeration (Months 7-8)**:
- Formulate central config equations as polynomial system
- Use Groebner basis to reduce system (WARNING: computationally expensive)
- Apply homotopy continuation for numerical solutions
- Certify finiteness: verify system is zero-dimensional
- **Certificate Generation (Months 8-9)**:
- Create CentralConfigurationCertificate for each solution
- Include: positions (exact algebraic numbers), , eigenvalues, symmetry group
- Export database to JSON
- Verify all certificates satisfy central config equations

Success Criteria:

- Minimum Viable Result (2-4 months): L1-L5 for Earth-Moon, basic 4-body configs
- Strong Result (6-8 months): Complete 4-body classification, stability analysis
- Publication-Quality Result (9 months): 5-body enumeration, certified database

Key Constraints:

- Use symbolic computation (SymPy) for exact solutions where possible

- Numerical solutions must have residuals < 10
- All eigenvalues computed with interval arithmetic bounds
- Database must include at least 50 certified configurations

References:

- Albouy Chenciner (2001): "Le problème des n corps et les distances mutuelles"
- Hampton Moeckel (2006): "Finiteness of relative equilibria in the planar four-body problem"
- Saari (2005): "Collisions, Rings, and Other Newtonian N-Body Problems"

Begin by implementing the circular restricted 3-body solver and finding L1-L5 for $= 0.012$ (Earth-Moon).

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (Months 1-4)

Core Achievements:

- L1-L5 Lagrange points computed for Earth-Moon system
- Verification: L4/L5 are equilateral triangles
- Basic 4-body central configurations found (collinear, square, kite)
- Stability analysis: eigenvalue computation implemented

Validation:

- Match L1 position to within 0.1
- Reproduce Hampton-Moeckel count for equal masses (N=4)

Deliverables:

- Python module `central`$_configs.pywithsolvers$
- Jupyter notebook demonstrating Earth-Moon L4/L5
- JSON database with 10+ certified configurations

0.5.2 Strong Result (Months 4-8)

Extended Capabilities:

- Complete 4-body enumeration for 5+ mass ratios
- Symmetry classification: C1, Cn, Dn groups
- Morse index computation for all configurations
- Stability boundaries: identify bifurcations as masses vary
- Comparison to 3+ literature sources

Publications Benchmark:

- Reproduce Figures from Albouy Moeckel (2005)
- Match eigenvalue spectra to within 1

Deliverables:

- Database with 50+ configurations
- Stability diagrams (mass ratio vs eigenvalues)
- Symmetry classification report

0.5.3 Publication-Quality Result (Months 8-9)

Novel Contributions:

- 5-body central config enumeration for specific masses
- Finiteness certificate via Groebner basis dimension
- Formal verification: translate proofs to Lean/Coq
- Interactive visualization tool (3D plots, animations)
- Public database: 100+ configurations with certificates

Beyond Literature:

- Discover new 5-body configurations not in literature
- Improve computational methods (faster Groebner basis)
- Extend to spatial (3D) configurations

Deliverables:

- Arxiv preprint: "Complete Classification of Planar N-Body Central Configurations"
- GitHub repository with database and solvers
- Web interface: input masses → visualize all central configs

0.6 6. Verification Protocol

```python
def verify_central_config_database(database: list) -> dict:
    """
    Automated verification of entire database.
    """
    results = {
        'n_total': len(database),
        'n_verified': 0,
        'n_failed': 0,
        'failed_certificates': []
    }

    for cert in database:
        # Check 1: Self-verification
        if not cert.verify():
            results['n_failed'] += 1
            results['failed_certificates'].append(cert)
            continue

        # Check 2: Central config equations satisfied
        positions = np.array(cert.positions)
        masses = np.array(cert.masses)

        verification = verify_central_configuration(positions,
           masses)

        if not verification['is_central_configuration']:
            results['n_failed'] += 1
            results['failed_certificates'].append(cert)
            continue

        # Check 3: Eigenvalue count matches dimension
        n_expected_eigenvalues = cert.n_bodies * cert.dimension
        if len(cert.eigenvalues) != n_expected_eigenvalues:
            results['n_failed'] += 1
            results['failed_certificates'].append(cert)
            continue

        results['n_verified'] += 1

    return results
```

0.7 7. Resources and Milestones

0.7.1 Essential References

- **Classical Papers**:
- Euler (1767): Collinear solutions
- Lagrange (1772): Equilateral triangle

- Moulton (1910): Families of periodic orbits
- **Modern Theory**:
- Albouy Chenciner (2001): "Le problème des n corps"
- Hampton Moeckel (2006): "Finiteness of relative equilibria"
- Moeckel Simó (1995): "Bifurcation of spatial central configurations"
- **Textbooks**:
- Saari (2005): *Collisions, Rings, and Other Newtonian N-Body Problems*
- Meyer, Hall, Offin (2009): *Introduction to Hamiltonian Dynamical Systems*

0.7.2 Milestone Checklist

Month 1: L1-L5 computed for Earth-Moon

Month 2: 3-body collinear solutions verified

Month 3: 4-body solver implemented

Month 4: First 10 configurations certified

Month 5: Stability analysis complete

Month 6: Symmetry classification implemented

Month 7: 50+ configurations in database

Month 8: 5-body enumeration begun

Month 9: Final database exported, all certificates verified

End of PRD 27