PRD 28: Nekhoroshev Stability and Exponential Timescales

Pure Thought AI Challenge 28

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 Hamiltonian Dynamics
Timeline: 6-9 months
Difficulty: High
Prerequisites: Hamiltonian mechanics, perturbation theory, Fourier analysis, symplectic geometry

---

0.1 1. Problem Statement

0.1.1 Scientific Context

Nekhoroshev stability theory (1977) provides a fundamental complement to KAM theory for understanding long-term stability in nearly-integrable Hamiltonian systems. While KAM theory guarantees eternal stability on measure-large invariant tori, it fails near resonances where tori break down. Nekhoroshev theory fills this gap by proving that even in resonant regions, the system exhibits **super-exponentially slow diffusion** over timescales that grow exponentially with the inverse perturbation strength.

For a near-integrable Hamiltonian H = H(I) + H(I,), Nekhoroshev's theorem states:

Main Result: If H satisfies a *steepness* (quasi-convexity) condition, then for all initial conditions and times |t| < T$_e xp\ exp(^{-a}), theactionvariablesremainclosetotheirinitialvalues:$

|I(t) - I(0)| < ^{b}

where a, b > 0 depend on dimension and the steepness properties of H.

This result has profound implications for **solar system stability**: with $10^{-3}(ratio of planetary to solar mas$

0.1.2 Core Question

Can we rigorously verify Nekhoroshev stability conditions for realistic Hamiltonian systems and compute explicit exponential stability timescales?

Key challenges:

- **Steepness verification**: Checking H is steep (quasi-convex) requires proving det(^{2}H/I^2) > C > 0 globally
- **Optimal exponents**: Constants a, b depend on dimension and steepness in complex ways
- **Resonance structure**: Exponential time depends on Fourier spectrum of H
- **N-planet problem**: Solar system requires handling multiple gravitational perturbations
- **Certificate generation**: Stability bounds must be machine-checkable with interval arithmetic

0.1.3 Why This Matters

- **Celestial mechanics**: Explains stability of solar system over Gyr timescales
- **Accelerator physics**: Particle beam stability in synchrotrons, colliders
- **Plasma confinement**: Charged particle motion in tokamaks
- **Astrodynamics**: Long-term satellite orbit prediction
- **Mathematical physics**: Universal mechanism for slow chaos in Hamiltonian systems

0.1.4 Pure Thought Advantages

Nekhoroshev theory is **ideal for pure thought investigation**:

- Based on **symbolic perturbation theory** (no numerical integration needed)
- Steepness conditions verifiable via **computer algebra** (exact Hessian computation)
- Exponential estimates computed from **Fourier coefficients** (symbolic)
- All bounds **certified via interval arithmetic** (rigorous error control)
- NO numerical simulations until final verification phase
- NO empirical fitting of stability times

0.2 2. Mathematical Formulation

0.2.1 Hamiltonian Setup

Consider a nearly-integrable Hamiltonian on the phase space (I,) ^{n}Ö^{n} :
H(I,) = H(I) + H(I,)
where:

- H(I): integrable part (e.g., Kepler Hamiltonian for planets)
- H(I,): perturbation (e.g., planet-planet gravitational interactions)
- $>$ 0: small parameter (typically $10^{-3} forsolarsystem$)

Hamilton's equations:

```
dI/dt = - H /       = -    H    /
d  /dt =  H  / I   =   (I) +     H    /  I
```

where (I) = H/I are the unperturbed frequencies.

0.2.2 Steepness Conditions

Definition (Steepness): H is **steep** (or quasi-convex) if there exists a convex function S(I) such that:
$|^{|}|S/I^{|C}$
for all multi-indices || 3, and the Hessian satisfies:
det(^{2}S/I^2) m > 0
uniformly on a domain D ^{n}.
Key Examples:

- **Strictly convex**: H(I) = ½⟨I, AI⟩ with A positive definite (harmonic oscillators)
- **Kepler problem**: H = -/(2I) (steep in I > 0)
- **Non-convex but steep**: Many physical Hamiltonians satisfy weaker quasi-convexity

Verification Strategy: Use symbolic differentiation to compute ^{2}H/I^2 exactly, then prove positivity via:

- Interval arithmetic bounds on eigenvalues
- SOS (sum-of-squares) decomposition
- Gröbner basis elimination

0.2.3 Nekhoroshev Theorem

Theorem (Nekhoroshev 1977): Let H = H(I) + H(I,) with H steep and H real-analytic. Then there exist constants a, b, C, > 0 such that for all $<$ and all initial conditions (I,) D $\times$ ^{n} :

|I(t) - I| $<$ $^b forall|t| < T_exp := Cexp(^{-a})$

Exponents:

- **Steep case** (convex): a = 1/(2n), b = 1/(2n)
- **Super-steep case** (exponentially convex): a = 1/2, b $\to$ 1/2
- **General quasi-convex**: a 1/(2n log(1/))

Interpretation: Actions diffuse at most $^b overexponentiallylongtimes.For = 10^{-3}, n = 5, T_exp$ $exp(10^{3/10})$ $10^13 yearsageofuniverse.$

0.2.4 Resonance Width Formula

Near a resonance k · (I) 0 (k $^n integervector), the perturbation H has significant Fourier component :$

H$k(I) = {}^2 ... {}^2 H(I,)e^{-ik\mathring{u}}d...dn$

Resonance width *(in action space):*

k $(|Hk|)^{1/2}$

Diffusion mechanism*: Actions can drift by* $_k whentrajectoryspendstime$ $1/|k\mathring{u}|neartheresonance.$

Nekhoroshev's key insight*: Exponential growth* $exp(^{-a})arisesfromthenumber$ *of* $resonancesthesystemm$

0.2.5 Certificates

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

- ***Steepness certificate****: Interval arithmetic proof that min eigenvalue($^2H/I^2$) $> m > 0$ on domain*
- ***Fourier bound certificate****: Rigorous bounds on |Hk| for all |k| Kmax*
- ***Exponential time certificate****: Lower bound Texp Tmin from certified constants a, b, C*
- ***Diffusion bound certificate****: Upper bound* $sup_{tT}|I(t) - I(0)| <^b witherrormargins$

Export format*: JSON with rational/interval arithmetic entries:*

```
{
  "steepness_constant": {"lower": "0.95", "upper": "1.05"},
  "exponent_a": {"value": "0.1", "precision": "1e-3"},
  "exponential_time_years": {"lower": "1e15", "infinite": false},
  "diffusion_bound_AU": {"value": "1e-8", "certified": true}
}
```

0.3 3. Implementation Approach

0.3.1 Phase 1 (Months 1-2): Steepness Verification

***Goal**: Symbolically compute Hessian of H and prove steepness.*

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

def compute_steepness_certificate(H0_symbolic: sp.Expr,
                                  action_vars: list,
                                  domain: dict) -> dict:
    """
    Verify  H   is steep by proving    H   /   I    is positive definite.

    Args:
        H0_symbolic: Symbolic expression for  H  (I)
        action_vars: List of action variables [I1, I2, ..., In]
        domain: Dictionary {I1: (min, max), I2: (min, max), ...}

    Returns:
        Certificate with minimum eigenvalue bounds
    """
    n = len(action_vars)

    # Compute Hessian symbolically
    hessian = sp.Matrix(n, n, lambda i, j:
                        sp.diff(H0_symbolic, action_vars[i], action_vars[j]))

    print(f"Symbolic Hessian computed: {hessian}")

    # Eigenvalue bounds via interval arithmetic
    from mpmath import iv

    min_eigenvalue = float('inf')
    max_eigenvalue = float('-inf')

    # Sample domain with interval arithmetic grid
    n_samples = 20
    for I_point in generate_interval_grid(domain, n_samples):
        # Substitute interval values
        hessian_interval = evaluate_matrix_interval(hessian, action_vars, I_point)

        # Compute eigenvalue bounds
        eigvals_interval = compute_eigenvalue_bounds_interval(hessian_interval)

        min_eigenvalue = min(min_eigenvalue, eigvals_interval['min'])
        max_eigenvalue = max(max_eigenvalue, eigvals_interval['max'])

    is_steep = min_eigenvalue > 0

    return {
        'is_steep': is_steep,
        'min_eigenvalue': min_eigenvalue,
        'max_eigenvalue': max_eigenvalue,
        'steepness_constant': min_eigenvalue if is_steep else None,
        'certificate_type': 'interval_arithmetic',
        'precision_digits': mp.dps
    }


def kepler_hamiltonian_steepness(n_planets: int) -> dict:
    """
    Verify steepness for n-planet Kepler Hamiltonian.

     H   =       -      /(2 I   )  (Kepler terms for each planet)

    Hessian:     H    /      II        =      /  I
        (diagonal, positive definite)
    """
    # Symbolic variables
    I_vars = sp.symbols(f'I1:{n_planets+1}', positive=True, real=True)
    mu_vars = sp.symbols(f'mu1:{n_planets+1}', positive=True, real=True)

    # Kepler Hamiltonian
    H0 = sum(-mu_vars[i] / (2*I_vars[i]) for i in range(n_planets))

    # Compute steepness
    domain = {I_vars[i]: (0.1, 10.0) for i in range(n_planets)}  #
        AU-scale actions

    cert = compute_steepness_certificate(H0, I_vars, domain)

    return cert


def interval_eigenvalue_bound_symmetric(A_intervals: np.ndarray) ->
    dict:
    """
    Compute rigorous eigenvalue bounds for symmetric interval matrix.

    Uses Gershgorin circle theorem with interval arithmetic.
    """
    n = A_intervals.shape[0]

    lambda_min = float('inf')
    lambda_max = float('-inf')

    for i in range(n):
        # Gershgorin disk center: diagonal entry
        center = A_intervals[i, i]

        # Radius: sum of off-diagonal absolute values
        radius = sum(abs(A_intervals[i, j]) for j in range(n) if j != i)

        lambda_min = min(lambda_min, center.a - radius.b)  # Lower bound
        lambda_max = max(lambda_max, center.b + radius.b)  # Upper bound

    return {'min': lambda_min, 'max': lambda_max}
```

***Validation**: Test on harmonic oscillator $H = ½^2I^2$ (should give min eigenvalue = min ^{2}).*

0.3.2 Phase 2 (Months 2-3): Resonance Analysis

***Goal**: Compute Fourier spectrum of perturbation H and estimate resonance widths.*

```
def compute_fourier_coefficients_perturbation(H1_symbolic: sp.Expr,
                                               angle_vars: list,
                                               k_max: int = 10) -> dict:
    """
    Compute Fourier coefficients  H   (I) for |k|    k_max.

     H  (I,  ) =         H   (I) e^{ i k   }
    """
    n = len(angle_vars)

    fourier_coeffs = {}

    for k in generate_integer_vectors(n, k_max):
        # Integrate  H  (I,  ) * e^{- i k   } over     ^n
        integrand = H1_symbolic * sp.exp(-sp.I * sum(k[i]*angle_vars[i]
                                                      for i in range(n)))

        # Symbolic integration (may be expensive)
        H1_k = (1/(2*sp.pi)**n) * sp.integrate(integrand,
                                                *[(theta, 0, 2*sp.pi)
                                                  for theta in
                                                     angle_vars])

        fourier_coeffs[tuple(k)] = H1_k

    return fourier_coeffs


def planetary_perturbation_hamiltonian(planets: list) -> sp.Expr:
    """
    Construct  H  for planet-planet gravitational perturbations.

     H  = -G     <      m  m  / | r   -  r  |

    Expand in Legendre polynomials.
    """
    n = len(planets)

    # Action-angle coordinates
    I_vars = sp.symbols(f'I1:{n+1}', positive=True)
    theta_vars = sp.symbols(f'theta1:{n+1}', real=True)

    # Convert to Cartesian (via Delaunay elements)
    positions = [action_angle_to_cartesian(I_vars[i], theta_vars[i])
                 for i in range(n)]

    H1 = 0
```

```python
    for i in range(n):
        for j in range(i+1, n):
            r_ij = positions[i] - positions[j]
            r_ij_norm = sp.sqrt(r_ij.dot(r_ij))

            H1 += -planets[i]['G'] * planets[i]['mass'] * planets[j]['mass'] / r_ij_norm

    # Expand to desired order in
    H1_expanded = sp.series(H1, planets[0]['mass']/planets[0]['M_sun'], 0, n=3).removeO()

    return H1_expanded


def resonance_width_estimate(k: np.ndarray,
                             epsilon: float,
                             H1_k: float) -> float:
    """
    Width of k-resonance in action space.

          ~    (  |  H    |) / |k|
    """
    width = np.sqrt(epsilon * abs(H1_k)) / np.linalg.norm(k)

    return width


def resonance_overlap_criterion(resonance_widths: dict,
                                frequency_map: callable) -> bool:
    """
    Check if resonances overlap (Chirikov criterion).

    Overlap              +          > |I_{ k  } - I_{ k  }|

    where I_k is center of k-resonance.
    """
    k_vectors = list(resonance_widths.keys())

    for i, k1 in enumerate(k_vectors):
        for k2 in k_vectors[i+1:]:
            # Resonance centers (solve  k   (I) = 0)
            I_k1 = find_resonance_center(k1, frequency_map)
            I_k2 = find_resonance_center(k2, frequency_map)

            if I_k1 is None or I_k2 is None:
                continue

            # Check overlap
            separation = np.linalg.norm(I_k1 - I_k2)
            combined_width = resonance_widths[k1] + resonance_widths[k2]

            if combined_width > separation:
                return True  # Resonances overlap     no Nekhoroshev stability

    return False  # Well-separated resonances
```

Validation: *Compute Fourier spectrum for Jupiter-Saturn perturbation, verify dominant k = (5, -2) Great Inequality resonance.*

0.3.3 Phase 3 (Months 3-4): Exponential Time Estimates

Goal: *Compute optimal exponents a, b and stability time T_exp.*

```
def nekhoroshev_exponents_optimal(dimension: int,
                                  steepness_type: str,
                                  epsilon: float) -> dict:
    """
    Compute optimal Nekhoroshev exponents a, b.

    Args:
        dimension: Number of degrees of freedom n
        steepness_type: 'convex', 'steep', 'quasi_convex'
        epsilon: Perturbation parameter

    Returns:
        Exponents a, b and constants C
    """
    if steepness_type == 'convex':
        # Best case: strictly convex  H
        a = 1 / (2 * dimension)
        b = 1 / (2 * dimension)
        C = 1.0

    elif steepness_type == 'steep':
        # Quasi-convex (most physical systems)
        a = 1 / (2 * dimension)
        b = 1 / (4 * dimension)  # Worse diffusion bound
        C = 0.5

    elif steepness_type == 'super_steep':
        # Exponentially convex (rare)
        a = 1 / 2
        b = 1 / 2
        C = 2.0

    else:
        # Generic quasi-convex with logarithmic corrections
        log_factor = np.log(1/epsilon) if epsilon > 0 else 1
        a = 1 / (2 * dimension * log_factor)
        b = 1 / (4 * dimension)
        C = 0.1

    return {'a': a, 'b': b, 'C': C, 'type': steepness_type}


def compute_exponential_stability_time(epsilon: float,
                                       exponents: dict,
                                       time_unit: str = 'years') ->
                                           dict:
    """
    Compute T_exp = C exp((     /   )^a).
    """
    a = exponents['a']
    C = exponents['C']

    # Characteristic scale       (depends on system)
    epsilon_0 = 1.0  # Normalized units

    # Exponential time
    if epsilon > 0:
        T_exp_normalized = C * np.exp((epsilon_0 / epsilon)**a)
    else:
        T_exp_normalized = float('inf')

    # Convert to physical units
    if time_unit == 'years':
        orbital_period = 1.0  # Normalize to 1 year for outer planets
        T_exp_years = T_exp_normalized * orbital_period
    else:
        T_exp_years = T_exp_normalized

    return {
        'T_exp_normalized': T_exp_normalized,
        'T_exp_years': T_exp_years,
        'exponent_a': a,
        'log_T_exp': a * np.log(1/epsilon) if epsilon > 0 else
            float('inf')
    }


def solar_system_nekhoroshev_stability() -> dict:
    """
    Apply Nekhoroshev theory to the solar system.

    Key parameters:
    - n = 8 planets (neglect Mercury as interior planet)
    -     ~ m_Jupiter / m_Sun ~ 10^{-3}
    -  H  : sum of Kepler Hamiltonians (steep)
    -  H  : planetary perturbations (real-analytic)
    """
    # System parameters
    n_planets = 8
    epsilon = 1e-3  # Jupiter mass / Sun mass

    # Nekhoroshev exponents for 8-planet system
    exponents = nekhoroshev_exponents_optimal(
        dimension=n_planets,
        steepness_type='steep',  # Kepler H   is steep
        epsilon=epsilon
    )

    # Compute exponential time
    stability = compute_exponential_stability_time(epsilon, exponents)

    # Compare to solar system age
    age_solar_system_years = 4.5e9
    age_universe_years = 13.8e9

    stability_margin = stability['T_exp_years'] / age_solar_system_years

    return {
        'dimension': n_planets,
        'perturbation_parameter': epsilon,
        'exponent_a': exponents['a'],
        'exponent_b': exponents['b'],
        'T_exp_years': stability['T_exp_years'],
        'age_solar_system_years': age_solar_system_years,
        'stability_margin': stability_margin,
        'verdict': 'STABLE' if stability_margin > 10 else 'UNSTABLE'
    }
```

Validation: *Reproduce* $T_exp\ exp(10^{3/10})\ 10^13 years for solar system (matches literature estimates)$.

0.3.4 Phase 4 (Months 4-6): Action Diffusion Bounds

Goal: *Prove rigorous upper bounds |I(t) - I(0)|* $< {}^b fort < T_exp$.

```
def action_diffusion_bound_certificate(H0: callable,
                                       H1: callable,
                                       epsilon: float,
                                       time_horizon: float,
                                       initial_action: np.ndarray) -> dict:
    """
    Generate certificate for action diffusion bound.

    Proves: |I(t) - I  | <   ^b for all t     T_horizon
    """
    # Compute Nekhoroshev constants
    dimension = len(initial_action)
    exponents = nekhoroshev_exponents_optimal(dimension, 'steep', epsilon)

    a, b = exponents['a'], exponents['b']

    # Check time is within exponential bound
    T_exp = compute_exponential_stability_time(epsilon, exponents)['T_exp_normalized']

    if time_horizon > T_exp:
        return {
            'certified': False,
            'reason': f'Time horizon {time_horizon} exceeds T_exp = {T_exp}'
        }

    # Diffusion bound: |I(t) - I  | <   ^b
    diffusion_bound = epsilon**b

    # Certificate using interval arithmetic propagation
    I_interval = propagate_actions_interval_arithmetic(
        H0, H1, epsilon, initial_action, time_horizon
    )

    max_deviation = max(abs(I_interval[i].b - initial_action[i])
                       for i in range(dimension))

    certified = max_deviation < diffusion_bound

    return {
        'certified': certified,
        'diffusion_bound': diffusion_bound,
        'max_deviation_computed': max_deviation,
        'time_horizon': time_horizon,
        'T_exp': T_exp,
        'safety_margin': diffusion_bound / max_deviation if certified
           else 0
    }


def propagate_actions_interval_arithmetic(H0: callable,
                                          H1: callable,
                                          epsilon: float,
                                          I0: np.ndarray,
                                          T: float,
                                          n_steps: int = 1000) -> list:
    """
    Propagate actions using interval arithmetic to get rigorous bounds.

    dI/dt = -      H    /

    Returns: List of interval boxes [I_min, I_max] at time T
    """
    from mpmath import iv

    dt = T / n_steps
    dimension = len(I0)

    # Initialize interval boxes
    I_intervals = [iv.mpf([I0[i], I0[i]]) for i in range(dimension)]

    for step in range(n_steps):
        # Compute RHS of dI/dt using interval arithmetic
        # (requires interval evaluation of    H   /     )
        dI_dt_intervals = compute_action_derivative_interval(
            H1, I_intervals, epsilon
        )

        # Euler step with interval arithmetic
        for i in range(dimension):
            I_intervals[i] += dt * dI_dt_intervals[i]

    return I_intervals
```

Validation: *Verify bounds for 2-planet system (Jupiter-Saturn) over 1 Gyr, compare to numerical integration.*

0.3.5 Phase 5 (Months 6-8): Optimal Constants and Sharpness

Goal: *Determine optimal (smallest) exponents a achieving given stability time.*

```
def optimize_nekhoroshev_constants(H0: callable,
                                    H1: callable,
                                    epsilon: float,
                                    desired_time: float) -> dict:
    """
    Find optimal constants a, b, C in Nekhoroshev estimate.

    Goal: Maximize a (sharper result) subject to T_exp      desired_time.
    """
    from scipy.optimize import minimize_scalar

    dimension = estimate_dimension(H0)

    def objective(a_trial):
        # For given a, compute achievable T_exp
        C = 1.0  # Fix normalization
        T_exp = C * np.exp((1/epsilon)**a_trial)

        # Penalty if T_exp < desired_time
        if T_exp < desired_time:
            return 1e10  # Infeasible
        else:
            return -a_trial  # Maximize a

    # Optimize over reasonable range
    result = minimize_scalar(objective, bounds=(0.01, 1.0),
        method='bounded')

    a_optimal = result.x
    b_optimal = a_optimal  # Typically b ~ a for optimal results

    return {
        'a_optimal': a_optimal,
        'b_optimal': b_optimal,
        'T_exp_achieved': np.exp((1/epsilon)**a_optimal),
        'desired_time': desired_time,
        'optimality': 'sharp' if result.fun < -0.1 else 'conservative'
    }


def compare_to_numerical_integration(H_total: callable,
                                      initial_conditions: np.ndarray,
                                      T_max: float,
                                      nekhoroshev_bound: float) -> dict:
    """
    Validate Nekhoroshev bound against numerical integration.

    Integrate Hamilton's equations and check |I(t) - I(0)| < bound.
    """
    from scipy.integrate import solve_ivp

    def hamiltonian_flow(t, y):
        # y = [I,    ]
        dimension = len(y) // 2
        I, theta = y[:dimension], y[dimension:]

        dI_dt = -compute_dH_dtheta(H_total, I, theta)
        dtheta_dt = compute_dH_dI(H_total, I, theta)

        return np.concatenate([dI_dt, dtheta_dt])

    # Integrate
    sol = solve_ivp(hamiltonian_flow,
                    (0, T_max),
                    initial_conditions,
                    method='DOP853',  # High-accuracy
                    rtol=1e-12, atol=1e-14)

    # Extract action variables
    dimension = len(initial_conditions) // 2
    I_trajectory = sol.y[:dimension, :]
    I_initial = initial_conditions[:dimension]

    # Compute maximum deviation
    max_deviation = np.max(np.linalg.norm(I_trajectory - I_initial[:,
        np.newaxis], axis=0))

    # Compare to Nekhoroshev bound
    bound_satisfied = max_deviation < nekhoroshev_bound

    return {
        'max_deviation': max_deviation,
        'nekhoroshev_bound': nekhoroshev_bound,
        'bound_satisfied': bound_satisfied,
        'safety_factor': nekhoroshev_bound / max_deviation if
            max_deviation > 0 else float('inf'),
        'integration_time': T_max,
        'n_timesteps': len(sol.t)
    }
```

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

Goal: *Generate machine-checkable certificates for all stability results.*

```
import json
from dataclasses import dataclass, asdict

@dataclass
class NekhoroshevCertificate:
    """Complete Nekhoroshev stability certificate."""

    # System identification
    hamiltonian_name: str
    dimension: int
    perturbation_parameter: float

    # Steepness certificate
    is_steep: bool
    steepness_constant: float
    steepness_proof_method: str  # 'interval_arithmetic', 'SOS',
        'symbolic'

    # Nekhoroshev constants
    exponent_a: float
    exponent_b: float
    constant_C: float

    # Stability estimates
    exponential_time_normalized: float
    exponential_time_years: float
    diffusion_bound: float

    # Verification
    numerical_validation: bool
    max_deviation_observed: float
    integration_time_years: float

    # Metadata
    computation_date: str
    precision_digits: int
    certificate_version: str

    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.is_steep,
            self.steepness_constant > 0,
            self.exponent_a > 0,
            self.exponent_b > 0,
            self.exponential_time_normalized > 0,
            self.diffusion_bound > 0
        ]

        if self.numerical_validation:
            checks.append(self.max_deviation_observed <
                self.diffusion_bound)

        return all(checks)


def generate_solar_system_certificate() -> NekhoroshevCertificate:
    """
    Generate complete Nekhoroshev certificate for solar system.
    """
    # Run all computations
    steepness = kepler_hamiltonian_steepness(n_planets=8)
    stability = solar_system_nekhoroshev_stability()

    # Numerical validation (expensiveuse reduced time)
    validation_time = 1e6  # 1 Myr (much less than T_exp but feasible)
```

```python
    # validation = compare_to_numerical_integration(...)  # Commented for speed

    cert = NekhoroshevCertificate(
        hamiltonian_name='Solar System (8 planets)',
        dimension=8,
        perturbation_parameter=1e-3,

        is_steep=steepness['is_steep'],
        steepness_constant=steepness['steepness_constant'],
        steepness_proof_method='interval_arithmetic',

        exponent_a=stability['exponent_a'],
        exponent_b=stability['exponent_b'],
        constant_C=1.0,

        exponential_time_normalized=stability['T_exp_years'] / 1e9,  # In Gyr
        exponential_time_years=stability['T_exp_years'],
        diffusion_bound=1e-3**stability['exponent_b'],  # AU

        numerical_validation=False,  # Set to True after running validation
        max_deviation_observed=0.0,
        integration_time_years=validation_time,

        computation_date='2026-01-17',
        precision_digits=100,
        certificate_version='1.0'
    )

    return cert
```

Validation: *Export certificate, verify all fields satisfy logical constraints.*

0.4 4. Example Starting Prompt

Prompt for AI System:

You are tasked with applying Nekhoroshev stability theory to verify exponential-time stability of the solar system. Your goal is to:

- ***Verify Steepness*** *(Months 1-2):*
- *Construct the integrable Hamiltonian* $H =$ *-GMm/(2I) for 8 planets*
- *Compute the Hessian* $^2H/I^2$ *symbolically using SymPy*
- *Prove steepness by showing all eigenvalues are positive using interval arithmetic*
- *Generate a steepness certificate with rigorous bounds: min eigenvalue* $> C > 0$
- ***Analyze Perturbations*** *(Months 2-3):*
- *Construct the perturbation Hamiltonian H representing planet-planet gravitational interactions*

  - *Expand H in action-angle coordinates using Delaunay elements*
  - *Compute Fourier coefficients H for |k| 10 using symbolic integration*
  - *Identify dominant resonances (e.g., Jupiter-Saturn 5:2 Great Inequality)*
  - *Estimate resonance widths: = (|H|)*
  - ***Compute Exponential Times** (Months 3-4):*
  - *Determine optimal Nekhoroshev exponents for n=8 dimensions: a = 1/(2n) = 1/16*
  - *Calculate exponential stability time:* $T_e xp = exp((1/)^a) with = 10^{-3}$
- *Convert to physical units:* $T_e xp 10^1 3 years$
- *Compare to solar system age (4.5 Gyr) and verify stability margin > 10*^{3}
- ***Prove Diffusion Bounds** (Months 4-6):*
- *Use interval arithmetic to propagate action variables forward in time*
- *Prove |I(t) - I(0)| <* $^b = (10^{-3})^{1/16} 0.7 AU f or t < T_e xp$
- *Generate certificate with rigorous error bounds using mpmath (100-digit precision)*
- *Validate against numerical integration of Hamilton's equations over 1 Myr*
- ***Optimize Constants** (Months 6-8):*
- *Search for optimal (largest) exponent a achieving desired stability time*
- *Compare to best-known theoretical results (Niederman 2004, Guzzo et al. 2011)*
- *Identify sharpness: is a = 1/(2n) optimal or can it be improved?*
- ***Certificate Generation** (Months 8-9):*
- *Create NekhoroshevCertificate object containing all results*
- *Export to JSON with interval arithmetic bounds and metadata*
- *Self-verify certificate: check all constraints satisfied*
- *Compare to literature: reproduce Guzzo et al. (2005)* $T_e xpestimates f or Jupiter - Saturn$

***Success Criteria**:*

- *Minimum Viable Result (2-4 months): Steepness verified for Kepler Hamiltonian, basic exponential time estimate*
- *Strong Result (6-8 months): Full solar system analysis with rigorous diffusion bounds and numerical validation*
- *Publication-Quality Result (9 months): Optimal exponents, comparison to literature, machine-checkable certificates*

***Key Constraints**:*

- *Use ONLY symbolic mathematics and interval arithmetic (no floating-point until final validation)*
- *All bounds must be certified with explicit error margins*
- *Compare to at least 3 literature sources (Nekhoroshev 1977, Niederman 2004, Guzzo+ 2011)*
- *Generate JSON export for certificate database*

References:

- *Nekhoroshev (1977): Original theorem and proof outline*
- *Niederman (2004): Optimal exponents and steepness conditions*
- *Guzzo, Lega, Froeschlé (2005): Solar system application and numerical validation*
- *Morbidelli (2002): Modern Celestial Mechanics textbook treatment*

Begin by symbolically computing the Hessian of the Kepler Hamiltonian and proving steepness using interval arithmetic.

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (Months 1-4)

Core Achievements:

- *Symbolic Hessian computation for n-planet Kepler Hamiltonian*
- *Steepness verification: min eigenvalue > 0 certified via interval arithmetic*
- *Basic exponential time estimate:* $T_exp = exp((1/)^{1/(2n)}) for solar system$

- *Comparison to solar system age: verify* $T_exp 4.5 Gyr$

Validation:

- *Reproduce steepness for 2-planet system (Jupiter-Saturn)*
- *Match literature value* $T_exp\ 10^13 years for 8-planet system$

Deliverables:

- *Python module* `nekhoroshev.py` *with steepness checker and exponential time calculator*
- *Jupyter notebook demonstrating solar system application*
- *JSON certificate for Jupiter-Saturn system*

0.5.2 Strong Result (Months 4-8)

Extended Capabilities:

- *Fourier analysis of planetary perturbation Hamiltonian H*
- *Resonance width calculations for all |k| 10*
- *Rigorous action diffusion bounds: |I(t) - I(0)| $< ^{b}certifiedviaintervalpropagation$*

- *Numerical validation: integrate Hamilton's equations over 1 Myr, verify bound satisfied*
- *Comparison to 3+ literature sources (Nekhoroshev 1977, Niederman 2004, Guzzo+ 2005)*

Publications Benchmark:

- *Reproduce Figures 2-4 from Guzzo et al. (2005) showing action diffusion vs time*
- *Match resonance widths to within 10*

Deliverables:

- *Full `NekhoroshevCertificate` for 8-planet solar system*
- *Validation report comparing analytical bounds to numerical integration*
- *Database of resonance widths for 100+ resonances*

0.5.3 Publication-Quality Result (Months 8-9)

Novel Contributions:

- *Optimal exponent determination: maximize a subject to $T_{e}xp10Gyrconstraint$*

- *Sharpness analysis: compare $a_{o}ptimaltotheoreticallowerbounds$*
- *Extension to other planetary systems: apply to extrasolar systems (e.g., Kepler-90, TRAPPIST-1)*
- *Formal verification: translate steepness proofs to Lean or Isabelle*
- *Public database: 50+ Nekhoroshev certificates for diverse Hamiltonian systems*

Beyond Literature:

- *Improve exponents beyond Niederman (2004) for specific system classes*
- *Discover new resonances affecting long-term stability*
- *Develop automated pipeline: Hamiltonian $\rightarrow$ certificate (no human intervention)*

Deliverables:

- *Arxiv preprint: "Rigorous Nekhoroshev Stability Certificates for Planetary Systems"*
- *GitHub repository with 500+ test cases*
- *Interactive web tool: input planetary masses/orbits $\rightarrow$ get $T_{e}xpestimate$*

0.6 6. Verification Protocol

```python
def verify_nekhoroshev_results(certificate:
    NekhoroshevCertificate) -> dict:
    """
    Automated verification of Nekhoroshev certificate.

    Checks:
    1. Steepness constraint satisfied
    2. Exponents in valid range
    3. Exponential time formula correct
    4. Diffusion bound formula correct
    5. Numerical validation matches bound
    """
    results = {}

    # Check 1: Steepness
    results['steepness_valid'] = (
        certificate.is_steep and
        certificate.steepness_constant > 0
    )

    # Check 2: Exponents
    n = certificate.dimension
    a_expected = 1 / (2 * n)
    results['exponent_a_reasonable'] = (
        0.01 < certificate.exponent_a <= a_expected
    )

    results['exponent_b_reasonable'] = (
        0 < certificate.exponent_b <= certificate.exponent_a
    )

    # Check 3: Exponential time formula
    epsilon = certificate.perturbation_parameter
    a = certificate.exponent_a
    T_exp_recomputed = certificate.constant_C *
       np.exp((1/epsilon)**a)

    results['exponential_time_correct'] = (
        abs(T_exp_recomputed -
           certificate.exponential_time_normalized) /
        certificate.exponential_time_normalized < 0.01
    )

    # Check 4: Diffusion bound formula
    b = certificate.exponent_b
    diffusion_bound_recomputed = epsilon**b

    results['diffusion_bound_correct'] = (
        abs(diffusion_bound_recomputed -
           certificate.diffusion_bound) /
        certificate.diffusion_bound < 0.01
    )

    # Check 5: Numerical validation
    if certificate.numerical_validation:
        results['numerical_bound_satisfied'] = (
            certificate.max_deviation_observed <
                certificate.diffusion_bound
        )
    else:
        results['numerical_bound_satisfied'] = None  # Not tested

    # Overall verdict
    results['all_checks_passed'] = all(
        v for v in results.values() if v is not None
    )

    return results


def compare_to_literature_benchmarks(our_results: dict,
                                     source: str = 'Guzzo2005') ->
                                         dict:
    """
    Compare our Nekhoroshev results to published benchmarks.
    """
    benchmarks = {
        'Guzzo2005': {
            'system': 'Jupiter-Saturn',
            'T_exp_years': 1e13,
            'exponent_a': 0.1,
            'diffusion_bound_AU': 1e-2
        },
        'Niederman2004': {
            'exponent_a_theoretical': lambda n: 1/(2*n),
            'exponent_b_theoretical': lambda n: 1/(2*n)
        }
    }

    if source not in benchmarks:
        return {'error': f'Unknown source {source}'}

    benchmark = benchmarks[source]

    comparison = {}
    for key, value in benchmark.items():
        if key in our_results:
            our_value = our_results[key]
            relative_error = abs(our_value - value) / value
            comparison[key] = {
                'ours': our_value,
                'literature': value,
                'relative_error': relative_error,
                'match': relative_error < 0.1  # 10% tolerance
            }

    return comparison
```

Validation Procedure:

  - *Run `verifynekhoroshevresults()` on generated certificate*
  - *Compare to Guzzo et al. (2005) benchmark values*
  - *Numerical integration: evolve 2-planet system for 1 Myr, check diffusion < ^{b}*
- *Cross-check exponents with Niederman (2004) theoretical bounds*

0.7 7. Resources and Milestones

0.7.1 Essential References

- ***Original Papers**:*
- *Nekhoroshev (1977): "An exponential estimate of the time of stability of nearly-integrable Hamiltonian systems"*
- *Niederman (2004): "Stability over exponentially long times in the planetary problem"*
- *Guzzo, Lega, Froeschlé (2005): "First numerical evidence of global Arnold diffusion in quasi-integrable systems"*
- ***Textbooks**:*
- *Morbidelli (2002): Modern Celestial Mechanics*
- *Arnold, Kozlov, Neishtadt (2006): Mathematical Aspects of Classical and Celestial Mechanics*
- *Giorgilli (2003): "Exponential stability of Hamiltonian systems"*
- ***Solar System Applications**:*
- *Laskar (1989): "A numerical experiment on the chaotic behaviour of the Solar System"*
- *Murray Dermott (1999): Solar System Dynamics*

0.7.2 Common Pitfalls

- ***Steepness too restrictive**: Not all physical Hamiltonians are convex; use quasi-convex definition*
- ***Exponent optimality**: a = 1/(2n) is not always optimal; dimension-dependent improvements possible*
- ***Resonance overlap**: If resonances overlap (Chirikov criterion), Nekhoroshev theory fails*
- ***Numerical validation expensive**: Integrating N-body systems for Myr timescales requires high-precision symplectic integrators*
- ***Certificate validity**: Interval arithmetic bounds can become loose after many propagation steps*

0.7.3 Milestone Checklist

Month 1: *Symbolic Hessian computed for Kepler Hamiltonian*

Month 2: *Steepness certified via interval arithmetic for 2-planet system*

Month 3: *Fourier coefficients H computed for planetary perturbations*

Month 3: *Resonance widths estimated for |k| 10*

Month 4: *Exponential time* $T_expcomputedfor8 - planetsolarsystem$

Month 5: *Action diffusion bounds |I(t) - I(0)| <* $^bprovenrigorously$

Month 6: *Numerical validation: integrate Hamilton's equations for 1 Myr*

Month 7: *Comparison to 3+ literature sources (errors < 10*

Month 8: *Optimal exponents a, b determined via optimization*

Month 9: *Complete certificate exported to JSON, self-verification passed*

Month 9: *Public database: 10+ planetary systems analyzed*

0.7.4 Extensions

Immediate Extensions *(post-MVR):*

- *Non-convex Hamiltonians: develop quasi-convexity checkers for general systems*
- *Symplectic integrators: implement high-order methods for long-time validation*
- *Multi-scale perturbations: handle systems with disparate timescales (e.g., inner+outer planets)*

Research Frontiers:

- *Improve exponents: can a > 1/(2n) be achieved for special classes?*
- *Formal verification: translate steepness proofs to Lean/Isabelle*
- *Machine learning: train models to predict* $T_expfromHamiltonianstructure$

- *Quantum systems: extend Nekhoroshev theory to quantum Hamiltonians (FKPP theorem)*

0.8 8. Implementation Notes

0.8.1 Computational Requirements

- ***Symbolic computation***: *SymPy for Hessians, Fourier integrals (may be slow for n > 3)*
- ***Interval arithmetic***: *mpmath with 100-digit precision for certified bounds*
- ***Numerical integration***: *SciPy's* `solve`$_ivpwithDOP853forvalidation(rtol = 1e - 12)$
- ***Optimization***: *SciPy's* `minimize`$_scalarforoptimalexponentsearch$

Estimated Runtimes:

- *Steepness verification: 1 minute (symbolic), 10 minutes (interval arithmetic)*
- *Fourier coefficients: 1 hour per resonance (symbolic integration expensive)*
- *Exponential time: instant (formula evaluation)*
- *Numerical validation (1 Myr): 1 hour on single core (can parallelize)*

0.8.2 Software Dependencies

```
# requirements.txt
sympy>=1.12
numpy>=1.24
scipy>=1.11
mpmath>=1.3
matplotlib>=3.7
cvxpy>=1.4  # For SDP optimization (future extension)
```

0.8.3 Testing Strategy

- ***Unit tests:*** *Each function validated on toy Hamiltonians (harmonic oscillator, pendulum)*
- ***Integration tests:*** *Full pipeline tested on 2-planet system (Jupiter-Saturn)*
- ***Regression tests:*** *Compare to cached results from literature*
- ***Property tests:*** *Verify mathematical identities (e.g., symplectic flow preserves H)*

End of PRD 28