hexagon:2611.00392v1

Formal Verification of a Computer-Assisted Proof in Celestial Mechanics

Tejasvi Singh Tomar[0000-0003-4668-3639]

Abstract. Albouy, Fu and Sun conjectured in 2008 that four positive masses admit, for each cyclic order of the bodies, exactly one convex central configuration up to similarity; the question is Problem 10 of a 2012 list of open problems on the n-body problem. We recently proved the conjecture by reducing it to an inequality for a trace at the real zeros of an equation in four variables; a branch and bound that starts from a grid of 136,448 boxes establishes the inequality. In this paper we describe a formalization of the whole proof in Lean 4 and Mathlib in which Lean's kernel checks the computation, using `decide +kernel`, so that no compiler has to be trusted. Our checker stores an interval as two natural numbers in a biased fixed-point format. The kernel can then hand all the arithmetic to GMP. Functions are enclosed over boxes by first-order slopes. An unverified search records the decisions for each box in one natural number, and the checker replays them. We write each formula once over a type class; evaluating it at a type that pairs a real number with an interval then proves the enclosure correct. Checking the 3418 certificate theorems takes 7.6 hours of processor time, and the independent type checker nanoda accepts the proof. A version rebuilt on one eighth of the boxes passed three type checkers in 2.9 hours on a public registry of formal proofs.

Keywords: Lean 4 · Interval arithmetic · Proof by reflection · Central configurations · Computer-assisted proof

1 Introduction

A central configuration (CC) of n point masses is a configuration in which the acceleration of each body under Newtonian gravity points towards the centre of mass and is proportional to the distance from it. CCs are the shapes of the solutions of the n-body problem that rotate rigidly, and they are a basic object of celestial mechanics [26]. For four bodies, MacMillan and Bartky [23] showed that for any positive masses and any cyclic order of the bodies there is a convex CC with the bodies in that order; Xia [38] gave a shorter proof. Albouy, Fu and Sun [3] conjectured that this convex CC is unique up to similarity, and the conjecture is Problem 10 in the list of Albouy, Cabral and Santos [2]. It had been proved for four equal masses [1], for equal masses at the ends of a diagonal [22,28,3,21], and, with interval arithmetic, when each mass is at least 0.6 times the largest [32].

In [35] we proved the conjecture for all masses. We bounded the Hessian of the potential at a convex CC from below by a quarter of a form that vanishes

only on rotations and translations (Theorem B in Sect. 2); a covering argument then turns this nondegeneracy into uniqueness (Theorem A). An exact identity for the Hessian and the Cauchy–Schwarz inequality reduce Theorem B to an inequality, $\operatorname{tr} S(y) < 3/4$ for some $y \in \mathbb{R}^2$, that has to hold at each real zero of an explicit function P of four variables in a bounded region. We proved this inequality by a branch and bound in Python that uses Arb ball arithmetic [19]. The search starts from a grid of 136,448 boxes and takes about four minutes to run on 32 hardware threads.

A proof of this kind asks the reader to trust the program and the libraries below it, and to accept that the formulas in the code are those in the paper. To remove these assumptions, we formalized the whole proof in Lean 4 [10] and Mathlib [34]. The mathematical part is a conventional Mathlib development. For the computation we follow the reflection approach [6,5]: we define a checker as a function in the logic, prove it sound once, and let the kernel run it on each certificate. Lean offers two ways to run such a checker. The tactic `native_decide` compiles it and executes machine code, but the proof then depends on the axiom `Lean.ofReduceBool`, through which the compiler and the run-time system have to be trusted. The tactic `decide +kernel` instead has the kernel evaluate the Boolean term, and it adds no axiom. We use only `decide +kernel`. The kernel evaluates terms by rewriting. However, it hands fourteen operations on natural-number literals to GMP, among them addition, multiplication, division, comparisons, bitwise operations and shifts, and we wrote the checker so that all of its arithmetic is done by these operations.

In this paper we report what it took to make a nonlinear computer-assisted proof of this size checkable by Lean's kernel, and what the checks cost. Our checker stores an interval as two natural numbers that are both upper bounds, in a biased fixed-point format, so that each step of the interval arithmetic is an accelerated operation. We write the formulas of the proof once over a type class and evaluate them at the reals in the proofs and at intervals and slope jets in the checker. A carrier type that pairs a real number with its enclosure turns one evaluation into the soundness proof of a formula. The search is done by an unverified program that writes its decisions into a natural number; the checker replays them. We also describe the problems we met in keeping the proofs about the checker cheap for the kernel and for a second type checker, and how we cut the certificates so that the memory needed to check them stays bounded. On 32 hardware threads the certificate modules build in 19 minutes, or 7.6 hours of processor time, and the independent type checker nanoda also accepts the proof.

Section 2 summarizes the mathematics and Sect. 3 the formal statement. Section 4 describes the checker and Sect. 5 the checks, including a version that fits the limits of a public registry. Section 6 discusses related work, and Sect. 7 the limits of the design.

2 The Theorem and Its Computer-Assisted Proof

2.1 Central Configurations

Let $m_1, \dots, m_4 > 0$ be masses at distinct points $q_1, \dots, q_4 \in \mathbb{R}^2$. We write $r_{ij} = |q_i - q_j|$, $U = \sum_{i<j} m_i m_j / r_{ij}$, $M = \sum_i m_i$, $c = M^{-1} \sum_i m_i q_i$ and $I = \sum_i m_i |q_i - c|^2$. The configuration q is a CC if there is $\lambda \in \mathbb{R}$ with

$$\nabla_{q_i} U + \lambda m_i (q_i - c) = 0 \qquad (i = 1, \dots, 4); \tag{1}$$

then $\lambda = U/I > 0$. The CCs are the critical points of $UI^{1/2}$, a function invariant under similarities. At a CC, we fix λ and define two quadratic forms on $v \in (\mathbb{R}^2)^4$,

$$Q(v) = D^2\big(U + \tfrac{\lambda}{2} I\big)(q)[v, v], \qquad K(v) = \sum_{i<j} \frac{3 m_i m_j}{r_{ij}^3} \dot{r}_{ij}^2,$$

where $\dot{r}_{ij} = \langle q_i - q_j, v_i - v_j \rangle / r_{ij}$ is the derivative of r_{ij} in the direction v. At a convex configuration, $K(v) = 0$ exactly when v is an infinitesimal rotation or translation.

Theorem B ([35]). *At every convex CC of positive masses, $Q \geq K/4$.*

Theorem A ([35]). *For all positive masses $m_1, \dots, m_4$ there is, up to similarity, exactly one convex CC with the bodies in the cyclic order 1234. It is a nondegenerate local minimum of $UI^{1/2}$ on the space of shapes, and it depends analytically on the masses.*

Theorem A follows from Theorem B by a covering argument. We fix the similarity by placing q_1 at $(0, 0)$ and q_2 at $(1, 0)$, fix the orientation by the signs of the triangle areas, and let Z be the set of pairs (m, q) of positive masses and a convex CC q for m in this slice, with the cyclic order 1234. By Theorem B and the implicit function theorem, the projection $Z \to (0, \infty)^4$ is a local homeomorphism. It is also proper: a limit of such CCs with masses in a compact set has no collision, by a lemma of Shub [29] applied with varying masses, and it is not collinear, by Theorem B again. Since $(0, \infty)^4$ is connected, all fibres then have the same number of points, and the fibre over four equal masses is the square alone [1], [3, Theorem 1]. The argument needs only $Q \geq \varepsilon K$ for some $\varepsilon > 0$; Sect. 7 returns to this.

2.2 From the Hessian to a Bounded Computation

At a convex CC, Dziobek's equations [11] give $m_i m_j (r_{ij}^{-3} - \lambda') = \sigma A_i A_j$ for all $i < j$, where $\lambda' = \lambda / M$, A_l is the signed area of the triangle formed by the three bodies other than l, and $\sigma < 0$. From them we obtain the identity

$$Q = K - |\sigma|\, |L|^2, \qquad L = \sum_l A_l v_l \in \mathbb{R}^2. \tag{2}$$

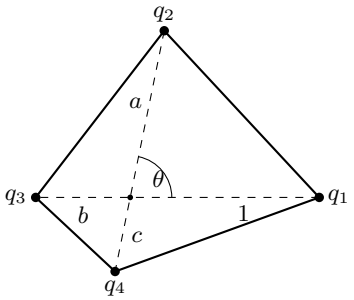


Fig. 1. The coordinates of [9], drawn at $a = 0.9$, $b = 0.5$, $c = 0.4$ and $x = \cos\theta = 0.2$, a point where all six g_i are positive. The diagonals meet at the origin.

The six rates $\dot r_e$, $e = ij$, satisfy one linear relation $\sum_e \tau_e \dot r_e = 0$, with $\tau_{ij} = A_i A_j r_{ij}$, and hence $L = \sum_e (\beta_e + \tau_e y)\dot r_e$ for all $y \in \mathbb{R}^2$, with vectors $\beta_e \in \mathbb{R}^2$ that depend only on the shape. The Cauchy–Schwarz inequality with the weights of K gives

$$|\sigma|\,|L|^2 \le \lambda_{\max}(S(y))\,K \le \operatorname{tr} S(y)\,K, \tag{3}$$

where $S(y) = \sum_e D_e(\beta_e + \tau_e y)(\beta_e + \tau_e y)^{\mathsf T}$ and $D_{ij} = |\sigma| r_{ij}^3/(3m_i m_j) = -(r_{ij}^{-3} - \lambda')\, r_{ij}^3/(3A_iA_j)$. On convex CCs, $\lambda' = (s_{12}s_{34} - s_{13}s_{24})/(s_{12} + s_{34} - s_{13} - s_{24})$ with $s_e = r_e^{-3}$. Thus $S(y)$ does not depend on the masses. By (2) and (3), it is therefore enough to show that each convex CC has a y with $\operatorname{tr} S(y) < 3/4$. The minimum of $\operatorname{tr} S(y)$ over y has supremum about 0.593 over convex CCs, and the constant $3/4$ leaves a margin of about 0.157.

2.3 Coordinates and Regions

We use the coordinates of Corbera, Cors and Roberts [9]. We label the bodies so that q_1q_2 is a longest side. At a convex CC the opposite side q_3q_4 is then a shortest side, and both diagonals are longer than all four sides. A similarity brings the configuration to the form

$$q_1 = (1,0), \quad q_2 = a(x,s), \quad q_3 = (-b,0), \quad q_4 = -c(x,s), \qquad s = \sqrt{1-x^2},$$

with the diagonals meeting at the origin (Fig. 1). The squared distances $R_e = r_e^2$ are polynomials, such as $R_{12} = 1 + a^2 - 2ax$ and $R_{24} = (a+c)^2$, and the order of the six distances becomes six inequalities $g_i \ge 0$, each affine in x; for instance $g_1 = R_{13} - R_{12} = b^2 + 2b - a^2 + 2ax$. If we eliminate λ' from Dziobek's equations, one equation remains,

$$P = \eta_{14}\eta_{23}(\eta_{13} + \eta_{24} + \eta_{34}) + \eta_{13}\eta_{24}(\eta_{14} + \eta_{23} - \eta_{34}) = 0, \tag{4}$$

where $\eta_e = |s_e - s_{12}|$. Our checker computes η_e as $R_e - R_{12}$ or $R_{12} - R_e$, whichever is nonnegative, written as a combination of the g_i with nonnegative coefficients,

times $\psi(R_e, R_{12})$ with $\psi(X,Y) = (X + \sqrt{XY} + Y)/((\sqrt{X} + \sqrt{Y})(XY)^{3/2})$, a form without cancellation. The points with $g \geq 0$ and $P = 0$ are exactly the convex CCs, up to relabelling and similarity [9, Lemma 3.1].

A priori bounds [9, Lemma 3.2] put every convex CC with $b \geq 1/8$ in the box $\Omega_{\mathrm{dir}} = [\frac{1}{2}, \frac{7}{4}] \times [\frac{1}{8}, 1] \times [0, \frac{7}{4}]$ of the coordinates (a, b, c). As $b \to 0$, the bodies 3 and 4 collide at the origin, which forms an equilateral triangle with q_1 and q_2, and m_3 and m_4 tend to 0 like b^3. For $b \leq 1/8$ we use the chart $a = 1 + b\alpha$, $c = b\gamma$, $x = \frac{1}{2} + b\xi$, in which the bounds give $(b, \alpha, \gamma) \in \Omega_{\mathrm{ch}} = [0, \frac{1}{8}] \times [-2, 1] \times [0, 3]$, and in which $\hat{P} = bP$ extends to $b = 0$ with $\partial\hat{P}/\partial\xi < 0$ there. The computation has to establish two claims:

(C1) if $(a, b, c) \in \Omega_{\mathrm{dir}}$, $-1 \leq x \leq 1$, $g_i(a, b, c, x) \geq 0$ for all i and $P(a, b, c, x) = 0$, then $\operatorname{tr} S(y) < 3/4$ for some $y \in \mathbb{R}^2$;
(C2) the same holds in the chart, for $(b, \alpha, \gamma) \in \Omega_{\mathrm{ch}}$, with $\hat{P}$ in place of P.

2.4 Why the Computation Is Awkward to Formalize

The claims are not inequalities over boxes. They quantify over the zeros of P in x and ask for a witness y at each zero. The checker therefore has to localize the zeros in each box, and we need a y that works on a whole piece; we find it in floating point and only have to verify it in the checker. The formulas also divide and take square roots. Division and square root in Mathlib are total, with $x/0 = 0$ and $\sqrt{x} = 0$ for $x < 0$, so we have to relate the checker's enclosures to total functions. Finally, the volume is large for a kernel that evaluates by rewriting: the Python search evaluated P 17.8 million times.

3 The Formal Statement

In Listing 1.1 we show the definitions that a reader has to check and the statements of Theorems A and B. Positions are in `V2` $= \mathbb{R} \times \mathbb{R}$ and configurations are maps from `Fin 4` to `V2`, so the bodies $1, \ldots, 4$ of Sect. 2 are `0`, ..., `3` here. In `IsCC`, `ss q i j` is r_{ij}^{-3} and `cm m q` is the centre of mass. `esum` sums over the six pairs, `dr` is $\dot{r}_{ij}$, and `wgeo` is $r_{ij}^{-3} - \lambda/M$ with $\lambda = U/I$. The form `hessQ` is an explicit formula, and `hessian_Phi` proves that it is the second Fréchet derivative of $U + \frac{\lambda}{2} I$, so the formula does not have to be trusted. `Order1234` is stated with signed areas, and `order1234_iff` proves it equivalent to the crossing of the diagonals q_1q_3 and q_2q_4. `Similar` allows reflections. We also prove uniqueness for every cyclic order, the analytic dependence on the masses, and the nondegeneracy as a local minimum on the shape space, which we construct as a compact real-analytic manifold of dimension 4.

For the main theorems, `#print axioms` lists `propext`, `Classical.choice` and `Quot.sound`; for the two theorems that collect the certificates (Sect. 4.7) it lists `propext` alone. No source file uses `sorry`, `admit`, `axiom` declarations, `native_decide`, `implemented_by`, `extern`, `unsafe` or `partial`.

The Mathlib part has 39 files with 12,790 lines, and 1,900 of these lines prove the checker sound. The checker `C4Check` has 941 lines in five files: `Num` (162 lines)

Listing 1.1. The definitions behind the main theorems and their statements (from `C4/Defs.lean`, `ShapeHess.lean`, `TheoremB.lean` and `TheoremA.lean`). The variables `m : Masses` and `q : Conf` are section variables.

```
def IsCC : Prop :=
  CollisionFree q ∧
    ∃ lam : ℝ, ∀ i, (∑ j, (m i * m j * ss q i j) • (q j - q i)) +
      (lam * m i) • (q i - cm m q) = 0
noncomputable def hessK (v : Conf) : ℝ :=
  esum fun i j => 3 * m i * m j * ss q i j * dr q v i j ^ 2
noncomputable def hessQ (v : Conf) : ℝ :=
  hessK m q v -
    esum fun i j => m i * m j * wgeo m q i j * dot (v i - v j) (v i - v j)
def Order1234 (q : Conf) : Prop :=
  0 < area q 0 * area q 2 ∧ 0 < area q 1 * area q 3 ∧ area q 0 * area q 1 < 0
def Similar (q q' : Conf) : Prop :=
  ∃ a b : ℝ, ∃ t : V2, a ^ 2 + b ^ 2 ≠ 0 ∧
    (q' = simc a b t q ∨ q' = simc a b t (mirror q))
theorem hessian_Phi {m : Masses} (hm : ∀ i, 0 < m i) {q : Conf}
    (hq : CollisionFree q) (v : Conf) :
    fderiv ℝ (fderiv ℝ (fun p : Conf => Upot m p + lamC m q / 2 * Iner m p))
        q v v = hessQ m q v
theorem theoremB (m : Masses) (hm : ∀ i, 0 < m i) (q : Conf) (hcc : IsCC m q)
    (hconv : IsConvex q) (v : Conf) : hessK m q v / 4 ≤ hessQ m q v
theorem theoremA (m : Masses) (hm : ∀ i, 0 < m i) :
    ∃ q, IsCC m q ∧ Order1234 q ∧ ∀ q', IsCC m q' → Order1234 q' → Similar q q'
```

for numbers, `Jet` (158) for jets and the class `Ops`, `Formulas` (300), `Search` (212) and `Grid` (109). We kept it free of Mathlib: it uses core Lean only, and the certificate modules import `C4Check` alone, so checking them never loads Mathlib.

4 The Checker

4.1 Evaluation by the Kernel

The tactic `decide +kernel` proves a proposition from its `Decidable` instance by having the kernel reduce the instance to `isTrue`. Each of our certificates states `f h = true` for a Boolean function `f` and a literal `h`. To check it, the kernel evaluates `f h` by weak-head reduction: it unfolds definitions, reduces matches and recursors, and calls GMP when one of `Nat.add`, `sub`, `mul`, `div`, `mod`, `gcd`, `beq`, `ble`, `land`, `lor`, `xor`, `shiftLeft`, `shiftRight` and `pow` is applied to two literals. It has no other arithmetic. In particular, `Int` has no accelerated operations of its own, and a `Nat` operation whose arguments do not reduce to literals, as happens in a proof with free variables, unfolds to the unary definition of `Nat`. Both facts shaped the design that we describe in the rest of this section.

4.2 Numbers

Since `Int` is not accelerated, we represent a real number by a natural number X with value $(X - 2^{256})/2^{96}$. Values thus start at -2^{160} and have absolute precision 2^{-96}, and the bias keeps all quantities natural numbers. An interval

Listing 1.2. Fixed-point intervals (from `C4Check/Num.lean`, abridged).

```
def Bb : Nat := 2 ^ 256
def C1 : Nat := 2 ^ 512 + 2 ^ 352      -- Bb^2 + Bb 2^96
def RND : Nat := 2 ^ 96 - 1
macro "sel " b:term:max t:term:max e:term:max : term =>
  `(Bool.casesOn (motive := fun _ => _) $b $e $t)
structure I where
  L : Nat
  H : Nat
/-- `(X - Bb) (Y - Bb) + Bb 2^96`, truncated at `0` -/
def pp (X Y : Nat) : Nat :=
  Nat.sub (Nat.add (Nat.mul X Y) C1) (Nat.shiftLeft (Nat.add X Y) 256)
def up (P : Nat) : Nat := Nat.shiftRight (Nat.add P RND) 96
def add (A B : I) : I :=
  ⟨Nat.sub (Nat.add A.L B.L) Bb, Nat.sub (Nat.add A.H B.H) Bb⟩
def neg (A : I) : I := ⟨A.H, A.L⟩
/-- the product, by the signs of the factors (`L ≤ Bb` means that every element
is `≥ 0`, `H ≤ Bb` that every element is `≤ 0`) -/
def mul (A B : I) : I :=
  sel (Nat.ble A.L Bb)
    (sel (Nat.ble B.L Bb) ⟨up (pm A.L B.L), up (pp A.H B.H)⟩
      (sel (Nat.ble B.H Bb) ⟨up (pp A.H B.L), up (pm A.L B.H)⟩
        ⟨up (pp A.H B.L), up (pp A.H B.H)⟩))
    (sel (Nat.ble A.H Bb) ... ...)     -- six more cases
def isqD (n : Nat) : Nat := let r := isq n; sel (Nat.ble (Nat.mul r r) n) r 0
def isqU (n : Nat) : Nat :=
  let s := Nat.succ (isq n); sel (Nat.ble n (Nat.mul s s)) s n
def sqrt (A : I) : Option I :=
  sel (Nat.ble A.L Bb)
    (some ⟨Nat.sub Bb (isqD (Nat.shiftLeft (Nat.sub Bb A.L) 96)),
      Nat.add Bb (isqU (Nat.shiftLeft (Nat.sub A.H Bb) 96))⟩)
    none
```

$\langle L, H\rangle$ contains v if $-v \le \mathrm{val}(L)$ and $v \le \mathrm{val}(H)$ (Listing 1.2). Since both components are upper bounds, of $-v$ and of v, one rounding direction is enough: we always round up. The truncated subtraction of `Nat` can only increase a result, so it is in fact safe wherever it occurs. We do not need to maintain an invariant such as $\mathrm{val}(L) + \mathrm{val}(H) \ge 0$, because an interval whose bounds cross contains no number, and statements about its elements then hold vacuously.

We form products directly on the codes. For $X = 2^{256} + x\,2^{96}$ and $Y = 2^{256} + y\,2^{96}$,

$$XY + 2^{512} + 2^{352} - 2^{256}(X + Y) = 2^{96}\big(2^{256} + xy\,2^{96}\big),$$

and therefore `pp X Y` followed by `up`, a shift by 96 bits that rounds up, gives the code of xy. The function `mul` chooses among nine cases by the signs of the factors [27], and the tests are written with `sel`, a macro for a bare `Bool.casesOn`. To compute a reciprocal, we divide 2^{192} by the distance of a code from 2^{256} with `Nat.div`, rounding down or up as the bound requires. Integer square roots are computed by Newton's iteration from above, starting at a power of two that is found by bisection on the exponent. The result is checked by squaring it once. If this check fails, the checker falls back to a bound that is valid but useless (0 below, the radicand above). Division and the square root return an `Option I`, which is `none` when the divisor may vanish or the radicand may be negative.

Listing 1.3. The class `Ops`, a generic formula, and the carrier `EI` (from `C4Check/Jet.lean`, `C4Check/Formulas.lean`, `C4/Carrier.lean` and `C4/ModeSound.lean`, abridged). In `Formulas.lean`, α has an instance `[Ops α]`.

```
class Ops (α : Type) extends Add α, Sub α, Mul α, Div α, Neg α where
  sqrt : α → α
  sq : α → α
  ofInt : Int → α
instance : Ops (Option I) where
  div x y := match x, y with | some a, some b => a.div b | _, _ => none
  ...
/-- `hd X Y = (Y^(-3/2) - X^(-3/2)) / (X - Y)`, in a form without cancellation -/
def hd (X Y : α) : α :=
  let sx := Ops.sqrt X
  let sy := Ops.sqrt Y
  (X + sx * sy + Y) / ((sx + sy) * X * sx * Y * sy)
def dirP (a b c x : α) : α :=
  let g := dirG a b c x
  let R12 := 1 + a * a - 2 * a * x
  ...
  E14 * E23 * (D113 + E34 + D124) + D113 * D124 * (E14 + E23 - E34)
def OEnc (v : ℝ) (o : Option I) : Prop := ∀ A, o = some A → I.mem v A
structure EI where
  r : ℝ
  i : Option I
  ok : OEnc r i
@[instance_reducible] def eiOps : Ops EI where
  mul x y := ⟨x.r * y.r, x.i * y.i, by
    rcases x with ⟨rx, ix, hx⟩; rcases y with ⟨ry, iy, hy⟩
    intro A h
    cases ix <;> cases iy <;> cases h
    exact I.mem_mul (hx _ rfl) (hy _ rfl)⟩
  ...
theorem dirP_enc :
    OJEnc S zc (fun z => dirPR (z 0) (z 1) (z 2) (z 3))
      (@dirP _ (jetOps R) o0 o1 o2 o3) :=
  (@dirP (EJ S zc) (ejOps R hR)
    ⟨_, o0, h0⟩ ⟨_, o1, h1⟩ ⟨_, o2, h2⟩ ⟨_, o3, h3⟩).ok
```

4.3 Formulas Written Once

We write the formulas of Sect. 2, that is, the g_i, P, the coefficients D_e, τ_e, β_e, and $\operatorname{tr} S(y)$, only once, in `Formulas.lean`, for an arbitrary type with an instance of the class `Ops` (Listing 1.3). Integer literals are turned into `Ops.ofInt` by a low-priority `OfNat` instance. The same definitions can then be evaluated at four types: at $\mathbb{R}$ with the instance `opsReal` in the proofs, at `Option I` and at `Option J`, the jets of Sect. 4.4, in the checker, and at `Float` in numerical tests. In `Option I`, an undefined operation is recorded as `none`. We chose not to make `opsReal` a global instance, because with it in scope ordinary real arithmetic would be elaborated through `Ops`. The Mathlib modules `Chart` and `NormalForm` prove that the real evaluations equal the geometric quantities of Sect. 2. As a result, no formula is transcribed by hand between the mathematics and the checker.

We obtain the soundness of the checker's evaluations from carrier types. An element of `EI` consists of a real number, an `Option I`, and a proof that the option, when it is `some A`, contains the number. The instance `eiOps` performs each operation on both components and proves the enclosure with a lemma of

Listing 1.4. Slope jets (from `C4Check/Jet.lean`, abridged).

```
structure J where
  U : I
  C : I
  s0 : I
  s1 : I
  s2 : I
  s3 : I
/-- centred form `C + Σ_i s_i [-r_i, r_i]` -/
def cf (R : Rad) (C s0 s1 s2 s3 : I) : I :=
  let E := Nat.add (Nat.add (I.eterm s0 R.r0) (I.eterm s1 R.r1))
    (Nat.add (I.eterm s2 R.r2) (I.eterm s3 R.r3))
  ⟨Nat.add C.L E, Nat.add C.H E⟩
def range (R : Rad) (u : J) : I := u.U.inter (cf R u.C u.s0 u.s1 u.s2 u.s3)
/-- `u v - ū v̄ = u (v - v̄) + v̄ (u - ū)` -/
def mul (R : Rad) (u v : J) : J :=
  let s0 := (u.U.mul v.s0).add (v.C.mul u.s0)
  ...
  let s3 := (u.U.mul v.s3).add (v.C.mul u.s3)
  let C := u.C.mul v.C
  ⟨(u.U.mul v.U).inter (cf R C s0 s1 s2 s3), C, s0, s1, s2, s3⟩
```

`NumSound`; Listing 1.3 shows multiplication. If we evaluate a generic formula at `EI`, the first component of the result is the real formula and the second is the checker's computation, both by definition. The third component is then the enclosure theorem for that formula. The carrier `EJ` does the same for jets. This is why the enclosure of P over a box has a proof of one line (`dirP_enc`), and this proof does not change when the formula does. The real side needs no side conditions: division and square root on $\mathbb{R}$ are total, and the enclosure is claimed only when the checker returns `some`. For a quotient, this happens only if the divisor's interval excludes 0.

The usual alternative, used for instance in CoqInterval [25,24], is to reify expressions into a syntax tree and to prove an evaluator of the tree correct by induction. Carriers need neither a syntax nor reification. The price is that every formula has to be written against `Ops`, also on the Mathlib side, where we have to supply `opsReal` explicitly.

4.4 Slope Jets

Plain interval evaluation of P over a box of side $1/32$ overestimates its range badly, because each variable occurs many times in P. To reduce the overestimation, we use first-order centred forms with slopes [20]. A jet over a box $Z \subset \mathbb{R}^4$ with centre $\bar{z}$ consists of intervals U, C and $S_0, \ldots, S_3$; the coordinates of Z are the three coordinates of the cell and x. The jet encloses a function u if $u(z) \in U$ for $z \in Z$, $u(\bar{z}) \in C$, and for each $z \in Z$ there are $s_i \in S_i$ with $u(z) - u(\bar{z}) = \sum_i s_i(z_i - \bar{z}_i)$. The rules for the operations are algebraic identities, such as $uv - \bar{u}\bar{v} = u(v-\bar{v}) + \bar{v}(u-\bar{u})$ (Listing 1.4) and $\sqrt{u} - \sqrt{\bar{u}} = (u-\bar{u})/(\sqrt{u}+\sqrt{\bar{u}})$. As a consequence, their soundness proofs in `JetSound` need no mean value theorem. Each operation intersects the plain interval result with the centred form $C + \sum_i S_i[-r_i, r_i]$, where r_i bounds $|z_i - \bar{z}_i|$. Slopes also give us the interval

Listing 1.5. Replaying a hint (from `C4Check/Search.lean` and `C4/SearchSound.lean`, abridged).

```
def locF (M : Mode) (B : Box)
    (rec : Nat → List I → List I → Option (List I × Nat)) (h : Nat)
    (X : I) (work kept : List I) : Option (List I × Nat) :=
  sel (Nat.ble (Nat.land h 3) 1)
    (sel (Nat.beq (Nat.land h 3) 0)
      ((pstep M B X).bind fun X' =>
        sel (I.isEmpty X') (rec (Nat.shiftRight h 2) work kept) none)
      (rec (Nat.shiftRight h 2) work (X :: kept)))
    (sel (Nat.beq (Nat.land h 3) 2)
      ((pstep M B X).bind fun X' => rec (Nat.shiftRight h 2) (X' :: work) kept)
      (rec (Nat.shiftRight h 2)
        ((I.halves X).1 :: (I.halves X).2 :: work) kept))
def locH (M : Mode) (B : Box) :
    Nat → Nat → List I → List I → Option (List I × Nat)
  | _, h, [], kept => some (kept, h)
  | 0, _, _ :: _, _ => none
  | fuel + 1, h, X :: work, kept => locF M B (locH M B fuel) h X work kept
def leafH (M : Mode) (B : Box) (h : Nat) : Option Nat :=
  (xrange M B).bind fun X =>
    sel (I.isEmpty X) (some h)
      ((locH M B 4096 h [X] []).bind fun r => certAllH M B r.1 r.2)
def checkBoxH (M : Mode) : Nat → Box → Nat → Option Nat
  | 0, _, _ => none
  | fuel + 1, B, h =>
    sel (Nat.beq (Nat.land h 3) 0) (leafH M B (Nat.shiftRight h 2))
      ((checkBoxH M fuel (splitBox B (Nat.land h 3)).1 (Nat.shiftRight h 2)).bind
        fun h' => checkBoxH M fuel (splitBox B (Nat.land h 3)).2 h')
theorem sel_pos {α : Sort _} {b : Bool} (t e : α) (h : b = true) :
    Bool.casesOn (motive := fun _ => α) b e t = t := by
  subst h; rfl
def AllGood (M : Mode) (Sm : Sem) (B : Box) : Prop :=
  ∀ z0 z1 z2 x, B.mem z0 z1 z2 → (M.clamp = true → -1 ≤ x ∧ x ≤ 1) →
    (∀ i, 0 ≤ (Sm.G z0 z1 z2 x).get i) → Sm.P z0 z1 z2 x = 0 →
    ∃ y0 y1, trSR (Sm.C z0 z1 z2 x) y0 y1 < 3 / 4
theorem checkBoxH_sound (hM : ModeSound M Sm) : ∀ (fuel : Nat) (B : Box)
    (h h' : Nat), checkBoxH M fuel B h = some h' → AllGood M Sm B
```

Newton step in x: at a zero, $0 = P(\bar{z}) + \sum_{i<3} s_i(z_i - \bar{z}_i) + s_3(x - \bar{x})$, and this equation can be used to confine x when S_3 excludes 0.

4.5 Hints

The search is not part of the checker. An unverified program, `Gen.lean`, decides where to split a box, how to localize the zeros of P in x, and which y to use. We run `Gen` beforehand. It searches in the arithmetic of the checker and records the decisions in a *hint*: a natural number that the checker reads two bits at a time, starting from the least significant end (Listing 1.5). At a box, the symbol 0 makes the box a leaf, and the symbols 1, 2, 3 split it in half along coordinate 0, 1, 2. At a leaf, the checker first computes the interval of x allowed by the conditions $g_i \geq 0$ (they are affine in x). Unless this interval is empty, the hint continues with one symbol per step of the localization (0 drop, 1 keep, 2 Newton step, 3 bisect). Then, for each kept interval, it gives two 32-bit fields that place y on a grid of step 2^{-24} in $[-128, 128)^2$. The function `checkBoxH` replays

the hint and checks each decision. An interval can be dropped only if one step of the localization empties it, i.e., if the enclosure of some g_i is negative, the enclosure of P excludes 0, or the Newton step leaves nothing. A kept interval has to satisfy $\operatorname{tr} S(y) < 3/4$ in jet arithmetic. A wrong hint can of course make the check fail, but it cannot make it succeed wrongly: the soundness theorem `checkBoxH_sound` holds for all hints. The recursions carry explicit fuel (depth 100 for boxes, 4096 localization steps per leaf). This makes them structural, so that they can be unfolded by the kernel.

`Gen` replays each hint before writing it. Its search differs from the Python one in several respects, and on the direct region it splits 3,800 times, against 318,688 times for Python (Table 1). We do not know which of the differences actually accounts for this.

4.6 Writing for the Kernel

The same kernel also checks the soundness proofs, and in these proofs the checker's functions are applied to free variables. This caused us several problems. When the kernel meets a `Bool.casesOn` or a `match` at the head of a term, it reduces the condition or the discriminant to weak head normal form. If the condition compares an interval computed from variables, the arithmetic unfolds to unary numbers, and that is infeasible for numbers of this size. We dealt with this and with related problems in four ways.

Tests take their interval as an argument. We put each test on a computed interval into a small definition that takes the interval as a parameter (`negH`, `ifExcl`, `newtonStep`, `trOK`, `xtight`), and every other `match` is on an argument or goes through `Option.bind`. In `SearchSound` we decide each test by rewriting with `sel_pos` or `sel_neg` (Listing 1.5) from a hypothesis on the condition, so that the kernel never has to evaluate a condition.

Equation lemmas. With the step of the localization loop inlined into `locH`, Lean could not prove the equation lemmas of `locH`. We therefore moved the step into a separate definition, `locF`, and `locH` now recurses structurally on its fuel and calls `locF`.

Large constants in the elaborator. Unlike the kernel, the elaborator refuses to evaluate powers whose exponent exceeds the option `exponentiation.threshold` (256 by default). When it elaborates the enclosure theorems for the chart formulas, it meets the constants 2^{352} and 2^{512} of `I.mul` through the literal $1/2$. We raised the threshold to 512 in two theorems of `ModeSound`, and to 1024 in `NumSound`.

A second type checker. nanoda (Sect. 5.2) at first ran out of memory on our proofs, after using more than 80 GB. To locate the cause, we checked the export in windows of consecutive declarations, with the theorems before each window turned into axioms. Two test theorems that differed only in the name of one bound variable then showed the problem: nanoda accepted one and ran out of

Listing 1.6. Cells, a certificate, and the use of the certificates (from `C4Check/Grid.lean`, `C4Cert/Dir000.lean`, `C4Cert/Cover.lean` and `C4/Cap.lean`; hints shortened).

```
def iv32 (n : Nat) : I :=
  ⟨Nat.sub Bb (Nat.mul n (2 ^ 91)), Nat.add Bb (Nat.mul (Nat.succ n) (2 ^ 91))⟩
def dirBox (i j k : Nat) : Box := ⟨iv32 (16 + i), iv32 (4 + j), iv32 k⟩
def dirCell (n h : Nat) : Bool :=
  (checkBoxH dirMode depth (dirBox (n / 1568) (n / 56 % 28) (n % 56)) h).isSome
def allCells (P : Nat → Nat → Bool) : Nat → Nat → List Nat → Bool
  | n, e, [] => Nat.ble e n
  | n, e, h :: hs => sel (P n h) (allCells P (Nat.succ n) e hs) false
def Cover (P : Nat → Nat → Bool) (a b : Nat) : Prop :=
  ∀ n, a ≤ n → n < b → ∃ h, P n h = true
theorem c17 : allCells dirCell 4889 4944 [
    1028745948254294395337840738981584510784295054462 9...407766484658,
    1836186274882356288483575474687011094829909070287 8...800100330722,
    3886408183679609095334855245725328381380393398658 6...810783856434,
    147494596195641123668, 0, 0, 0, ...] = true := by
  decide +kernel
theorem dirCover : Cover dirCell 0 62720 := Cert.Dir000.cover.trans <| ...
theorem dir_cap_of (hC : Cover dirCell 0 62720) {a b c x : ℝ}
    (ha1 : 1 / 2 ≤ a) (ha2 : a ≤ 7 / 4) (hb1 : 1 / 8 ≤ b) (hb2 : b ≤ 1)
    (hc1 : 0 ≤ c) (hc2 : c ≤ 7 / 4) (hx1 : -1 ≤ x) (hx2 : x ≤ 1)
    (hG : ∀ i, 0 ≤ (dirGR a b c x).get i) (hP : dirPR a b c x = 0) :
    ∃ y0 y1, trSR (dirCertR a b c x) y0 y1 < 3 / 4
```

memory on the other. It does not identify terms that differ only in the names of bound variables until it has unfolded them, and on the tests of intervals computed from free variables this unfolding again reaches unary arithmetic. The lemmas `negH_of_ble`, `ifExcl_pos` and `ifExcl_neg` now state these tests once for a variable interval, and we changed the proofs to use them.

4.7 Certificates

We cut the regions into grids of cells: $40 \times 28 \times 56$ cells of side $1/32$ in Ω_{dir} and $8 \times 96 \times 96$ cells of side $1/64 \times 1/32 \times 1/32$ in Ω_{ch}, 136,448 in all. `Grid.lean` numbers the cells and defines `dirCell n h` to run `checkBoxH` on cell `n` with hint `h` (Listing 1.6). A certificate theorem states that `allCells dirCell n e` $[h_n, \ldots, h_{e-1}]$ is `true`, i.e., that each cell from n to $e-1$ passes with the hint listed for it. Such a theorem is proved by `decide +kernel`. Most hints are 0, for a cell with an empty range of x or one that can be dropped in one step; the longest hint has 11,114 bits. The lemma `allCells_cover` turns a certificate into a statement `Cover dirCell n e`. The modules chain these statements with `Cover.trans`, and `Cover.lean` proves `dirCover : Cover dirCell 0 62720` and `chCover : Cover chCell 0 73728`. From these two theorems, `checkBoxH_sound` and the carriers, `Cap.lean` then proves (C1) and (C2) for the real functions (`dir_cap_of` in Listing 1.6).

We can predict the kernel time of a certificate from the replay. Our script `mkcert.py` estimates the time of a cell as $1.15\,(0.07\,n_P + 0.21\,n_T) + 0.005$ seconds, where n_P and n_T count the jet evaluations of P and of $\operatorname{tr} S(y)$ in the replay of the cell, that is, about 80 ms and 240 ms per evaluation. Since the memory of a kernel check grows with the length of the theorem, the cells are cut into theorems

Table 1. The Python search of [35] and the generator `Gen`. Python counts are evaluations made during the search, on 32 processes. `Gen` counts are the jet evaluations in the replay of its final hints, which is the work the kernel repeats.

	Python		`Gen`	
	direct	chart	direct	chart
Initial cells	62,720	73,728	62,720	73,728
Splits	318,688	34	3,800	0
Leaves	381,408	73,762	66,520	73,728
Evaluations of P	17,422,672	379,455	36,673	1,749
Evaluations of $\operatorname{tr} S(y)$	645,551	18,608	49,527	18,523
Wall-clock time	245 s	10 s	10 min	6 min

of about 6 s of predicted time. The theorems are then grouped into 211 modules: `Dir000` to `Dir158` with 2573 theorems and `Ch000` to `Ch051` with 845. Even so, a module can need up to about 5 GB while the kernel checks its longest theorem, and the build script therefore checks the 52 modules with the longest theorems first, in batches. The model is optimistic: it predicted 20,209 s of kernel time for all modules, and they took 27,483 s, 36% more.

5 Results

5.1 Search and Kernel Check

In Table 1 we compare the Python search with `Gen`. The kernel in fact replays only the final decisions of `Gen`. Over both regions, they amount to 38,422 jet evaluations of P and 68,050 of $\operatorname{tr} S(y)$. The Python search, in contrast, evaluated P 17.8 million times. A jet evaluation of P costs about 80 ms in the kernel.

Our machine has 32 hardware threads and 128 GB of memory. On it, the 211 certificate modules took 19 minutes of wall-clock time and 7.6 hours of processor time. The slowest module took 173 s. The rest of the development built in about two minutes.

5.2 Independent Checks

The certificates rely on the kernel's GMP arithmetic far more than ordinary proofs do, so we checked the proof again with nanoda 0.4.17[1], a type checker for Lean written in Rust that comes with the toolchain. Its arithmetic on natural numbers is done by the Rust library num-bigint, and it reads the exports of lean4export. The check has three parts. Part A is the closure of the main theorems, with the 3418 certificates replaced by axioms of the same names and types. Part B is the closure of each certificate module with the standard axioms; these closures are checked 16 modules at a time. We then use a script to check that each certificate

[1] https://github.com/ammkrn/nanoda_lib

Table 2. lean4body v1.0 and the version in the Palomar registry. Times are wall-clock unless marked. The local times of v1.0 are on a machine with 32 hardware threads, and those of the registry version on a machine with 384 hardware threads that was also running other jobs; the registry version was built with at most 16 modules at a time.

		v1.0 registry version
Cells	136,448	17,056
Certificate theorems / modules	3418 / 211	7307 / 185
Build	21 min	12 min
Processor time	7.6 h (certificates)	2.8 h (all)
Memory of one module	about 5 GB	about 3.4 GB
nanoda, parts A, B and glue	29 min	9 min
Continuous integration, GitHub	–	28 min
Registry verification job	–	2 h 55 min

assumed in part A is proved exactly once in part B with the same type, that the declarations shared by the two parts are identical, and that part B uses no axiom except `propext`; declarations are compared by SHA-256 hashes of a canonical form. lean4export panics on a missing root name but still exits with status 0, so the script also searches the error output for a panic. The three parts took 29 minutes. As a control, a script adds 1 to a hint in the statement of a certificate in the export of its module, and nanoda rejects the result.

5.3 A Version for a Public Registry

We also submitted a version of the proof to the Palomar registry[2] of Lean formalizations. The registry requires the module system, at most 10,000 lines per file and the standard axioms, and its verification job must finish within 5.5 hours on 16 processors with 32 GB of memory. The job runs `lake comparator`. This tool checks that the statements in a `Challenge` file, which imports only Mathlib and copies the definitions, are those proved in a `Solution` file; it also checks the axioms and replays the solution in Lean's kernel and in nanoda. The registry replays the solution in con-ron as well, another type checker written in Rust that has its own arithmetic. We first ported lean4body to the module system as it was, but unfortunately the port did not fit: `leanchecker` alone took 8 hours, and con-ron needed more than 32 GB.

The registered version proves the same statements, Theorem B with its constant 1/4 included, and differs in the certificates. Cells twice as large in each direction turned out to be enough: $20 \times 14 \times 28$ and $4 \times 48 \times 48$ cells, 17,056 in all, one eighth of the original number. We wrote a new, greedy generator for the hints. It splits a box while a split along some dimension makes the box cheaper to check and localizes each leaf as cheaply as possible, and where the y chosen at the centre of a box does not certify it, it searches the 32-bit fields of y for one that does; this search lowered the predicted checking time by about 6%. A

[2] https://palomar-registry.org

further program, `Cut.lean`, replays the hints and predicts the kernel time of each box in each cell's tree of splits, in units of about 0.75 s. A cell over 4 units is cut into the topmost boxes of the tree that take at most 4 units, and each of these boxes is certified by a separate theorem. We chose this fine cut because of con-ron, whose memory grows with the time it spends on a single declaration. The predicate `Certified` in `Grid.lean` accepts a box if `checkBoxH` succeeds on it or if both halves of one of its splits are certified, and `Cap.lean` proves it sound. The result has 7307 theorems in 185 modules: 1309 for runs of cells and 5998 for pieces. Of the leaf boxes without a split, 44 remain above 4 units, the largest at 4.90. Outside the certificates the changes are small: the grids change in the checker, `Cap.lean` assembles the new certificates, every file became a module, and four of the definitions that the `Challenge` file copies moved to `Defs.lean`. Table 2 compares the two versions.

The registry's run took 10,507 s, 53% of its limit, with 508 s for the build, 17 s for the export and 9,859 s for the comparator. On a machine with 384 hardware threads, `leanchecker` replayed the registered solution in 17,752 s, against 9,855 s for the solution of version 1, whose certificates, made by an earlier version of the generator, prove only $\operatorname{tr} S(y) < 127/128$; the two ran side by side on the two hardware threads of one core. Their ratio, 1.80, is close to that of the predicted units, 1.83 (20,472 against 11,193). The control also had to be changed. Since a hint is replayed rather than trusted, a changed hint can still certify the cell: in version 1, adding 1 to the hint of the control changed only the coordinate of the first split, Lean's kernel proved the changed statement true, and nanoda, correctly, accepted it. The control therefore adds the smallest number for which Lean's kernel proves the changed statement false.

The repository of the registered version also runs a continuous-integration check on GitHub-hosted runners with four processors and 16 GB of memory. It runs three checkers: Lean's kernel (through `leanchecker`), nanoda and con-ron. Con-ron declines any axiom other than the standard three, so it checks the certificates, in part B, and not part A. The run took 28 minutes: 20 shards of 10 to 17 minutes each, then a final job of 11 minutes. Summed over the shards, the builds took 1.1 h, `leanchecker` 3.0 h, nanoda 2.8 h and con-ron 7.1 h, and no single process in a shard needed more than 0.9 GB of memory. All three checkers rejected the control.

6 Related Work

Proof by reflection [6] and the use of a proof assistant's evaluator for large computations [5] are old ideas. Coq gained a compiled implementation of strong reduction [13] and later machine integers and arrays in the kernel [4], which made checks such as those of SAT certificates feasible. Lean 4 [10] accelerates arithmetic on `Nat` literals in the kernel, and we rely on this for all of our certificates. The tactic `native_decide` offers compiled evaluation at the price of an axiom.

In Coq, Melquiond's methods for proving bounds by computation [25] grew into CoqInterval [24]; in HOL Light, Solovyev and Hales verified nonlinear

inequalities with Taylor intervals [30]. Their tool served the formal proof of the Kepler conjecture [14], whose more than 23,000 inequalities took approximately 5000 processor-hours on Microsoft Azure. Our computation is much smaller, but the claims it has to establish are not inequalities over boxes (Sect. 2.4). This is why our hints record localizations and witnesses as well as splits. Tucker's computer-assisted proof that the Lorenz attractor exists [37] was checked again by Immler in Isabelle/HOL with a verified ODE solver [17,18], and Gonthier's proof of the four colour theorem [12] runs its case analysis inside Coq.

The closest relatives in method are the SAT-based proofs in combinatorics, of the Boolean Pythagorean triples theorem [15], of Keller's conjecture [7] and of the empty hexagon theorem [16]. Their proofs were checked by proof checkers, including the verified checker cake_lpr [33], and the encoding of the empty hexagon theorem was verified in Lean [31]. There, as here, an untrusted search produces a certificate that a small trusted checker validates. In our case the checker is a Lean function, and the kernel is the only trusted executor. Lean4Lean [8] is a step toward a verified type checker for Lean, which would shrink the trusted base of results like ours further.

7 Discussion

The constant $1/4$ in Theorem B can be traded for checking time. The covering argument needs only some $\varepsilon > 0$, and on the same grids, with the same generator, certificates of $\operatorname{tr} S(y) < 127/128$ have about half the predicted checking time of those of $\operatorname{tr} S(y) < 3/4$ (11,193 units against 22,025). Version 1 of the same registry entry uses this threshold and proves only $Q \geq K/128$.

Parts of the checker can be reused. `Num.lean` and `Jet.lean` do not depend on the problem, and the carrier construction applies to any formula written against `Ops`. `Search.lean`, however, is specific to claims of the form (C1), with one variable localized and a witness chosen for each localized piece.

The design also has limits. The fixed-point format has absolute precision 2^{-96} and no values below -2^{160}. This is enough here because all quantities are bounded on the two regions, but the format would not be suitable for quantities of widely different sizes. Since the jets are first order, the boxes have to be small; higher-order Taylor models [30] might reduce the number of cells. Finally, Lean's kernel trusts GMP for the arithmetic of the certificates. This is one reason why we also ran nanoda and con-ron: nanoda does its arithmetic with num-bigint, and con-ron with its own code.

Use of Generative AI. Generative AI tools were used to find and prove the results of [35], to write the programs and the Lean formalization described in this paper, and to draft the text.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement

The formalization, the Python programs, and the logs of the build and the nanoda check are in the repository https://github.com/tejasvi/lean4body, release v1.0 (commit `58ac0fbf4d66`), archived on Zenodo [36]. The registry version is https://github.com/tejasvi/lean4body-uniqueness at commit `353653eceec4`, version 2 of entry PALOMAR-2026-10-02-000011 of the Palomar registry (https://palomar-registry.org/entry.html?id=PALOMAR-2026-10-02-000011&version=2), verified in run 37187361609 of https://github.com/PalomarRegistry/PalomarSubmission; its continuous-integration run is run 37182678749 of https://github.com/tejasvi/lean4body-uniqueness. Version 1 of the entry, at commit `9da70faf3423`, uses the threshold 127/128. The toolchain is Lean v4.35.0-rc3 with Mathlib at the matching tag; the Python programs use Python 3.14.4 and python-flint 0.9.0.

References

1. Albouy, A.: The symmetric central configurations of four equal masses. In: Hamiltonian Dynamics and Celestial Mechanics. Contemp. Math., vol. 198, pp. 131–135. AMS (1996)
2. Albouy, A., Cabral, H.E., Santos, A.A.: Some problems on the classical n-body problem. Celest. Mech. Dyn. Astron. **113**, 369–375 (2012)
3. Albouy, A., Fu, Y., Sun, S.: Symmetry of planar four-body convex central configurations. Proc. R. Soc. A **464**, 1355–1365 (2008)
4. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Interactive Theorem Proving (ITP 2010). LNCS, vol. 6172, pp. 83–98. Springer (2010)
5. Barendregt, H., Barendsen, E.: Autarkic computations in formal proofs. J. Autom. Reasoning **28**(3), 321–336 (2002)
6. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software (TACS 1997). LNCS, vol. 1281, pp. 515–529. Springer (1997)
7. Brakensiek, J., Heule, M., Mackey, J., Narváez, D.: The resolution of Keller's conjecture. In: Automated Reasoning (IJCAR 2020). LNCS, vol. 12166, pp. 48–65. Springer (2020)
8. Carneiro, M.: Lean4Lean: Towards a verified typechecker for Lean, in Lean. arXiv:2403.14064 (2024)
9. Corbera, M., Cors, J.M., Roberts, G.E.: Classifying four-body convex central configurations. Celest. Mech. Dyn. Astron. **131**, Paper 34 (2019)
10. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Automated Deduction (CADE 28). LNCS, vol. 12699, pp. 625–635. Springer (2021)
11. Dziobek, O.: Über einen merkwürdigen Fall des Vielkörperproblems. Astron. Nachr. **152**, 33–46 (1900)
12. Gonthier, G.: Formal proof—the four-color theorem. Notices Amer. Math. Soc. **55**(11), 1382–1393 (2008)
13. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: International Conference on Functional Programming (ICFP 2002), pp. 235–246. ACM (2002)

14. Hales, T., Adams, M., Bauer, G., Dang, T.D., Harrison, J., Hoang, L.T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, Q.T., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, T.H.A., Tran, N.T., Trieu, T.D., Urban, J., Vu, K., Zumkeller, R.: A formal proof of the Kepler conjecture. Forum Math. Pi **5**, e2 (2017)
15. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In: Theory and Applications of Satisfiability Testing (SAT 2016). LNCS, vol. 9710, pp. 228–245. Springer (2016)
16. Heule, M.J.H., Scheucher, M.: Happy ending: An empty hexagon in every set of 30 points. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). LNCS, vol. 14570, pp. 61–80. Springer (2024)
17. Immler, F.: Verified reachability analysis of continuous systems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). LNCS, vol. 9035, pp. 37–51. Springer (2015)
18. Immler, F.: A verified ODE solver and the Lorenz attractor. J. Autom. Reasoning **61**, 73–111 (2018)
19. Johansson, F.: Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. IEEE Trans. Comput. **66**(8), 1281–1292 (2017)
20. Krawczyk, R., Neumaier, A.: Interval slopes for rational functions and associated centered forms. SIAM J. Numer. Anal. **22**(3), 604–616 (1985)
21. Leandro, E.S.G.: Finiteness and bifurcations of some symmetrical classes of central configurations. Arch. Ration. Mech. Anal. **167**, 147–177 (2003)
22. Long, Y., Sun, S.: Four-body central configurations with some equal masses. Arch. Ration. Mech. Anal. **162**, 25–44 (2002)
23. MacMillan, W.D., Bartky, W.: Permanent configurations in the problem of four bodies. Trans. Amer. Math. Soc. **34**, 838–875 (1932)
24. Martin-Dorel, É., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in Coq. J. Autom. Reasoning **57**(3), 187–217 (2016)
25. Melquiond, G.: Proving bounds on real-valued functions with computations. In: Automated Reasoning (IJCAR 2008). LNCS, vol. 5195, pp. 2–17. Springer (2008)
26. Moeckel, R.: Central configurations. In: Llibre, J., Moeckel, R., Simó, C.: Central Configurations, Periodic Orbits, and Hamiltonian Systems, pp. 105–167. Advanced Courses in Mathematics CRM Barcelona, Birkhäuser (2015)
27. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM (2009)
28. Pérez-Chavela, E., Santoprete, M.: Convex four-body central configurations with some equal masses. Arch. Ration. Mech. Anal. **185**, 481–494 (2007)
29. Shub, M.: Appendix to Smale's paper: Diagonals and relative equilibria. In: Manifolds – Amsterdam 1970. Lecture Notes in Math., vol. 197, pp. 199–201. Springer (1971)
30. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: NASA Formal Methods (NFM 2013). LNCS, vol. 7871, pp. 383–397. Springer (2013)
31. Subercaseaux, B., Nawrocki, W., Gallicchio, J., Codel, C., Carneiro, M., Heule, M.J.H.: Formal verification of the empty hexagon number. In: Interactive Theorem Proving (ITP 2024). LIPIcs, vol. 309, pp. 35:1–35:19 (2024)
32. Sun, S., Xie, Z., You, P.: On the uniqueness of convex central configurations in the planar 4-body problem. Regul. Chaotic Dyn. **28**, 512–532 (2023)
33. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021). LNCS, vol. 12652, pp. 223–241. Springer (2021)

34. The mathlib Community: The Lean mathematical library. In: Certified Programs and Proofs (CPP 2020), pp. 367–381. ACM (2020)
35. Tomar, T.S.: Uniqueness of four-body convex central configurations for all masses. arXiv:2609.35632 (2026)
36. Tomar, T.S.: lean4body: programs, certificates and Lean formalization for the uniqueness of convex four-body central configurations, version 1.0. Zenodo (2026). https://doi.org/10.5281/zenodo.23004297
37. Tucker, W.: A rigorous ODE solver and Smale's 14th problem. Found. Comput. Math. **2**(1), 53–117 (2002)
38. Xia, Z.: Convex central configurations for the n-body problem. J. Differential Equations **200**, 185–190 (2004)