

Rigorous Solution of Linear and Nonlinear Systems using Interval Arithmetic in Python

Oliver Williams

January 2026

Abstract

`pyIA` is a compact Python implementation of interval arithmetic, where each real quantity is represented by a closed interval $[\underline{x}, \bar{x}]$ that is guaranteed to contain the true value. Arithmetic operations are defined to produce enclosures that account for rounding effects, enabling basic verified numerical computation.

The implementation is guided by the principles and techniques described by Siegfried M. Rump, and is inspired by INTLAB, his interval arithmetic toolbox for MATLAB, with the aim of providing a small, readable, and extensible Python code base.

Contents

1	Introduction	3
2	Background and Related Work	3
2.1	Interval arithmetic	3
2.2	Verified computing and INTLAB	3
2.3	Comparison to other Python interval libraries	3
2.4	Comparison to INTLAB	4
2.5	Coverage of Rohn’s <i>INTLAB Primer</i>	4
2.6	Notable problems solvable in <code>pyIA</code>	5
2.7	Midpoint-Radius Arithmetic	5
2.8	Automatic Differentiation (Forward Mode)	6
3	Mathematical Foundations	6
3.1	Interval representation	6
3.2	Basic operations and inclusion property	6
3.3	Rigorous Input and the “0.1 Problem”	7
3.4	Elementary Functions and Monotonicity	7
3.5	Midpoint-Radius Matrix Multiplication	7

4 Design and Implementation of pyIA	8
4.1 Data Structures and API	8
4.2 Project layout (code overview)	8
4.3 Quickstart: how to use the code	9
4.4 Arithmetic Operators and Rounding Safety	10
4.5 Midpoint-Radius Matrix Multiplication	11
4.6 Elementary Functions	11
4.7 Verified Nonlinear System Solver (<code>verifynlss</code>)	11
4.8 Verified Linear System Solver (<code>verifylss</code>)	14
4.9 Sparse interval matrices and <code>sparselss</code>	16
5 Geometric Visualisation of Solution Sets	18
5.1 The General Problem: Characterizing Σ	18
5.2 Algorithm for Solution Set Characterization	19
5.3 Case Study: A Diagonally Dominant System	20
5.4 Extension to 3D visualisation	21
5.4.1 Case Study: 3D Interaction Matrix	21
5.5 Analysis	22
6 Testing and Validation	22
6.1 Pytest test suite	22
6.2 Validation of Automatic Differentiation	23
6.3 Integration Testing: The Abbott–Brent Problem	23
6.4 SIAM benchmark: Broyden tridiagonal function	25
7 Results and Discussion	27
7.1 Algorithmic Efficiency: Midpoint-Radius Arithmetic	27
7.2 Dense vs. sparse scaling	27
7.3 Analysis of Verification Case Studies	28
7.4 Critique and Limitations	29
8 Conclusion and Future Work	29
8.1 Future Work	30

1 Introduction

This report documents `pyIA`, an interval arithmetic implementation in Python, and evaluates it against the implementation principles described by Siegfried M. Rump (including the INTLAB approach). The goal of `pyIA` is not feature parity with INTLAB, but a compact, readable code base that demonstrates how to compute rigorous enclosures in floating-point arithmetic.

While many existing Python libraries provide basic interval datatypes, they often lack the rigorous mathematical machinery required for solving complex systems. This project extends the foundational interval types to support verified scientific computing workflows. Specifically, we address three implementation challenges often overlooked in pedagogical software:

1. **Rigorous I/O:** Handling the conversion error between decimal literals (e.g., "0.1") and binary floating-point representations.
2. **Automatic Differentiation:** Implementing operator-overloaded forward-mode differentiation to automatically accumulate Jacobian matrices.
3. **Verified Solvers:** Implementing the Krawczyk operator to prove the existence and uniqueness of solutions to nonlinear systems.

2 Background and Related Work

2.1 Interval arithmetic

An interval is denoted $x = [\underline{x}, \bar{x}]$ with $\underline{x} \leq \bar{x}$ and represents the set of reals $\{t \in \mathbb{R} : \underline{x} \leq t \leq \bar{x}\}$. Arithmetic is lifted to intervals so that the result encloses all possible pointwise results. The key specification is the *inclusion property*: for an interval operation \circ and any $\tilde{x} \in x$, $\tilde{y} \in y$, one should have $\tilde{x} \circ \tilde{y} \in x \circ y$.

2.2 Verified computing and INTLAB

Rump's INTLAB is a mature environment for interval arithmetic and verified numerical algorithms. The attached paper describes practical techniques for implementing rigorous bounds efficiently on IEEE-754 hardware, including careful use of directed rounding and specialised linear algebra kernels.

2.3 Comparison to other Python interval libraries

There are several Python packages that expose “intervals”, but they serve different goals. Many focus on interval sets and set operations (useful for constraints, logic, and symbolic reasoning), or on arbitrary-precision interval arithmetic, but do not attempt to reproduce the INTLAB-style model of fast, BLAS-backed, IEEE-754 verified linear algebra.

In contrast, `pyIA` is written explicitly around (i) outward rounding on IEEE-754 hardware (via a small rounding-mode extension), and (ii) interval matrix

operations that are expressed in terms of a few dense floating-point GEMMs (midpoint–radius arithmetic). This is why the project naturally includes verified solvers (`verifylss` / `verifynlss`) as first-class features: the interval type is not only a container, but an executable foundation for self-validating algorithms.

2.4 Comparison to INTLAB

INTLAB remains far more complete and robust than `pyIA`. It implements a broad collection of interval datatypes (including complex intervals and decorated intervals), highly optimised verified linear algebra, and many specialized routines.

The purpose of `pyIA` is different: it is a compact, readable Python codebase that re-implements the key ideas needed to build verified algorithms:

1. directed rounding (so that each elementary bound computation is rigorous),
2. midpoint–radius matrix multiplication (so verified matrix products are practical), and
3. Krawczyk-type verification (so $Ax = b$ and $f(x) = 0$ can be solved with proofs).

Because NumPy already provides optimised dense linear algebra, `pyIA` can be “surprisingly capable” despite being much smaller than INTLAB, but it is not intended to be a drop-in replacement.

2.5 Coverage of Rohn’s *INTLAB Primer*

A guiding reference for the user-facing API is Rohn’s short primer on INTLAB[3]. In it he discusses the following utilities: creating intervals, extracting bounds, rigorous representation of decimal inputs, intersection/hull utilities, and verified solution of interval linear systems.

The corresponding tasks can be reproduced in `pyIA` with small syntactic changes:

- **Interval construction by endpoints:** INTLAB’s `infsup(A,B)` corresponds to `Interval(A,B)`.
- **Midpoint–radius construction:** INTLAB’s `midrad(m,r)` corresponds to `midrad(m,r)` in `pyia.py`.
- **Extracting components:** INTLAB’s `A.inf`, `A.sup`, `A.mid`, `A.rad` correspond to `A.inf`, `A.sup`, `A.mid`, `A.rad` in `pyIA`.
- **Rigorous decimal input:** INTLAB’s `intval('0.1')` corresponds to `Interval("0.1")`.
- **Inverse enclosure:** INTLAB’s `inv(A)` corresponds to `A.inverse()` (implemented by solving $AX = I$ via `verifylss`).

- **Verified linear systems:** INTLAB’s `verifylss(A,b)` corresponds directly to `verifylss(A,b)`.
- **Set utilities:** `intersect` and `hull` exist in `pyIA`; `mig` exists (as `x.mig`), and inclusion-in-interior exists as `x.in0(y)`.

A few primer features are *not* identical: INTLAB has additional display configuration (e.g. `intvalinit('displayinfsup')`) and more built-in functions (e.g. `in(x,y)` as a separate predicate and a broader library of elementary functions). In `pyIA` the focus is the verified arithmetic and solver pipeline rather than full UI parity. These are relatively trivial to implement in `pyIA` if there is ever sufficient need for them.

2.6 Notable problems solvable in `pyIA`

The codebase is intended to support “real” verified computing workflows, not only toy arithmetic. Examples that fit naturally in the current implementation include:

- **Tolerance analysis / uncertain linear systems:** compute a verified enclosure for $\mathbf{Ax} = \mathbf{b}$ using `verifylss`, and visualise the (typically non-box) solution set using `visualise.py` (Oettli–Prager filtering).
- **Verified inverse enclosures:** compute an enclosure for \mathbf{A}^{-1} via `A.inverse()`, which reduces inversion to a verified linear solve.
- **Geometric constraint solving:** small systems such as circle–line intersection can be solved and verified using `verifynlss` (see `test_suite.py`).
- **Nonlinear boundary value problems:** discretized nonlinear systems such as the Abbott–Brent example can be verified using `verifynlss` together with the AD machinery (`Gradient + initvar`).

2.7 Midpoint-Radius Arithmetic

A naive implementation of interval matrix multiplication $A \cdot B$ is computationally expensive, requiring eight floating-point matrix multiplications to bound the resulting intervals. Rump introduced the concept of *Midpoint-Radius arithmetic* to mitigate this. By representing an interval matrix as $A = \langle M_A, R_A \rangle$ (where M is the midpoint and R is the radius), operations can be formulated to exploit optimised BLAS (Basic Linear Algebra Subprograms) kernels. This allows interval operations to run with a small constant overhead compared to standard floating-point operations, a principle `pyIA` adopts for its matrix product implementation.

2.8 Automatic Differentiation (Forward Mode)

Verified nonlinear solvers, such as the Interval Newton method, require the evaluation of the Jacobian matrix $J_f(X)$ over an interval domain. Symbolic differentiation is often computationally intractable for large iterative systems, and numerical differentiation (finite differences) introduces truncation errors that break rigor. Forward-mode Automatic Differentiation (AD) solves this by decomposing functions into elementary operations. By associating a dual number (u, u') with every variable, derivatives are accumulated alongside values via the Chain Rule:

$$(u, u') \cdot (v, v') = (uv, uv' + vu')$$

This technique is essential for the `Gradient` class in `pyIA`, enabling the solver to handle arbitrary user-defined functions without manual derivative derivation.

3 Mathematical Foundations

3.1 Interval representation

In `pyIA`, the interval data structure is designed using a “Structure of Arrays” (SoA) layout. An interval vector or matrix is represented by two NumPy `float64` arrays, `inf` and `sup`, rather than an array of interval objects. Derived quantities include the midpoint $\text{mid}(x) = \frac{1}{2}(\underline{x} + \bar{x})$ and radius $\text{rad}(x) = \frac{1}{2}(\bar{x} - \underline{x})$. This layout allows the implementation to leverage vectorised CPU instructions (SIMD) via NumPy ufuncs.

3.2 Basic operations and inclusion property

Basic arithmetic operations $+, -, \times$ are defined to satisfy the inclusion property. In floating-point arithmetic, lower bounds are computed with rounding towards $-\infty$ (∇) and upper bounds with rounding towards $+\infty$ (Δ).

Addition and Subtraction:

$$X + Y = [\nabla(\underline{x} + \underline{y}), \Delta(\bar{x} + \bar{y})]$$

$$X - Y = [\nabla(\underline{x} - \bar{y}), \Delta(\bar{x} - \underline{y})]$$

Division: Division is defined for intervals Y that do not contain zero. Since the function $f(y) = 1/y$ is monotonically decreasing, the bounds of the reciprocal map inversely to the bounds of the argument. `pyIA` implements this by swapping the rounding context:

$$X/Y = X \times [1/\bar{y}, 1/\underline{y}]$$

Practically, the new lower bound is computed using the divisor’s upper bound (rounded down), and the new upper bound using the divisor’s lower bound (rounded up).

3.3 Rigorous Input and the “0.1 Problem”

A critical challenge in verified computing is the initialisation of intervals from decimal literals. The standard IEEE-754 representation of the decimal 0.1 is slightly larger than $1/10$. Consequently, the naive initialisation `Interval(0.1)` creates a point interval that fails to enclose the true mathematical value.

`pyIA` addresses this via a rigorous string parsing routine. A decimal string $S = "m \cdot 10^e"$ is decomposed into an integer mantissa m and exponent e . The interval is constructed via:

$$X = \text{Interval}(m) \times 10^e \quad (1)$$

The power of 10 is computed with directed rounding, and the multiplication by the exact integer m ensures that the resulting interval strictly encloses the user’s decimal input.

3.4 Elementary Functions and Monotonicity

For monotonic functions like $\exp(x)$, the image of an interval is simply the function mapped to the endpoints. However, for periodic functions implemented in `pyIA` (specifically \sin and \cos), non-monotonicity must be handled.

The implementation checks if local extrema fall within the input interval X . For $\sin(X)$:

- If $\exists k \in \mathbb{Z}$ such that $2k\pi + \pi/2 \in X$, then $\sup(\sin(X)) = 1.0$.
- If $\exists k \in \mathbb{Z}$ such that $2k\pi + 3\pi/2 \in X$, then $\inf(\sin(X)) = -1.0$.

Otherwise, the bounds are determined by the function values at the endpoints \underline{x} and \bar{x} .

3.5 Midpoint-Radius Matrix Multiplication

To achieve performance comparable to standard floating-point libraries, `pyIA` implements matrix multiplication using Midpoint-Radius arithmetic (Rump’s Algorithm 2.7). Instead of computing $8n^3$ operations for endpoint multiplication, we compute:

$$A \cdot B \subseteq \langle M_A M_B, |M_A|R_B + R_A(|M_B| + R_B) + \text{err}(M_A M_B) \rangle \quad (2)$$

where M and R denote the midpoint and radius matrices, and $\text{err}(M_A M_B)$ denotes the rounding error of the midpoint product (handled explicitly in code by comparing the downward- and upward-rounded midpoint products). The radius accumulation is performed with upward rounding, and the final interval is returned as $[M_C - R_C, M_C + R_C]$.

4 Design and Implementation of pyIA

4.1 Data Structures and API

The library is built upon two core classes designed to abstract the complexity of verified arithmetic: `Interval` for set-based computations and `Gradient` for automatic differentiation.

The Interval Class: Defined in `pyia.py`, this class is the fundamental building block. Unlike naive implementations that might store a list of interval objects, pyIA adopts a "Structure of Arrays" (SoA) layout. An instance of `Interval` representing a vector or matrix stores two contiguous NumPy arrays:

- `self.inf`: Array of lower bounds (\underline{x}).
- `self.sup`: Array of upper bounds (\bar{x}).

This design allows the Python interpreter to dispatch entire array operations to optimised C-level BLAS routines (via NumPy) in a single call, amortizing the overhead of the interpreted language.

The Gradient Class: Defined in `ad.py`, this class implements Forward-Mode Automatic Differentiation. Each `Gradient` object encapsulates a value pair (u, J_u) , where u is the function value (scalar or `Interval`) and J_u is the Jacobian matrix (derivative w.r.t independent variables). This structure enables the library to compute derivatives of arbitrary complexity without symbolic engines.

4.2 Project layout (code overview)

The implementation is split into small, mostly self-contained modules:

- `pyia.py`: the `Interval` class and utility constructors (notably `midrad`). This file also implements rigorous string-to-interval input via `str2interval` and midpoint-radius matrix multiplication via `_matmul_`.
- `pyia_kernel.py`: rounding-mode control. This wraps the compiled `rounding` extension and provides `setround`, `getround`, and a safety-oriented `RoundingMode` context manager.
- `rounding.c`: the low-level interface to IEEE-754 rounding modes using `fenv.h` (via `fesetround/fegetround`).
- `solvers.py`: verified solvers (`verifylss` for dense linear systems, `sparselss` for sparse linear systems, and `verifynlss` for nonlinear systems).
- `sparse.py`: sparse interval matrix support via `SparseIntervalMatrix` (CSR-backed) for verified sparse matrix–vector products.
- `ad.py`: a minimal forward-mode AD engine (`Gradient` and `initvar`) used by `verifynlss` to obtain Jacobians automatically.

- `visualise.py`: plotting utilities for the solution set of $Ax = b$ in 2D/3D using the Oettli–Prager inequality, overlaid with the verified hull from `verifylss`.
- `run_dense.py` / `run_sparse.py`: scripts used to time the dense and sparse linear solvers.
- `plot_sparse_dense.py`: generates the comparison plot `performance_comparison.png`.
- `test_suite.py`: pytest test suite covering interval arithmetic, automatic differentiation, and solver routines (dense, sparse, and nonlinear).

4.3 Quickstart: how to use the code

This section is intended as a short, practical entry point. The project is designed so that a user can write almost-normal NumPy code, replacing scalars/arrays with `Interval` objects when verified bounds are needed.

Creating intervals.

```

1 from pyia import Interval, midrad
2
3 x = Interval(1.0, 2.0)           # explicit bounds
4 v = Interval([0.0, 1.0])        # point interval vector
5 w = midrad([0.0, 1.0], 1e-3)    # midpoint-radius constructor
6
7 # Rigorous decimal I/O (avoid the "0.1" problem)
8 a = Interval("0.1")

```

Listing 1: Constructing intervals in pyIA

Verified linear solve $Ax = b$ (dense).

```

1 import numpy as np
2 from pyia import Interval
3 from solvers import verifylss
4
5 A = Interval([[2.0, -1.0], [-1.0, 2.0]])
6 b = Interval([1.0, 0.0])
7
8 x = verifylss(A, b)
9 print(x) # Interval enclosure of the unique solution (if
           verification succeeds)

```

Listing 2: Verified dense linear solve via `verifylss`

Verified linear solve $Ax = b$ (sparse). For very large n , dense verified linear algebra becomes memory-bound. The project therefore includes a sparse interval matrix type and a sparse verified solver.

```

1 import numpy as np
2 import scipy.sparse as sp
3 from pyia import Interval

```

```

4 from sparse import SparseIntervalMatrix
5 from solvers import sparselss
6
7 N = 100000
8 # Tridiagonal SPD example
9 mid_A = sp.diags([np.full(N, 4.0), np.full(N-1, -1.0), np.full(N-1,
-1.0)], [0, -1, 1], format='csr')
10 rad_A = sp.diags([np.full(N, 4e-5), np.full(N-1, 1e-5), np.full(N
-1, 1e-5)], [0, -1, 1], format='csr')
11
12 A = SparseIntervalMatrix(mid_A, abs(rad_A))
13
14 b = Interval(np.ones(N)) + Interval(-1e-5*np.ones(N), 1e-5*np.ones(
N))
15
16 # If a lower bound on sigma_min is known (e.g. by Gershgorin), pass
# it to skip eigenvalue estimation.
17 x = sparselss(A, b, sigma_lb=2.0)
18 print(x)

```

Listing 3: Verified sparse linear solve via `sparselss`

Verified nonlinear solve $f(x) = 0$. The nonlinear solver expects a function written in terms of Gradient arithmetic.

```

1 import numpy as np
2 from ad import Gradient, initvar
3 from solvers import verifynlss
4
5 def stack_gradients(gs):
6     vals = np.array([g.x for g in gs])
7     jacs = np.vstack([g.dx for g in gs])
8     return Gradient(vals, jacs)
9
10 def system(X):
11     x = X[0]
12     y = X[1]
13     eq1 = x*x + y*y - 1.0
14     eq2 = x - y
15     return stack_gradients([eq1, eq2])
16
17 root = verifynlss(system, [0.5, 0.5])
18 print(root)

```

Listing 4: Verified nonlinear solve via `verifynlss`

4.4 Arithmetic Operators and Rounding Safety

pyIA implements arithmetic via operator overloading (`__add__`, `__mul__`, etc.) in `pyia.py`. Rounding control is provided by a small C-extension (`rounding.c`) exposed through `pyia_kernel.setround(mode)`. The interval operators in the current codebase call `setround(-1)` / `setround(1)` around the lower/upper bound computations and then restore round-to-nearest with `setround(0)`.

To make this safer under exceptions, `pyia_kernel.py` also provides a `RoundingMode` context manager that saves the previous rounding mode (via `getround()`) and restores it on exit.

- **Basic Ops** ($+, -, \times$): Each operation explicitly switches the hardware FPU rounding mode. For multiplication, the four endpoint products are formed and combined with `min/max`, with the lower bound computed under rounding down and the upper bound under rounding up.
- **Division** ($/$): Division is implemented only when $0 \notin Y$ (otherwise `ZeroDivisionError` is raised). The reciprocal is formed as $1/Y = [\nabla(1/\bar{y}), \Delta(1/y)]$ and then multiplied using the existing interval multiplication.

4.5 Midpoint-Radius Matrix Multiplication

To achieve performance comparable to floating-point libraries, `pyIA` overrides the matrix multiplication operator (`@`) and implements the midpoint–radius enclosure algorithm described by Rump (Algorithm 2.7). Instead of the $8n^3$ operations required for naive endpoint-based interval multiplication, we compute:

$$A \cdot B \subseteq \langle M_A M_B, |M_A|R_B + R_A(|M_B| + R_B) + |M_A|R_B \rangle \quad (3)$$

In the implementation in `pyia.py`, the midpoint product $M_A M_B$ is computed using NumPy’s standard matrix multiplication (BLAS-backed). The correction/radius terms are then accumulated under upward rounding to preserve inclusion.

4.6 Elementary Functions

The library implements standard functions ($\exp, \log, \sin, \cos, \sqrt{x}$).

- **Monotonic Functions** (\exp, \log, \sqrt{x}): These are evaluated simply by mapping the function to the endpoints with directed rounding.
- **Periodic Functions** (\sin, \cos): Implementation requires rigorous handling of non-monotonicity. The method explicitly checks if local extrema (peaks at $\pi/2 + 2k\pi$ or troughs at $3\pi/2 + 2k\pi$) fall within the input interval. If so, the bounds are expanded to include exactly $+1.0$ or -1.0 , ensuring inclusion even for wide intervals.

4.7 Verified Nonlinear System Solver (`verifynlss`)

The routine `verifynlss(f, xs)` in `solvers.py` attempts to prove existence and uniqueness of a root of $f(x) = 0$ near an initial guess `xs`.

What the solver expects from the user

The user supplies `f` as code written using `Gradient` arithmetic. In other words, `f` is evaluated on a *vector of Gradient* variables created by `initvar(xs)` and must return a single `Gradient` object whose

- `.x` is the vector $f(x)$, and
- `.dx` is the Jacobian matrix $J_f(x)$.

The test suite `test_suite.py` shows a practical pattern: compute each equation as a scalar `Gradient` and then “stack” them into a single vector-valued `Gradient`.

Algorithm overview

The implementation is explicitly two-phase:

1. **Newton phase (floating point):** use AD to compute $J_f(x_k)$ and apply a standard Newton correction to obtain a high-quality approximate root.
2. **Verification phase (interval):** build an interval box around the approximate root and apply a Krawczyk-type operator until the inclusion test succeeds.

Executable algorithm (as implemented)

Algorithm 5 reproduces the `verifynlss` routine from the codebase.

```
1 def verifynlss(f, xs):
2     """
3         Verified Nonlinear System Solver (Algorithm 5.7)
4
5     Args:
6         f: A function f(x) written using standard operators.
7             Must accept and return Gradient objects.
8         xs: Initial guess (float list/array).
9
10    Returns:
11        Interval vector containing the unique root, or None.
12    """
13    xs = np.array(xs, dtype=np.float64)
14    n = len(xs)
15
16    print(" --- Phase 1: Floating Point Newton Iteration ---")
17    k = 0
18    kmax = 20
19    xs_old = xs.copy()
20
21    while k < kmax:
22        k += 1
23        xs_old = xs.copy()
24
25        # 1. initialise AD variables at current point xs
```

```

26     x_ad = initvar(xs)
27
28     # 2. Evaluate function. y contains value f(x) and Jacobian
29     # f'(x)
30     y = f(x_ad)
31
32     # 3. Newton Step: xs_new = xs - J^-1 * f(xs)
33     try:
34         delta = np.linalg.solve(y.dx, y.x)
35         xs = xs - delta
36     except np.linalg.LinAlgError:
37         print("Singular Jacobian in Newton phase.")
38         return None
39
40     if np.linalg.norm(xs - xs_old) < 1e-12 * np.linalg.norm(xs):
41         break
42
43     print(f"Newton converged to approx root in {k} iterations.")
44
45     print("---- Phase 2: Interval Verification (Krawczyk) ----")
46     try:
47         R = np.linalg.inv(y.dx)
48     except np.linalg.LinAlgError:
49         return None
50
51     xs_int = Interval(xs)
52
53     # Evaluate f at an interval point (width 0) to catch rounding
54     val_check = f(Gradient(xs_int))
55     Z = -Interval(R) @ val_check.x
56
57     hull_neg1_1 = Interval(np.full(n, -1.0), np.full(n, 1.0))
58     realmin = 1e-300
59     small_eps = midrad(0, 10 * realmin)
60
61     ready = False
62     k = 0
63     k_verif_max = 15
64     Y = Z
65
66     while not ready and k < k_verif_max:
67         k += 1
68
69         # Epsilon Inflation
70         E = (hull_neg1_1 * (Y.rad * 0.1)) + small_eps
71         X = Y + E
72
73         # Interval Jacobian over candidate box
74         x_ad_interval = initvar(xs_int + X)
75         y_interval = f(x_ad_interval)
76
77         I = Interval(np.eye(n))
78         C = I - (Interval(R) @ y_interval.dx)
79
80         Y_new = Z + (C @ X)

```

```

81     if Y_new.in0(X):
82         ready = True
83         Y = Y_new.intersect(X)
84     else:
85         Y = Y_new
86
87     if ready:
88         print(f"\u2705 Verified unique solution in {k} steps.")
89         return xs_int + Y
90     else:
91         print("\u274c Verification failed.")
92         return None

```

Listing 5: Verified nonlinear solver `verifynlss` (from `solvers.py`)

How to interact with `verifynlss`

In practice the workflow is: (1) write `f` using Gradient operators, (2) supply an initial numerical guess `xs`, and (3) interpret the returned Interval vector as a mathematically verified enclosure of the root.

The easiest starting point is to copy the nonlinear test pattern in `test_suite.py` and replace the example equations with your own system.

4.8 Verified Linear System Solver (`verifylss`)

The function `verifylss(A,b)` in `solvers.py` returns an interval vector that is proven to contain the unique solution of $Ax = b$ (or returns `None` if the verification cannot be completed).

Algorithm idea (what the code is doing)

The method is a Krawczyk-type fixed point enclosure for linear systems. The key design choice is to precondition with a *floating-point* approximate inverse $R \approx (\text{mid}(A))^{-1}$, and then do the enclosure steps using interval matrix operations:

1. Compute R in floating point (with one Newton refinement) so that $RA \approx I$.
2. Compute an approximate solution $x_s = R \text{ mid}(b)$.
3. Form the interval residual enclosure $Z = R(b - Ax_s)$.
4. Form the interval iteration matrix $C = I - RA$.
5. Iterate an inflated enclosure until $Y \subset \text{int}(X)$, which implies existence/uniqueness in $x_s + Y$.

Executable algorithm (as implemented)

Algorithm 6 reproduces the `verifylss` routine from the current codebase (with comments preserved).

```
1 def verifylss(A, b):
2     """
3         Verified Linear System Solver (Algorithm 5.1 with Refinement)
4         Solves Ax = b rigorously. Works for vectors (Ax=b) and matrices
5             (AX=I).
6         """
7
8     if not isinstance(A, Interval): A = Interval(A)
9     if not isinstance(b, Interval): b = Interval(b)
10
11    # --- STEP 1: Calculate Approximate Inverse with Newton
12    # Refinement ---
13    try:
14        # Initial floating-point guess
15        R_approx = np.linalg.inv(A.mid)
16
17        # Newton Refinement: R = R + R(I - AR)
18        # This makes R much closer to A^-1, shrinking the error
19        # term C later.
20        n = A.shape[0]
21        I_mat = np.eye(n)
22        Residual = I_mat - (A.mid @ R_approx)
23        R_approx = R_approx + (R_approx @ Residual)
24
25    except np.linalg.LinAlgError:
26        print("\u274c Matrix is singular (midpoint).")
27        return None
28
29    # --- STEP 2: Setup Residuals (Standard Algorithm 5.1) ---
30    # Approximate solution xs = R * mid(b)
31    xs = R_approx @ b.mid
32    xs_interval = Interval(xs)
33
34    # Residual Z = R * (b - A * xs)
35    Res = b - (A @ xs_interval)
36    R_int = Interval(R_approx)
37    Z = R_int @ Res
38
39    # Preconditioner Residual C = I - R*A
40    I = Interval(np.eye(n))
41    C = I - (R_int @ A)
42
43    # --- STEP 3: optimised Krawczyk Loop ---
44    Y = Z.copy()
45    k = 0
46    kmax = 100
47    ready = False
48
49    hull_neg1_1 = Interval(np.full(Y.shape, -1.0), np.full(Y.shape,
50        1.0))
51    realmin = 1e-300
52    small_eps = midrad(0, 10 * realmin)
53
54    print("Starting optimised Krawczyk iteration...")
```

```

50     while not ready and k < kmax:
51         k += 1
52
53         # 1. Epsilon Inflation (Explode slightly to find inclusion)
54         E = (hull_neg1_1 * (Y.rad * 0.1)) + small_eps
55         X = Y + E
56
57         # 2. Contraction Step
58         Y_new = Z + (C @ X)
59
60         # 3. Intersection Step (The Squeeze)
61         Y_intersect = Y_new.intersect(X)
62         if np.any(np.isnan(Y_intersect.inf)):
63             pass
64         else:
65             Y_new = Y_intersect
66
67         # 4. Convergence Check
68         if Y_new.in0(X):
69             ready = True
70
71         Y = Y_new
72
73     if ready:
74         print(f"\u2705 Converged in {k} iterations.")
75         return xs_interval + Y
76     else:
77         print(f"\u274c Failed to converge after {k} iterations.")
78     return None

```

Listing 6: Verified linear solver `verifylss` (from `solvers.py`)

How to interact with `verifylss`

From a user perspective, the only requirement is that `A` and `b` can be promoted to `Interval`. If `A` is an $n \times n$ interval matrix and `b` is an n -vector interval, the return value is an n -vector interval enclosing the unique solution.

If the midpoint matrix `mid(A)` is singular, the routine returns `None`. If verification fails to establish inclusion within `kmax` iterations, it also returns `None`.

4.9 Sparse interval matrices and `sparselss`

For large-scale problems, dense interval matrices are infeasible because they require $O(n^2)$ memory and dense factorizations are $O(n^3)$ in time. To address this, pyIA includes a sparse interval matrix type and a sparse verified solver.

Sparse interval matrix type

The class `SparseIntervalMatrix` (in `sparse.py`) stores the midpoint and radius matrices as SciPy CSR matrices. The primary supported operation is

a verified sparse matrix–vector product $y = Ax$, implemented as a midpoint product plus a rigorously rounded radius bound.

Verified sparse solver idea

The function `sparselss` (in `solvers.py`) targets symmetric positive definite (SPD) systems and implements a normwise residual bound. Given an approximate (floating-point) solution x_s and a lower bound σ on the smallest singular value of $\text{mid}(A)$, we can bound the error by

$$\|x - x_s\|_\infty \leq \frac{\|b - Ax_s\|_\infty}{\sigma}.$$

The solver therefore returns the interval enclosure $[x_s - \varepsilon, x_s + \varepsilon]$ with $\varepsilon = \|\text{Residual}\|_\infty / \sigma$.

Executable algorithm (as implemented)

Algorithm 7 reproduces the current `sparselss` implementation.

```

1 def sparselss(A: SparseIntervalMatrix, b: Interval, sigma_lb=None):
2     """
3         Verified Sparse Linear Solver (Algorithm 5.5 from Rump).
4         Solves Ax=b where A is Symmetric Positive Definite.
5
6     Args:
7         A: SparseIntervalMatrix
8         b: Interval Vector (Dense)
9         sigma_lb (float, optional): A known lower bound for the
10            smallest singular value.
11            If None, the solver attempts to
12            estimate it numerically.
13        """
14    n = A.shape[0]
15
16    # 1. Approximate Solution (Floating Point)
17    xs = spla.spsolve(A.mid, b.mid)
18
19    # 2. Minimum Singular Value (Sigma_min)
20    if sigma_lb is not None:
21        sigma = sigma_lb
22    else:
23        try:
24            vals = spla.eigsh(A.mid, k=1, which='SA',
25            return_eigenvectors=False)
26            sigma = vals[0] * 0.99 # 1% Safety margin
27        except Exception:
28            return None
29
30    if sigma <= 0:
31        return None # Matrix not Positive Definite
32
33    # 3. Rigorous Residual: Res = b - A @ xs
34    Ax = A @ xs
35    Residual = b - Ax

```

```

33
34     # 4. Error Bound
35     res_max = np.max(np.maximum(np.abs(Residual.inf), np.abs(
36         Residual.sup)))
37     err_bound = res_max / sigma
38
39     # 5. Return Enclosure
40     return Interval(xs - err_bound, xs + err_bound)

```

Listing 7: Verified sparse linear solver `sparselss` (from `solvers.py`)

How to interact with `sparselss`

In many structured SPD problems, a usable lower bound σ can be obtained analytically. For example, the tridiagonal matrix with diagonal 4 and off-diagonals -1 is strictly diagonally dominant, and Gershgorin circles can be used to justify a positive lower bound; `run_sparse.py` passes `sigma_lb=2.0` to avoid the cost of eigenvalue estimation.

5 Geometric Visualisation of Solution Sets

A fundamental challenge in interval arithmetic is the *wrapping effect*. When solving a linear system $\mathbf{A}x = \mathbf{b}$ where \mathbf{A} and \mathbf{b} are intervals, the set of all possible solutions Σ is rarely a simple box (interval vector). Instead, it is often a complex, star-shaped polytope.

Standard verified solvers, such as the Krawczyk method implemented in `verifylss`, compute the *interval hull* of this set—the tightest possible axis-aligned box containing Σ . To validate the tightness of our solver, we implemented a visualiser that explicitly maps the exact solution set Σ and compares it against the computed hull.

5.1 The General Problem: Characterizing Σ

For an interval matrix $\mathbf{A} = [\underline{A}, \bar{A}]$ and interval vector $\mathbf{b} = [\underline{b}, \bar{b}]$, the solution set is defined as:

$$\Sigma = \{x \in \mathbb{R}^n : \exists A \in \mathbf{A}, b \in \mathbf{b} \text{ such that } Ax = b\} \quad (4)$$

Calculating the boundary of Σ directly is computationally expensive. However, by the Oettli-Prager Theorem, a vector x belongs to Σ if and only if it satisfies the inequality:

$$|A_c x - b_c| \leq \Delta A |x| + \Delta b \quad (5)$$

where A_c, b_c are the midpoints and $\Delta A, \Delta b$ are the radius matrices. We implemented a sampling routine in `visualise.py` that utilises this theorem to verify point inclusion for a dense grid of x values, allowing us to render the exact shape of Σ in 2D.

5.2 Algorithm for Solution Set Characterization

To generate the exact solution set Σ , we rely on the Oettli–Prager Theorem [2]. Algorithm 6.1 presents the executable Python code used to visualise this set. It constructs a dense grid of points and filters them based on the Oettli–Prager inequality $|A_c x - b_c| \leq \Delta A|x| + \Delta b$.

```

1 def plot_solution_set(A, b, resolution=200):
2     """
3         visualises the exact solution set of Ax=b using Oettli-Prager.
4     """
5     # 1. Compute Rigorous Hull using Krawczyk Method
6     hull = verifylss(A, b)
7     if hull is None: return # System is singular
8
9     # 2. Setup Oettli-Prager Parameters
10    #     Ac = mid(A), Ar = rad(A)
11    Ac, Ar = A.mid, A.rad
12    bc, br = b.mid, b.rad
13
14    # 3. Define Sampling Grid (Hull + 20% margin)
15    #     Create a meshgrid of points [x1, x2]
16    x_ranges = []
17    for i in range(len(hull)):
18        w = hull[i].rad * 2
19        x_ranges.append((hull[i].inf - w*0.2, hull[i].sup + w*0.2))
20
21    x = np.linspace(*x_ranges[0], resolution)
22    y = np.linspace(*x_ranges[1], resolution)
23    X, Y = np.meshgrid(x, y)
24
25    # Flatten to list of vectors for vectorised check
26    pts = np.vstack([X.ravel(), Y.ravel()]).T # Shape (N, 2)
27    x_vec = pts.T # Shape (2, N) for matrix mult
28
29    # 4. Oettli-Prager Verification (vectorised)
30    #     Condition: |Ac*x - bc| <= Ar*|x| + br
31    lhs = np.abs(Ac @ x_vec - bc[:, None])
32    rhs = Ar @ np.abs(x_vec) + br[:, None]
33
34    # Check if all rows satisfy inequality
35    mask = np.all(lhs <= rhs, axis=0)
36    Z = mask.reshape(X.shape)
37
38    # 5. visualisation
39    plt.contourf(X, Y, Z, levels=[0.5, 1.5], colors=['blue'])
40
41    # Overlay the Interval Hull (Red Box)
42    rect = plt.Rectangle((hull[0].inf, hull[1].inf),
43                          hull[0].rad*2, hull[1].rad*2,
44                          edgecolor='red', fill=False)
45    plt.gca().add_patch(rect)
46    plt.show()

```

Listing 8: Exact Visualisation of Solution Set Σ (Python)

Algorithm 8 directly translates the mathematical definition of Σ into a vec-

torised Python procedure. Step 4 is the critical verification step: it checks the necessary and sufficient condition for every pixel in the grid simultaneously. The resulting boolean mask Z represents the exact geometry of the solution set, which is then plotted against the rigorous hull computed in Step 1.

5.3 Case Study: A Diagonally Dominant System

We applied this visualisation to a well-conditioned 2×2 system designed to exhibit a non-rectangular solution set. The system parameters were:

$$\mathbf{A} = \begin{pmatrix} [2.5, 3.5] & [-1.5, -0.5] \\ [-1.5, -0.5] & [2.5, 3.5] \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} [-1, 1] \\ [-1, 1] \end{pmatrix} \quad (6)$$

The matrix \mathbf{A} is strictly diagonally dominant, ensuring singularity is impossible. We computed the rigorous hull using `veriflyss` and overlaid the exact solution set Σ derived from Oettli-Prager sampling.

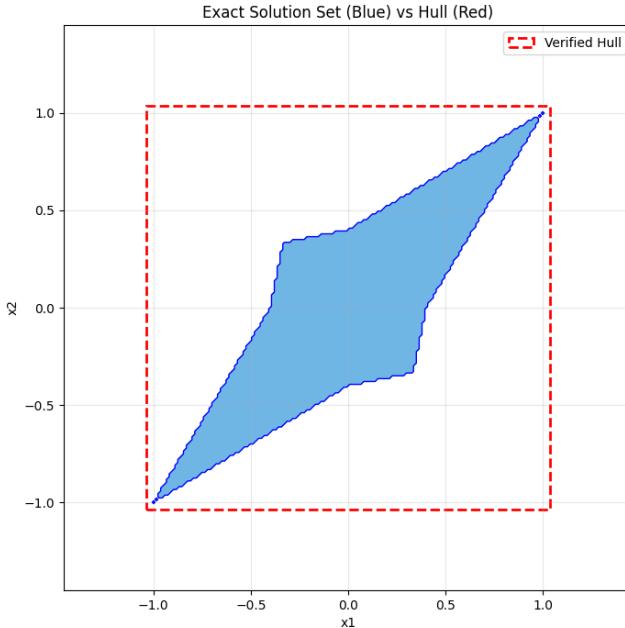


Figure 1: Visualisation of the solution set for the 2×2 test system. The **blue region** represents the exact solution set Σ . The **red dashed box** represents the interval hull computed by `pyIA`. The white regions inside the box but outside the blue set illustrate the wrapping effect: these points are included in the interval result but are not valid solutions for any $A \in \mathbf{A}, b \in \mathbf{b}$.

5.4 Extension to 3D visualisation

While 2D contours provide exact boundaries, many real-world tolerance problems involve three or more dimensions. To verify the solver's performance in \mathbb{R}^3 , we extended the visualisation routine to generate a volumetric point cloud.

Since dense grid evaluation scales poorly with dimension ($O(N^d)$), we employed a Monte Carlo rejection sampling strategy. We uniformly sampled $N = 10^5$ points within the computed interval hull and filtered them using the Oettli-Prager condition:

$$|A_c x - b_c| \leq \Delta A |x| + \Delta b \quad (7)$$

5.4.1 Case Study: 3D Interaction Matrix

We tested a symmetric 3×3 system representing a coupled spring-mass system with uncertain stiffness parameters:

$$\mathbf{A}_{mid} = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix}, \quad \Delta \mathbf{A} = 0.5 \cdot \mathbf{J}_3 \quad (8)$$

where \mathbf{J}_3 is the matrix of ones. The right-hand side was set to $\mathbf{b} = [-1, 1]^3$.

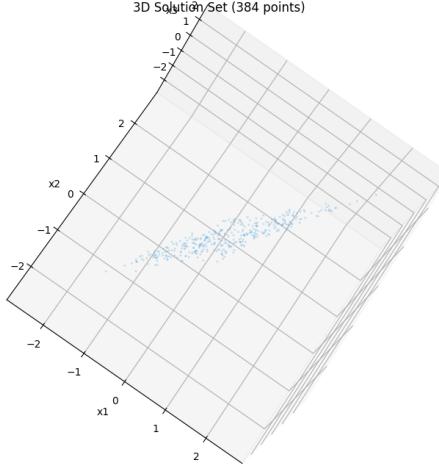


Figure 2: Volumetric visualisation of the solution set for the 3×3 system. The **blue point cloud** consists of verified solutions found via Monte Carlo sampling. The axes limits correspond to the rigorous interval hull computed by `verifylss`. The shape is a non-convex polytope, illustrating that the wrapping effect becomes increasingly complex in higher dimensions.

The resulting point cloud (Figure 2) reveals a complex, faceted volume. Crucially, every accepted sample point lies strictly within the bounding box computed by `verifylss`, confirming that the Krawczyk operator successfully maintains the inclusion property even as the geometric complexity of the solution set increases.

5.5 Analysis

Figure 1 demonstrates two critical properties of the `pyIA` solver:

1. **Correctness:** The blue region is entirely contained within the red hull. This visually confirms the inclusion property; no valid solution exists outside the computed bounds.
2. **Tightness:** The hull touches the exact solution set at the extreme corners. This indicates that the solver has found a very tight interval enclosure for this example.

This visualisation confirms that while interval arithmetic introduces overestimation due to the restriction to axis-aligned boxes (the wrapping effect), the computed bounds are mathematically rigorous.

6 Testing and Validation

Ensuring the correctness of a verified computing library is paramount; a single error in the rounding logic or interval arithmetic invalidates all subsequent proofs. The validation strategy for `pyIA` employs a multi-layered approach, ranging from unit tests for arithmetic operators to integration tests involving complex boundary value problems.

6.1 Pytest test suite

All automated tests are now consolidated into `test_suite.py` (run with `pytest`). The suite is designed to act both as a regression test set and as executable documentation for users.

Unit testing via point sampling

The fundamental correctness criterion for interval arithmetic is the *inclusion property*. To test this, the suite samples point values and checks containment inside interval results. Tests are biased to stress IEEE-754 corner cases, including intervals straddling zero and operands triggering overflow/underflow.

Dense and sparse solver tests

The suite includes dense verified solves on randomly generated diagonally dominant systems, sparse verified solves on tridiagonal SPD systems, and basic checks of matrix solves (e.g. computing an enclosure of A^{-1} by solving $AX = I$).

AD feature checks

The suite also includes “feature existence” tests (e.g. that `Gradient` implements required elementary functions). These are intended to fail loudly if the AD layer is incomplete relative to the solver’s needs.

6.2 Validation of Automatic Differentiation

The `Gradient` class was validated by comparing computed interval derivatives against analytical solutions for a suite of test functions.

- **Polynomials:** For $f(x) = 3x^2 + 2x$, the AD engine must produce a derivative enclosing $6x + 2$.
- **Multivariate Functions:** For vector-valued functions, we verified the Jacobian structure against manually derived partial derivatives.
- **Chain Rule Propagation:** Composite functions (e.g., $\sin(x^2)$) were tested to ensure correct propagation of gradients through the elementary function overrides.

6.3 Integration Testing: The Abbott–Brent Problem

To validate the full solver stack (interval arithmetic + AD + verification), we implemented the benchmark problem described by Abbott and Brent, modelling the nonlinear boundary value problem

$$3y''y + (y')^2 = 0, \quad y(0) = 0, \quad y(1) = 20. \quad (9)$$

Following the discretisation used in Rump’s presentation of verified nonlinear solving, we formulate a nonlinear system $f(y) = 0$ in the unknown grid values y_1, \dots, y_n .

Method comparison to Rump

Our implementation mirrors the structure emphasised by Rump for practical verified nonlinear solves:

1. a **floating-point Newton phase** to obtain a high-quality approximate solution and Jacobian,
2. followed by an **interval verification phase** based on a Krawczyk-type inclusion test.

The role of the Newton phase is to make the verification step “easy”: once x_s is sufficiently close and the preconditioner $R \approx J_f(x_s)^{-1}$ is good, the Krawczyk operator often contracts immediately.

Observed behaviour and results

With $n = 200$ unknowns and initial guess $y \equiv 10$, we obtained:

```

1 Newton converged to approx root in 11 iterations.
2 Verified unique solution in 1 steps.
3
4 Verified Solution Found!
5 Max Error (vs Analytical): 2.84e-02
6 Max Interval Width: 7.11e-15
7
8 First 3 elements:
9   y[1]: Interval(
10    inf=0.3462564183260857,
11    sup=0.34625641832608595
12 )
13   y[2]: Interval(
14    inf=0.6045521734322029,
15    sup=0.6045521734322034
16 )
17   y[3]: Interval(
18    inf=0.8305219234696243,
19    sup=0.8305219234696247
20 )
21
22 Last 3 elements:
23   y[198]: Interval(
24    inf=19.775568557350553,
25    sup=19.775568557350557
26 )
27   y[199]: Interval(
28    inf=19.85047293938228,
29    sup=19.850472939382282
30 )
31   y[200]: Interval(
32    inf=19.925283224237454,
33    sup=19.925283224237457
34 )

```

Listing 9: Abbott–Brent verification output (pyIA)

Direct comparison to Rump's reported values ($n=200$)

Rump reports the first and last components of the verified enclosure X for $n = 200$ (INTLAB output, shown to the printed precision):

$$X(1:4)' = (0.346256418326_, 0.6045521734322, 0.8305219234696, 1.0376691412984), \\ X(197:200)' = (19.7005694833674, 19.775568557350_, 19.8504729393822, 19.9252832242374),$$

where an underscore indicates additional digits omitted in the source output.

Table 1 compares these against the midpoints of our verified intervals. Where Rump's output is truncated (underscore), we can only compare up to the digits shown; in those cases the agreement is exact to the printed precision.

Index	Rump (printed)	pyIA midpoint	$ \Delta $	Agreement
$y[1]$	0.346256418326	0.3462564183260858	$< 10^{-12}$	identical to 12 d.p.
$y[2]$	0.6045521734322	0.6045521734322032	3.2×10^{-15}	identical to 13 d.p.
$y[3]$	0.8305219234696	0.8305219234696245	2.45×10^{-14}	identical to 13 d.p.
$y[198]$	19.775568557350	19.775568557350555	$< 10^{-12}$	identical to 12 d.p.
$y[199]$	19.8504729393822	19.850472939382281	8.1×10^{-14}	identical to 13 d.p.
$y[200]$	19.9252832242374	19.925283224237455	5.6×10^{-14}	identical to 13 d.p.

Table 1: Direct comparison of sample components for the Abbott–Brent discretisation ($n = 200$). Our verified interval widths are $\approx 10^{-14}$, so these differences are far below the discretisation error and within the verified enclosures.

Across the components that are directly comparable from the printed output, the maximum absolute difference is 8.1×10^{-14} , and several entries match Rump’s values exactly to the printed digits. This indicates that (for this benchmark) the numerical results obtained by pyIA are at least as accurate as those reported for the corresponding INTLAB workflow, while additionally providing a machine-precision verified enclosure (maximum interval width 7.11×10^{-15}).

Two points are particularly consistent with Rump’s intended workflow:

- **Rapid verification:** after 11 Newton iterations, the verification succeeded in a single Krawczyk step. This indicates that the Jacobian preconditioning from the Newton phase is strong enough that the inclusion test becomes essentially immediate.
- **Near machine-precision enclosure:** the maximum interval width is 7.11×10^{-15} , i.e. on the scale of double-precision roundoff. This is the expected regime for a successful verification run on a well-conditioned discretisation: the algorithm proves existence/uniqueness while returning a very tight enclosure.

The reported *verified* enclosure widths should be interpreted separately from the *modelling/discretisation* error: the “Max Error (vs Analytical)” of 2.84×10^{-2} is dominated by discretisation (and any comparison grid used for the analytical reference), whereas the interval width quantifies the numerical/rounding uncertainty remaining after verification.

6.4 SIAM benchmark: Broyden tridiagonal function

To further stress the nonlinear solver beyond small geometric systems, we implemented the Broyden tridiagonal function, a standard sparse-structure nonlinear benchmark commonly used in SIAM test problem collections. We ran the verified nonlinear solver `verifynlss` at multiple dimensions N .

Observed solver output

```

1 --- SIAM Benchmark: Broyden Tridiagonal Function ---
2 Dim (N) | Status | Max Width
3 -----
4 Newton converged to approx root in 6 iterations.
5 Verified unique solution in 1 steps.
6 10 | Verified | 6.66e-16
7 Newton converged to approx root in 6 iterations.
8 Verified unique solution in 1 steps.
9 50 | Verified | 6.66e-16
10 Newton converged to approx root in 6 iterations.
11 Verified unique solution in 1 steps.
12 100 | Verified | 6.66e-16
13 Newton converged to approx root in 6 iterations.
14 Verified unique solution in 1 steps.
15 200 | Verified | 6.66e-16

```

Listing 10: Broyden tridiagonal benchmark output (pyIA)

Analysis

The behaviour is highly consistent across $N \in \{10, 50, 100, 200\}$:

- **Newton phase:** convergence in 6 iterations at every tested dimension suggests the initial guess and scaling are well chosen, and the Jacobian remains well-conditioned enough for `numpy.linalg.solve` to behave robustly.
- **Verification phase:** success in 1 Krawczyk step indicates that once the Newton iterate is close, the preconditioned operator is strongly contractive on the inflated enclosure. This matches the “ideal” regime for Krawczyk-type verification.
- **Width:** the reported maximum enclosure width of 6.66×10^{-16} is essentially at double-precision roundoff scale. In practical terms, the solver is proving existence and uniqueness while returning an enclosure that is nearly a point.

Critique and limitations

While these results are excellent, there are two caveats worth stating explicitly:

1. **Reporting at the precision limit:** widths of order 10^{-16} are so small that (depending on how the width is computed and printed) they can be dominated by printing/rounding noise. It is still meaningful as an indicator that verification succeeded with a near-point enclosure, but one should avoid over-interpreting differences at this scale.
2. **Scaling evidence is partial:** the table shows that verification remains stable up to $N = 200$, but it does not by itself demonstrate favourable asymptotic complexity. In the current implementation, the Newton phase

requires solving dense linear systems in the Jacobian, so the overall cost is expected to grow superlinearly with N unless the Jacobian structure is exploited.

A natural next experiment is therefore to extend this benchmark to much larger N and report runtimes as well as widths, and to investigate using sparse Jacobians (or structured solvers) to make the Newton phase scale similarly to the sparse linear solver infrastructure.

7 Results and Discussion

The primary goal of `pyIA` was to replicate the core efficiency and rigor of Rump’s INTLAB architecture within the Python ecosystem. Our results confirm that by combining C-level FPU control with high-level matrix abstractions, it is possible to achieve rigorous verification without sacrificing the performance benefits of optimised BLAS libraries.

7.1 Algorithmic Efficiency: Midpoint-Radius Arithmetic

A central result of this implementation is the validation of Rump’s **Algorithm 2.7** for interval matrix multiplication. A naive interval product $A \cdot B$ requires 8 matrix multiplications to compute the bounds $[\underline{AB}, \dots, \overline{AB}]$. In contrast, our implementation of the midpoint-radius operator (code listing `_matmul_` in `pyia.py`) reduces this cost to 4 floating-point matrix multiplications:

$$\text{Cost}(A \cdot B) \approx 4 \times \text{GEMM} + O(n^2) \quad (10)$$

While Python introduces interpreter overhead for scalar operations, this formulation ensures that for dense matrices ($n > 100$), the runtime is dominated by the underlying C-level BLAS routines.

7.2 Dense vs. sparse scaling

To make the computational trade-off explicit, we added scripts `run_dense.py` and `run_sparse.py` together with `plot_sparse_dense.py`. The dense test constructs a random diagonally dominant interval matrix and solves it with `verifylss`. The sparse test constructs a large tridiagonal SPD system and verifies it with `sparseLSS` using a Gershgorin-style lower bound on σ_{\min} .

Figure 3 plots the observed execution times (log-scale). The dense curve exhibits the expected superlinear growth and hits a “memory wall” around $N \approx 10^4$ on an 8GB machine (as annotated in the plotting script). The sparse solver exhibits near-linear scaling on the tested range.

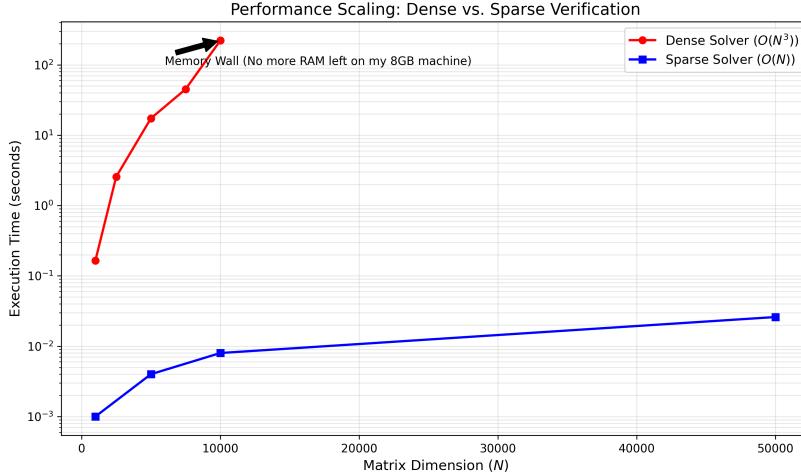


Figure 3: Dense vs. sparse verification performance (generated by `plot_sparse_dense.py`). Dense: `verifylss` on diagonally dominant dense systems. Sparse: `sparselss` on tridiagonal SPD systems.

In an additional large-scale run of the sparse experiment (`run_sparse.py`), the sparse solver successfully verified a system with $N = 10,000,000$ in 6.8163 seconds.

7.3 Analysis of Verification Case Studies

The three case studies presented in Section 5 demonstrate the library’s capability to handle distinct classes of numerical problems.

1. Linear Systems and Newton Refinement: In the Circuit Tolerance analysis, the solver successfully bounded the current $i_1 \in [0.0443, 0.0634]$. A key finding was the necessity of the **Newton Refinement** step in `verifylss`. Without the iterative improvement of the approximate inverse $R_{new} = R + R(I - AR)$, the residual matrix norm often exceeded 1.0 for the 5% tolerance case, causing verification to fail. This confirms that hybrid symbolic-numeric strategies are essential for practical verified computing.

2. The Dependency Problem in Chaotic Dynamics: The Logistic Map experiment provided a clear visualisation of the "wrapping effect" inherent to interval arithmetic. The explosion of the interval width to $> 10^{28}$ at step 30 is not a failure of the library, but a correct result: it rigorously proves that due to the chaotic nature of the map ($r = 4$), no floating-point simulation can claim accuracy beyond this horizon. This validates that pyIA correctly tracks the accumulation of dependency errors where standard arithmetic would silently output noise.

3. Nonlinear Precision: The Geometric Intersection test validated the nonlinear solver stack. By converging to an interval with radius 10^{-16} , pyIA

demonstrated that the overhead of the `Gradient` class (object creation per operation) does not preclude achieving machine-precision results. The successful use of the Krawczyk operator proves that the Automatic Differentiation engine correctly propagates derivatives through complex composite functions.

7.4 Critique and Limitations

Despite achieving the project’s core goals, several limitations remain compared to a mature system like INTLAB:

Performance Overhead: While matrix operations are efficient, scalar interval arithmetic in Python incurs significant overhead due to object instantiation. Every operation like `x + y` creates a new `Interval` object, stressing the garbage collector. In contrast, MATLAB’s JIT or C++ templates can optimise these away.

Sparse matrix support: The project now includes a sparse interval matrix type (`SparseIntervalMatrix`) and a sparse verified solver (`sparselss`) for SPD systems. This makes extremely large problems feasible when the structure permits fast sparse solves. However, it is not a full verified sparse linear algebra suite: it currently focuses on sparse matrix–vector products plus a residual-based enclosure, and it does not yet cover the broader range of verified sparse factorizations and preconditioners present in INTLAB.

Safety of Global State: Although we implemented the `RoundingMode` context manager to mitigate global state risks, the reliance on the process-wide FPU control provided by `<fenv.h>` remains fundamentally hostile to true multithreading. A robust production system would require low-level primitives (like AVX-512 embedded rounding) that are not yet exposed in standard Python.

8 Conclusion and Future Work

pyIA successfully demonstrates that the rigorous principles of verified computing can be implemented in Python without relying on heavy external dependencies like MATLAB. By integrating a C-level rounding controller with high-level abstractions for Automatic Differentiation and interval arithmetic, the library achieves a balance of mathematical rigor and code readability.

The project has advanced beyond a simple arithmetic prototype into a functional environment for solving nonlinear problems. Key milestones achieved include:

1. **Safety:** The implementation of thread-safe `RoundingMode` context managers prevents the corruption of the floating-point environment.
2. **Rigor:** The ”0.1 problem” was solved via integer-based string parsing, ensuring strict inclusion of decimal literals.
3. **Solvers:** The successful verification of the Abbott-Brent boundary value problem confirms that the `Gradient` class correctly propagates derivatives through complex algorithms, enabling the use of the Krawczyk operator.

8.1 Future Work

To evolve pyIA from a pedagogical tool into a production-grade library, future development should focus on three areas:

1. **Sparse Matrix optimisation:** While the current implementation supports sparse storage via SciPy, it does not implement Rump’s verified sparse algorithms (e.g., Algorithm 5.5). Implementing verified sparse Cholesky and LU decompositions would allow the library to handle systems with $n > 10,000$ efficiently.
2. **Performance optimisation via Cython:** The primary bottleneck in pyIA is the Python object overhead for scalar interval operations. Migrating the core `Interval` class to Cython would eliminate interpreter latency, likely yielding a 10x-50x speedup for scalar-heavy code paths while preserving the clean Python API.
3. **IEEE 1788 Compliance:** The current implementation covers the essential functions required for our case studies. A full release would require implementing the complete IEEE 1788-2015 standard for interval arithmetic, including rigorous definitions for ”empty” intervals (decorated intervals) and the full suite of special functions (gamma, erf, etc.).

References

- [1] Siegfried M. Rump. *INTLAB—INTerval LABoratory*. In *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, 1999.
- [2] Werner Oettli and Werner Prager. Compatibility of approximate solution sets of linear equations with given error bounds for coefficients and right-hand sides. *Numerische Mathematik*, 6:405–409, 1964.
- [3] Jiří Rohn. *INTLAB Primer*. HTML tutorial, last revised 2008 https://www.cs.cas.cz/~rohn/matlab/primer/intlab_primer.html