Sirpy Problem 1

The Problem That Hides a Cliff: SIRPY Solves an Equation — and Finds the Edge Nobody Told It About

First in a weekly series. Each week, until SIRPY launches, we post one problem, the exact input we handed the solver, the exact output it returned, and an honest read of what happened. No slides. No hand-waving. Numbers you can check.

Why we're starting here

We could have opened with something that looks hard. Instead we chose something that looks *easy* and quietly isn't — because the gap between "looks easy" and "is easy" is exactly where a solver earns your trust or loses it.

Here is the whole problem:

$$y'' = 2y^3, \qquad y(0) = -\tfrac{1}{10}, \qquad y(9) = -1$$

That's it. A second-order boundary value problem on the interval $[0, 9]$. Two endpoints pinned down, and a nonlinear right-hand side. Nothing about it looks threatening.

It has an exact closed-form solution:

$$y(x) = \frac{1}{x - 10}$$

You can verify it by hand in thirty seconds. Differentiate twice, and $y'' = 2/(x-10)^3 = 2y^3$. Check the endpoints: $y(0) = -1/10$ and $y(9) = 1/(9-10) = -1$. It fits.

Where the cliff is

Look again at that solution: $y = 1/(x - 10)$.

It has a **pole at** $x = 10$ — a point where the function rockets to infinity. And $x = 10$ sits *exactly one unit past the right end of our interval.* On $[0, 9]$ the solution is smooth and perfectly finite, sliding from -0.1 down to -1. But the entire way across, it is leaning toward a cliff just out of frame. At $x = 9$ the function is -1; a single step further and it plunges toward $-\infty$.

This is what makes the problem a genuine test rather than a warm-up. A solver that isn't paying attention can march across $[0, 9]$, feel the solution steepening, and either destabilize or — worse — return a confident, wrong answer without any indication that a singularity is breathing on the interval's neck. The honest behavior is twofold: solve the finite part correctly, *and* notice the cliff.

Exactly what we handed the solver

No reformulation. No conversion to a first-order system. No mesh design. No initial guess. We typed the problem essentially as written:

```
from sirpy import solve_bvp

r = solve_bvp(
    order=2,
    f_str="2*y**3",
    x0=0, x1=9,
    left_bc={0: -0.1, 1: "gamma"},    # y(0) = -0.1; y'(0) unknown
    right_bc={0: -1},                 # y(9) = -1
    iterations=40, partition=16,
    verbose=False,
)
```

The one thing worth explaining is `left_bc={0: -0.1, 1: "gamma"}`. We know $y(0) = -0.1$. We do **not** tell SIRPY the initial slope $y'(0)$ — we mark it unknown with the label `"gamma"` and let the solver recover it from the boundary condition at the far end. That's the whole job of a boundary value problem: find the missing initial data that makes the solution land exactly where it must at $x = 9$.

We never mention the pole. We never hint that a singularity exists near the interval. SIRPY gets the equation and two endpoint values, and nothing else.

Once the boundary value problem is solved, we have the one piece of data we were missing: the initial slope $y'(0)$. With the equation and *both* initial values now in hand, we let SIRPY continue the solution forward as an initial value problem from $x = 0$ — a single expansion,

not the 16 local pieces the BVP used. It is this continuation that reads its own coefficients and reports where the solution's nearest singularity lies. We still never tell it a pole exists; we simply let one expansion run far enough to feel it.

What came back

1. It recovered the missing slope to the last digit

The unknown initial slope for the exact solution is $y'(0) = -1/(0-10)^2 = -0.01$. SIRPY returned:

```
recovered y'(0) = -0.010000000000000
exact     y'(0) = -0.010000000000000
```

Fifteen digits, exact. It found the initial condition we deliberately withheld.

2. It matched the exact solution to machine precision

Across the whole interval, the largest discrepancy between SIRPY's solution and $1/(x-10)$ was:

```
max error vs exact = 1.11e-16
```

That is 10^{-16} — the floor of double-precision arithmetic. There is no meaningful room below this number; it is as close to "exactly right" as a 64-bit machine can represent. Point by point:

x	SIRPY $y(x)$	exact $1/(x-10)$	|error|
0.00	−0.100000000000	−0.100000000000	0.0
1.00	−0.111111111111	−0.111111111111	0.0
2.00	−0.125000000000	−0.125000000000	1.4e−17
3.00	−0.142857142857	−0.142857142857	0.0
4.00	−0.166666666667	−0.166666666667	0.0
5.00	−0.200000000000	−0.200000000000	0.0
6.00	−0.250000000000	−0.250000000000	0.0
7.00	−0.333333333333	−0.333333333333	5.6e−17
8.00	−0.500000000000	−0.500000000000	0.0
8.50	−0.666666666667	−0.666666666667	0.0
9.00	−1.000000000000	−1.000000000000	0.0

Most points are dead-on to twelve printed digits, with errors of zero or a single unit in the last bit.

3. The answer is a function, not a table of points

This is the part that separates SIRPY from a conventional solver, and it deserves emphasis. Most solvers hand you an array: a list of (x, y) pairs, and the quiet hope that whatever lives between them is close enough. SIRPY hands you the solution **as a polynomial you can read, differentiate, and evaluate anywhere.**

Here is the beginning of what it returned, centered at $x = 0$:

$$y(x) = -0.1 - 0.01\,x - 0.001\,x^2 - 0.0001\,x^3 - 10^{-5}x^4 - 10^{-6}x^5 - \cdots$$

Now here is why that's remarkable. The exact Taylor series of $1/(x - 10)$ has coefficients $c_k = -1/10^{\,k+1}$ — that is, $-0.1, -0.01, -0.001, -0.0001, ...$, each ten times smaller than the last. Compare that against what SIRPY actually computed:

```
c[ 0] = -1.000000000000000e-01     exact  -1e-01
c[ 1] = -9.999999999999997e-03     exact  -1e-02
c[ 2] = -1.000000000000000e-03     exact  -1e-03
c[ 3] = -9.999999999999999e-05     exact  -1e-04
c[ 4] = -9.999999999999999e-06     exact  -1e-05
c[ 5] = -1.000000000000000e-06     exact  -1e-06
c[ 6] = -9.999999999999998e-08     exact  -1e-07
c[ 7] = -9.999999999999997e-09     exact  -1e-08
...
```

Coefficient by coefficient, SIRPY reconstructed the exact power series of a function it was never shown. It was given an equation and two boundary values, and it returned the analytic fingerprint of $1/(x - 10)$ to fifteen significant figures.

4. And it found the cliff — on its own

This is the result we care about most. After solving on $[0, 9]$ and continuing the solution forward as a single expansion (as described above), SIRPY reported:

```
detected pole at x = 10.000000000000   (exact: x = 10)
pole order = 1
(nearest-singularity estimate; two independent estimators agree)
```

To twelve decimal places. **SIRPY located the singularity at $x = 10$ — a point outside the interval it was asked about, that we never mentioned — and correctly identified it as a simple (order-1) pole.** It did this by reading the structure of the coefficients it computed: a series whose terms shrink at a steady geometric rate is announcing precisely where its own radius of convergence ends, and SIRPY listens. Two independent estimators inside the solver agreed on the location, which is why it reports the result with high confidence rather than a guess.

That is the difference between a solver that *computes* and one that *understands the function it hands you*. One gives you numbers on $[0, 9]$. The other gives you the numbers, the formula behind them, and a warning that there's a cliff at $x = 10$.

How the accuracy was built

We didn't reach machine precision by magic, and we won't pretend we did. SIRPY has two independent knobs — the **iterations** (the degree of each Taylor piece) and the **partition** (how many subintervals the interval is cut into) — and accuracy improves along either one. The headline result above (10^{-16}) uses both turned up together: iterations = 40 *and* partition = 16. To show that each knob works on its own, the two sweeps below move one at a time, leaving the other lower — so neither sweep alone reaches the headline number, and that's the point: they show the *trend*, not the peak.

Sweep 1 — let SIRPY choose the partition automatically, raise the number of iterations:

iterations	max error
10	4.32e−02
20	1.13e−06
30	2.12e−09
40	3.84e−12

Sweep 2 — hold iterations at 20, raise the number of subintervals:

partition	max error
8	1.13e−06
12	1.46e−08
16	3.94e−10
20	1.81e−11
24	1.89e−12

Two things to read from these tables. First, the error falls fast and *consistently* along both axes — the sign of a method that is genuinely converging to the true solution, not plateauing near it. Second, and this is a point of principle: at `iterations=10` the error is a visible 4×10^{-2}. That is a first pass, and SIRPY does not dress it up as anything more. The accuracy is *earned* by raising resolution, and the solver reports where it stands at every stage rather than claiming a precision it hasn't reached. That honesty is the whole point — a solver you can trust is one that tells you when it isn't done yet.

What this one problem shows

- **You bring the equation as written.** No first-order rewrite, no mesh, no initial guess. SIRPY recovered the missing initial slope itself.
- **The answer is a function.** A readable polynomial you can differentiate and evaluate anywhere — not a grid of points with gaps between them.
- **It matches the truth to machine precision** — 10^{-16} — and reconstructs the exact power series coefficient by coefficient.
- **It sees what it wasn't told.** Without any hint that a singularity existed, it located the pole at $x = 10$ to twelve decimal places and classified it correctly.
- **It's honest about the climb.** Low-resolution passes are labeled as first passes; accuracy is shown being built, not asserted. ## Next week

A problem where the interesting question isn't *how accurately* it's solved — but whether a standard, widely used tool can even **state** it. We think the answer will surprise you.

SIRPY is a differential-equation solver in testing, not yet released — a product of Luminary Quantum Institute, LLC (LQI), developed at the Mitra Institute of Education (MIE). Every number above comes from a single run of the solver; nothing is quoted from memory. if you have a problem you think is difficult, send it our way. We'll run it, tell you exactly what happened, and show you the result. Whether SIRPY solves it directly, hands it to another solver, or reaches a limit, you'll know exactly why.

Figures (attach in this order):

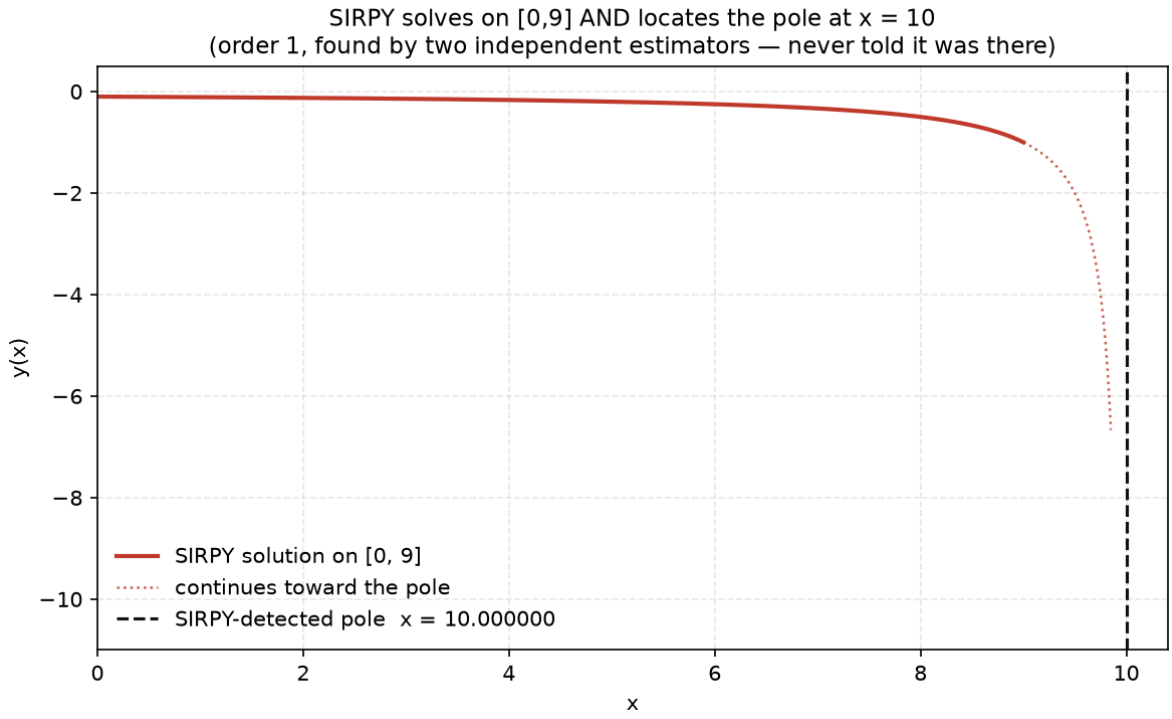

Figure 1: solves on $[0, 9]$ and locates the pole at $x = 10$

SIRPY returns the solution as a polynomial

$y'' = 2y^3, \quad y(0) = -0.1, \quad y(9) = -1 \qquad \text{exact: } y = \dfrac{1}{x - 10}$

$$y(x) = -0.1 - 0.01\,x - 0.001\,x^2 - 10^{-4}x^3 - 10^{-5}x^4 - 10^{-6}x^5 - \cdots$$

The exact Taylor series of 1/(x-10) is $c_k = -1/10^{k+1}$

```
c[0] = -1.000000000000e-01      exact  -1.000000000000e-01
c[1] = -1.000000000000e-02      exact  -1.000000000000e-02
c[2] = -1.000000000000e-03      exact  -1.000000000000e-03
c[3] = -1.000000000000e-04      exact  -1.000000000000e-04
c[4] = -1.000000000000e-05      exact  -1.000000000000e-05
c[5] = -1.000000000000e-06      exact  -1.000000000000e-06
```

recovered to machine precision — SIRPY was never told the answer

Figure 2: the solution as a polynomial, coefficients vs. exact

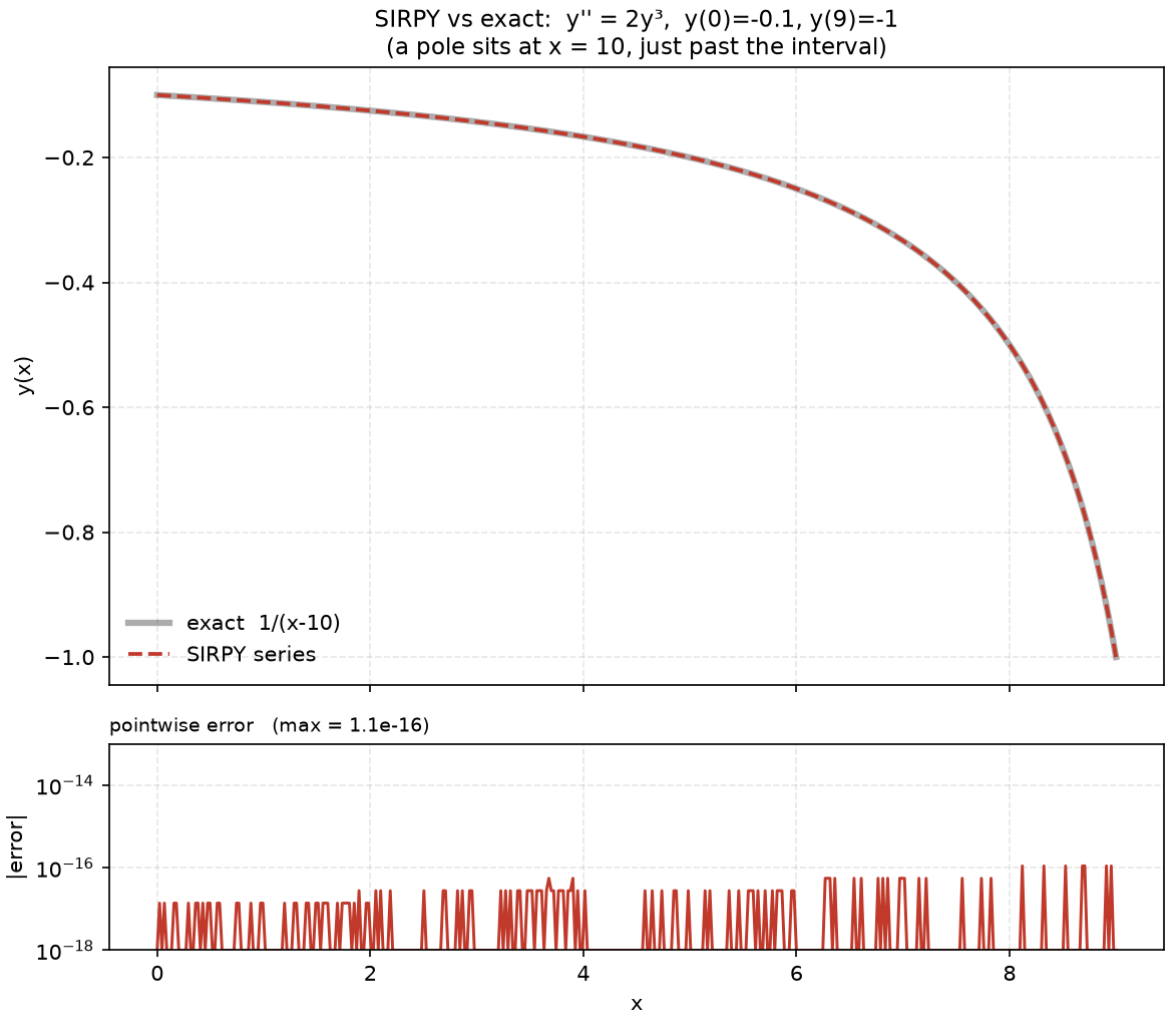

Figure 3: SIRPY vs. exact, with the error floor

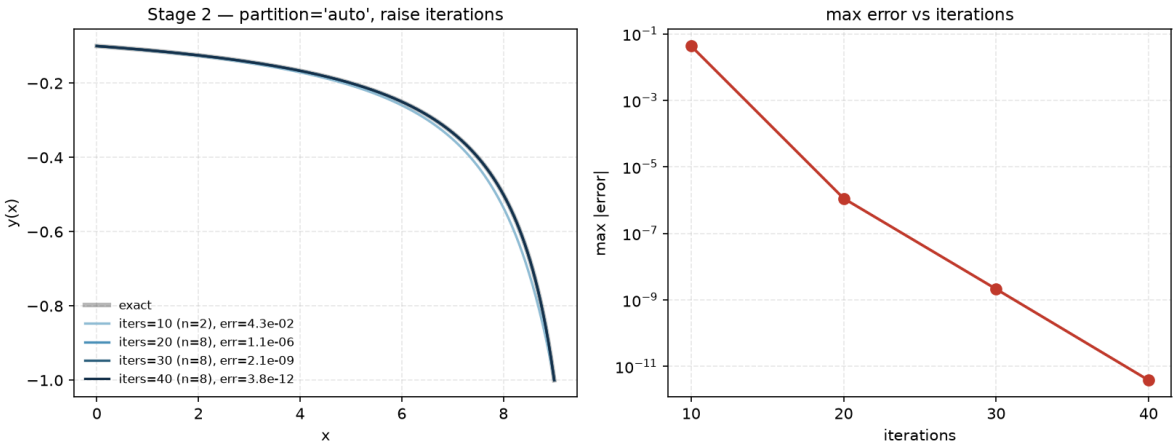

Figure 4: accuracy vs. iterations

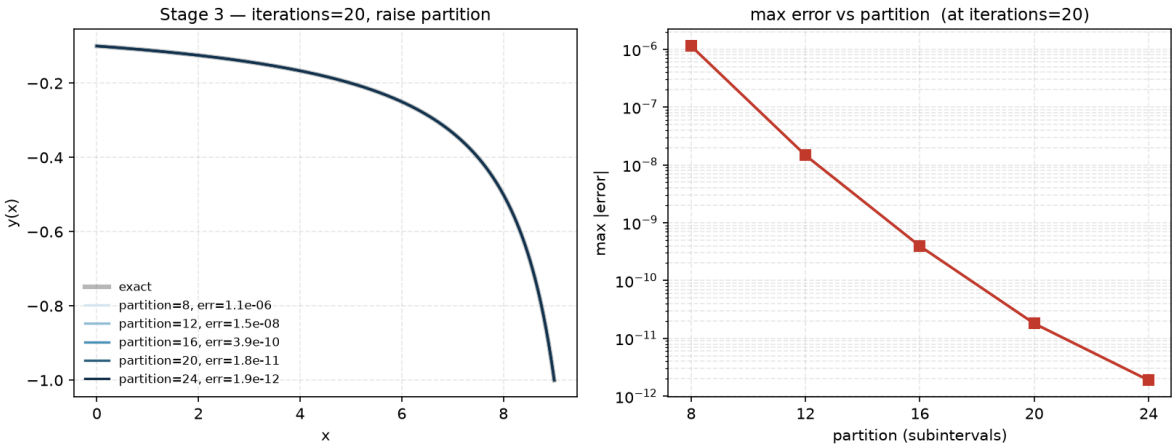

Figure 5: accuracy vs. partition