nilsolve(version 0): A Toolkit for Nonlinear ODEs and Bifurcation Analysis

Nilanjan

1. What is nilsolve?

`nilsolve` is a terminal-first, lab-grade Python library designed for robust numerical analysis of nonlinear dynamical systems. It is built explicitly for research workflows where automation, reproducibility, and failure safety matter more than convenience defaults.

The library is organized into two independent but complementary phases.

2. Core Features (High-Level)

Phase 1: Initial Value Problems (IVP)

Phase 1 solves time-dependent ODE systems of the form

$$\dot{y}(t) = f(t, y, p), \quad y(t_0) = y_0.$$

Key features:

- **Stiffness-aware solver ranking** across RK45, DOP853, BDF, and Radau.
- **Automatic solver selection** under wall-time and pilot budgets.
- **Reproducible solves** producing CSV, PNG (optional), and JSON reports.
- **Parameter overrides from CLI** without code modification.
- **1D and 2D parameter scans** with decision reuse.
- **Failure-safe execution**: individual failures are recorded, not fatal.

Typical use cases:

- stiff chemical kinetics
- slow–fast biological models
- epidemiological models
- automated parameter sweeps

Phase 2: Equilibrium Continuation and Stability

Phase 2 analyzes steady states defined implicitly by

$$g(y, \mu) = 0,$$

where μ is a continuation parameter.

Key features:

- Equilibrium continuation using predictor–corrector methods.
- Newton correction with optional analytic Jacobians.
- Linear stability analysis via Jacobian eigenvalues.
- Localized Hopf detection (`HOPF_LOC`).
- Branch-level CSV output, report JSON, and optional plots.

Typical use cases:

- bifurcation diagrams
- stability boundaries
- oscillatory onset detection

3. Solver Ranking and Decision Philosophy

Solver ranking in `nilsolve` is not accuracy-first. It is designed for *automation robustness.*

During ranking, each candidate solver is evaluated on:

- successful completion under a pilot wall-time cap
- numerical stability (no blow-up, no excessive step rejection)
- runtime and function evaluation cost
- tolerance robustness

The resulting decision answers the question:

Which solver is most reliable and efficient for repeated automated use?

This explains why implicit solvers (BDF, Radau) often rank highest, even for mildly non-stiff systems. Explicit solvers are retained for regimes where stiffness is absent or accuracy requirements dominate.

Once selected, the solver decision can be reused across scans, ensuring reproducibility and avoiding re-ranking overhead.

4. Installation and Import Model

Installation

From PyPI:

```
pip install nilsolve
pip install "nilsolve[plot]" # optional plotting
```

Editable install (recommended for development):

```
source .venv/bin/activate
pip install -e .
```

Import Philosophy

User-defined systems live *outside* the library. They do not import internal solver logic.

The only required import for Phase 1 is:

```
from nilsolve import ProblemSpec
```

For Phase 2:

```
from nilsolve.core.cont.spec import EquilibriumSpec
```

5. Phase 1 Example: Simple ODE System

Defining a system

Create a file `mymodel_simple.py`:

```
import numpy as np
from nilsolve import ProblemSpec

state_names = ["x"]

def build_problem() -> ProblemSpec:
   y0 = np.array([1.0])
   t0, t1 = 0.0, 10.0

   def rhs(t, y, p):
      return np.array([-y[0]])

   return ProblemSpec(
      rhs=rhs,
      t_span=(t0, t1),
      y0=y0,
      params={},
      state_names=state_names,
   )
```

Running Phase 1 from the terminal

Rank solvers:

```
nilsolve rank --system mymodel_simple.py
```

Solve and write outputs:

```
nilsolve solve --system mymodel_simple.py --out-prefix outputs/simple
```

Generated artifacts:

```
outputs/simple_solution.csv
outputs/simple_report.json
outputs/simple_solution.png (if plotting enabled)
```

6. Phase 2 Example: Equilibrium Continuation

Defining an equilibrium system

Create `mymodel_eq.py`:

```
import numpy as np
from nilsolve.core.cont.spec import EquilibriumSpec

state_names = ["x"]

def build_equilibrium_problem() -> EquilibriumSpec:
   params = {"mu": 0.0}
   y_guess = np.array([0.0])

   def f(y, p):
      return np.array([y[0] - p["mu"]])

   def jac(y, p):
      return np.array([[1.0]])

   return EquilibriumSpec(
      f=f,
      y_guess=y_guess,
      params=params,
      param_name="mu",
      jac=jac,
      state_names=state_names,
   )
```

Running Phase 2

```
nilsolve cont eq \
  --system mymodel_eq.py \
  --param mu \
  --mu-start -2 --mu-stop 2 \
  --ds 0.1 \
  --out-prefix outputs/eq
```

Outputs:

```
outputs/eq_branch.csv
outputs/eq_report.json
outputs/eq_branch.png (optional)
```

7. Summary

`nilsolve` provides:

- robust IVP solving with automated solver selection
- reproducible parameter scans
- equilibrium continuation with stability diagnostics
- terminal-first workflows suitable for lab automation

Phase 1 and Phase 2 are independent, but can be applied to the same system to connect time-domain dynamics with bifurcation structure.