

Class-based ODE solvers and event detection in SciPy

David R Hagen¹ and Nikolay Mayorov²

¹Applied BioMath, Concord, Massachusetts ²WayRay, Russia

Abstract

In SciPy 1.0, a new interface for solving ODE initial value problems was added to `scipy.integrate`. The `solve_ivp` function takes a initial vector of states, the time derivative of those states, and various options and returns the solution over time, like the `odeint` function it is intended to replace. The new interface provides a few features not previously available, such as event detection and dense solutions, along with various usability improvements. The `OdeSolver` class, an abstract base class which all solvers in the new framework inherit, is an extensible replacement for the `ode` class. This pure-Python base class makes it easy to add and use new solvers, whether in a user's code, an external library, or SciPy itself. The solver classes can be used by themselves to walk through the integration one step at a time if specialized behavior not provided by `solve_ivp` is desired. Currently implemented are the RK23, RK45, Radau, BDF, and LSODA solvers.

Background

An initial value problem (IVP) is a combination of a system of explicit first-order ordinary differential equations (ODEs) and a vector of initial conditions (ICs).

$$\frac{dy}{dt}(t) = f(t, y(t))$$
$$y(t_0) = y_0$$

Here, y is the vector of states. Its value at t other than t_0 can be determined by numeric integration of the system of ODEs f , starting at (t_0, y_0) and taking small steps in t , using dy/dt to project new values of y at each step, until all desired values of $y(t)$ are obtained.

Problems of this form are central to many fields of science and engineering, usually as a kind of simulation. A system can be modeled as an IVP if its internal state can be represented as a set of real numbers and the change in that state can be written as a function of the current state. This includes proteins transformed by reactions in systems biology, planets moved by gravity in astronomy, and prey eaten by predators in ecology.

Many numeric integration methods have been written. Which method is best depends on the structure of the system of ODEs and the desired accuracy, among many other things. When working on a new problem, the user often tests the methods available in his platform to find one that works efficiently and reliably.

Previous SciPy integrators

SciPy has included numeric integrators for IVPs since time immemorial via the `scipy.integrate.odeint` function, which produces the solution to the IVP over a given time interval, and the `scipy.integrate.ode` class, which provides an interface for taking one step at a time through the solution. By wrapping some C and Fortran libraries, a basic ODE interface was obtained. This interface has a number of warts and limitations that the new interface seeks to address:

- The `odeint` function wraps the LSODA Fortran library alone and cannot be used with any other method.
- By default, the signature for the ODE system is $f(y, t)$, whose arguments are opposite the convention in this field.
- It is not straightforward to implement additional algorithms for the `ode` class beyond the five for which it was designed.
- Three of the five methods are not reentrant; if used in a multithreaded environment, undefined behavior will result.
- There is no event detection (the ability to record when given functions of the state cross zero, from above or from below, and optionally terminating the integration early upon detection).
- There is no possibility to obtain a dense solution (an object which can provide an accurate value of y at any t , not just predefined t , by storing the intermediate polynomials).

Porcelain

Most users will interact with the new solvers through the `solve_ivp` convenience function. This function internally handles the standardization of inputs, stepping of the solver, storing of the solution, detection of events, and reporting of status. Users who do not need to implement their own solvers or do unusual procedures at each step of the solver will likely only ever use `solve_ivp`. `solve_ivp` is intended to replace `odeint` in SciPy as the standard IVP solving function.

- ```
solve_ivp(fun, t_span, y0, method='RK45', t_eval=None,
dense_output=False, events=None, vectorized=False,
**options) -> OdeResult
```
- `fun: (t: float, y: array_like) -> array_like` System of ODEs.
  - `t_span: Tuple[float, float]` Integration interval  $(t_0, t_f)$ . The solver starts with  $t=t_0$  and ends at  $t=t_f$ .
  - `y0: array_like` Initial conditions.
  - `method: str | Type[OdeSolver]` If a `str`, the solver is looked up by name among the built-in solvers. If a subclass of `OdeSolver` (either a built-in one or a user-defined one), it is used as the solver.
  - `t_eval: Optional[array_like]` Values of  $t$  at which to store the solution, must be sorted and lie within `t_span`. If `None`, the solution at each step of the solver is stored.
  - `events: Optional[List[(t: float, y: array_like) -> float]]` or single callable Either a single event function or a list of event functions. Each event function must be a continuous function of  $t$ . If the sign of the event function changes in a single step of the solver, a root finding algorithm is applied to find the precise value of  $t$  where `event(t, y(t)) == 0`. Each event may have the following attributes:
    - `terminal: bool` If `True`, the integration terminates when this event is detected. The default is `False`.
    - `direction: bool` If positive, event will only trigger going negative to positive. If negative, event will only trigger going positive to negative. If 0, event will trigger in either direction.To create an event function (a callable with attributes), the user can either make a class with `__call__` implemented or directly assign attributes to any Python function (e.g. `event.terminal = True`).
  - `vectorized: bool` If `True`, then `fun` accepts multiple time points at once. This speeds up certain operations. The default is `False`.

The following options are accepted by one or more built in methods.

- `first_step: Optional[float]` Initial step size. Default is determined by the method.
- `max_step: float` Maximum allowed step size. Default is `inf`.
- `rtol, atol: float` Relative and absolute tolerance.
- `jac: array_like | (t: float, y: ndarray) -> ndarray | None` Jacobian, the derivative of `fun` with respect to  $y$ . If `None`, the Jacobian is computed via finite differences. Used by Radau, BDF, and LSODA methods.
- `jac_sparsity: array_like` Sparsity of the Jacobian. This only affects the finite difference `jac`. Used by Radau and BDF methods.
- `lband, uband: int | None` Bandwidth of the Jacobian. Used by LSODA method.
- `min_step: float` The minimum allowed step size. Default is 0. Used by LSODA method.

An object of type `OdeResult` is returned. This is simply a bunch object that holds the solution and diagnostic information.

- `.t: ndarray` The times at which the solution was obtained.
- `.y: ndarray` The solution at  $t$ .
- `.sol: OdeSolution | None` The dense solution, which is callable at any time within `t_span`.
- `t_events: List[ndarray] | None` The times of detected events.
- `nfev: int` Number of function evaluations.
- `njev: int` Number of Jacobian evaluations.
- `nlu: int` Number of LU decompositions
- `status: int` Reason integration stopped. -1: a step failed. 0: reached the end of `t_span`. 1: a terminal event occurred.
- `message: str` human-readable reason for step failure.
- `success: bool` `True` if reached end of interval or terminal event detected. `False` if step failed.

## Plumbing

The internals of the new solvers revolve around the `OdeSolver` abstract base class. The central idea is that each ODE solver method is a subclass of `OdeSolver`. Each integration is a new instance of the appropriate class, initialized with the ODE function, the initial conditions, and other parameters. The solver can be advanced one step at a time, each step mutating the internal state of the solver. `OdeSolver` is intended to replace `ode` in SciPy as the class-based IVP structure.

### Constructor

```
OdeSolver(fun, t0, y0, t_bound, *, vectorized, **options)
```

Every `OdeSolver` has this signature as its constructor. User-created classes must follow this if they want to be usable by `solve_ivp`. Every solver takes solver-specific options. It raises a warning if it receives options it does not understand. A warning strikes a balance between raising an error (which would make it more difficult to try different solvers) and silence (which could confuse the user when options were not obeyed).

### Public attributes/properties/methods

Clients of a solver can expect that these attributes, properties, and methods are present on all instances of `OdeSolver`.

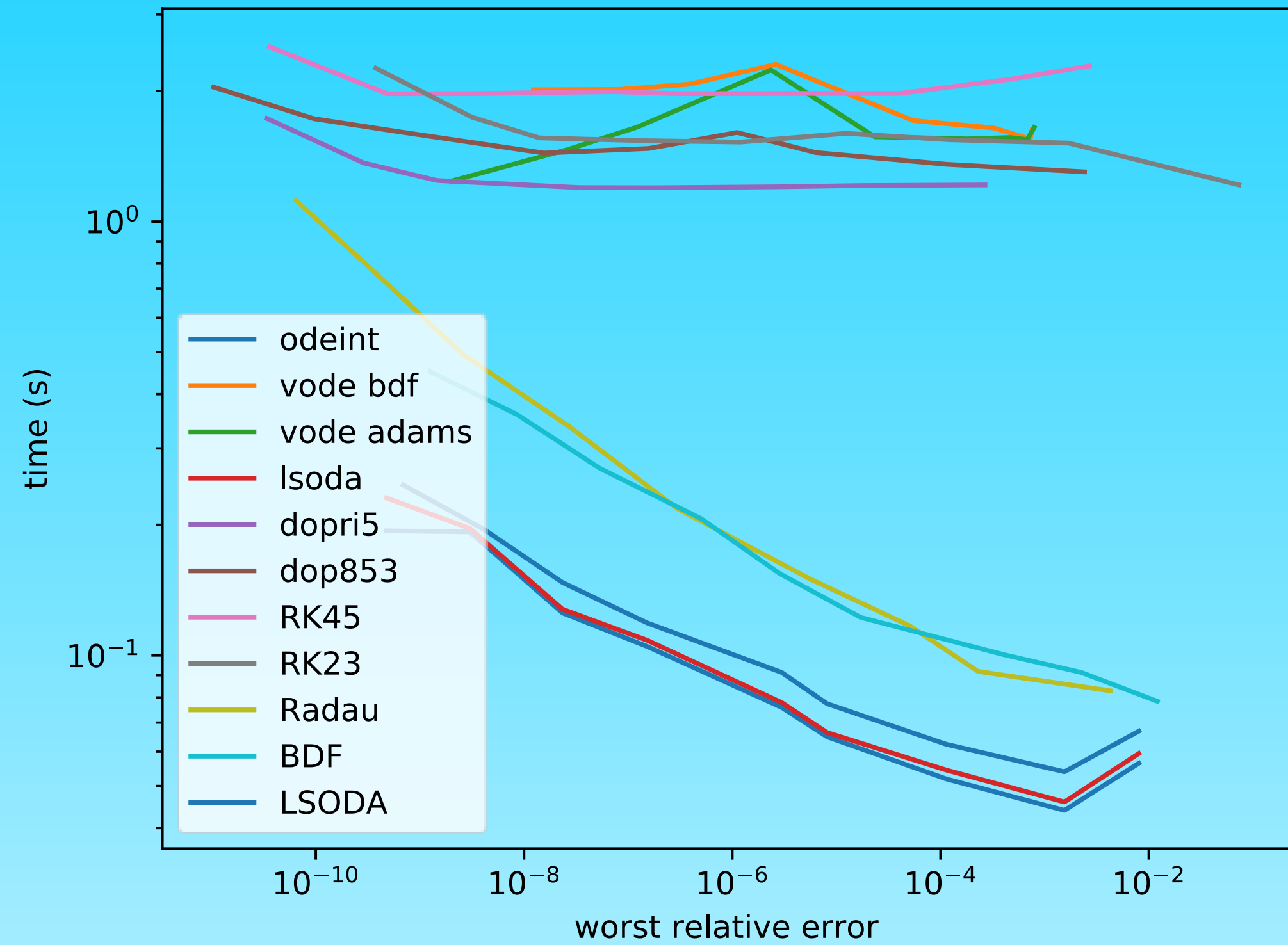
- `.t: float` Current time of solver.
- `.y: ndarray` Current state vector of solver.
- `.t_old: Optional[float]` Previous time of solver. `None` if before first step.
- `.step_size: Optional[float]` Absolute difference between  $t$  and `t_old`. `None` if before first step.
- `.step()` Advances the solver by one step.
- `.dense_output()` -> `DenseOutput` Returns a callable that, given a value between `t_old` and  $t$ , will return the state at that value using interpolation.

### Abstract methods

New subclass of `OdeSolver` need to implement only these two methods.

- `._step_impl()` The actual implementation of a single step.  $t$ ,  $y$ , and any other internal state is updated here.
- `._dense_output_impl()` -> `DenseOutput` The actual implementation of the interpolant for the previous step. Each solver has a specific subclass of `DenseOutput`, depending on how interpolation is done with that solver.

## Performance



The figure above is an example comparison of all ODE solvers in SciPy. This comparison was made with a stiff system having 32 states (specifically the EGF-NGF model from systems biology). The three lines at the bottom are `odeint`, the `lsoda` method of `ode`, and the LSODA method of `solve_ivp`. Unsurprisingly, they have similar performance because they all wrap the same algorithm. There is a mild performance penalty to using the new wrapper written in Python. The two lines in the middle are the new stiff solvers, Radau and BDF. Future work will investigate if cythonizing the Python classes improves the performance of the new solvers. The cluster of lines at the top are the non-stiff solvers, all of which perform very poorly on this stiff problem.