idesolver Documentation

Release 1.1.0

Joshua T Karpel

Nov 19, 2020

CONTENTS

IDESolver is a package that provides an interface for solving real- or complex-valued integro-differential equations (IDEs) of the form

$$\frac{dy}{dx} = c(y, x) + d(x) \int_{\alpha(x)}^{\beta(x)} k(x, s)\, F(y(s))\, ds,$$
$$x \in [a, b], \quad y(a) = y_0.$$

Integro-differential equations appear in many contexts, particularly when trying to describe a system whose current behavior depends on its own history. The IDESolver is an iterative solver, which means it generates successive approximations to the exact solution, using each approximation to generate the next (hopefully better) one. The algorithm is based on a scheme devised by Gelmi and Jorquera.

If you use IDESolver in your work, please consider citing it.

Quickstart A brief tutorial in using IDESolver.

Manual Details about the implementation of IDESolver. Includes information about running the test suite.

API Detailed documentation for IDESolver's API.

Frequently Asked Questions These are questions are asked, sometimes frequently.

Change Log Change logs going back to the initial release.

CHAPTER

ONE

QUICKSTART

Suppose we want to solve the integro-differential equation (IDE)

$$\frac{dy}{dx} = y(x) - \frac{x}{2} + \frac{1}{1+x} - \ln(1+x) + \frac{1}{(\ln(2))^2}\int_0^1 \frac{x}{1+s}\, y(s)\, ds,$$
$$x \in [0,1], \quad y(0) = 0.$$

The analytic solution to this IDE is $y(x) = \ln(1+x)$. We'll find a numerical solution using IDESolver and compare it to the analytic solution.

The very first thing we need to do is install IDESolver. You'll want to install it via *pip* (*pip install idesolver*) into a virtual environment.

Now we can create an instance of `IDESolver`, passing it information about the IDE that we want to solve. The format is

$$\frac{dy}{dx} = c(y,x) + d(x)\int_{\alpha(x)}^{\beta(x)} k(x,s)\, F(y(s))\, ds,$$
$$x \in [a,b], \quad y(a) = y_0.$$

so we have

$$\begin{aligned} a &= 0 \\ b &= 1 \\ y(a) &= 0 \end{aligned}$$

$$\begin{aligned} c(x,y) &= y(x) - \frac{x}{2} + \frac{1}{1+x} - \ln(1+x) \\ d(x) &= \frac{1}{(\ln(2))^2} \\ k(x,s) &= \frac{x}{1+s} \\ f(s) &= y(s) \end{aligned}$$

$$\begin{aligned} \alpha(x) &= 0 \\ \beta(x) &= 1. \end{aligned}$$

In code, that looks like (using `lambda` functions for simplicity):

```
import numpy as np
```

(continues on next page)

(continued from previous page)

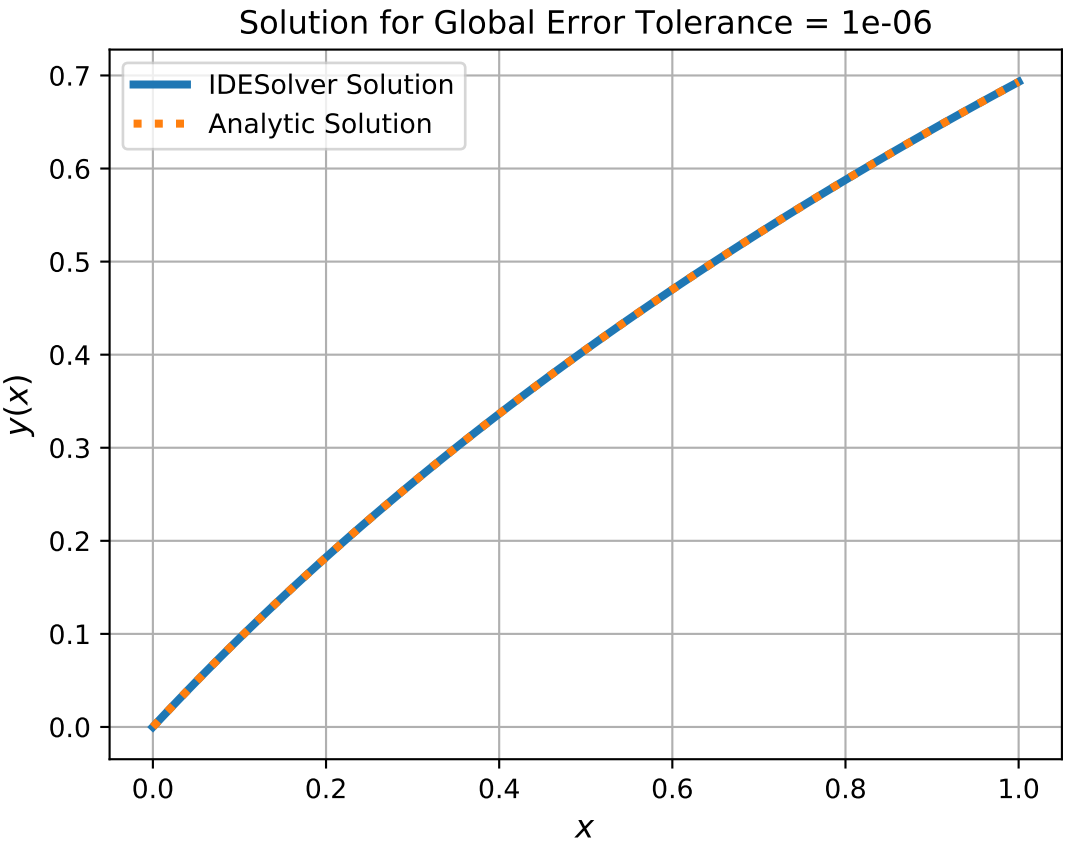
```
from idesolver import IDESolver

solver = IDESolver(
    x = np.linspace(0, 1, 100),
    y_0 = 0,
    c = lambda x, y: y - (.5 * x) + (1 / (1 + x)) - np.log(1 + x),
    d = lambda x: 1 / (np.log(2)) ** 2,
    k = lambda x, s: x / (1 + s),
    f = lambda y: y,
    lower_bound = lambda x: 0,
    upper_bound = lambda x: 1,
)
```

To run the solver, we call the `solve()` method:

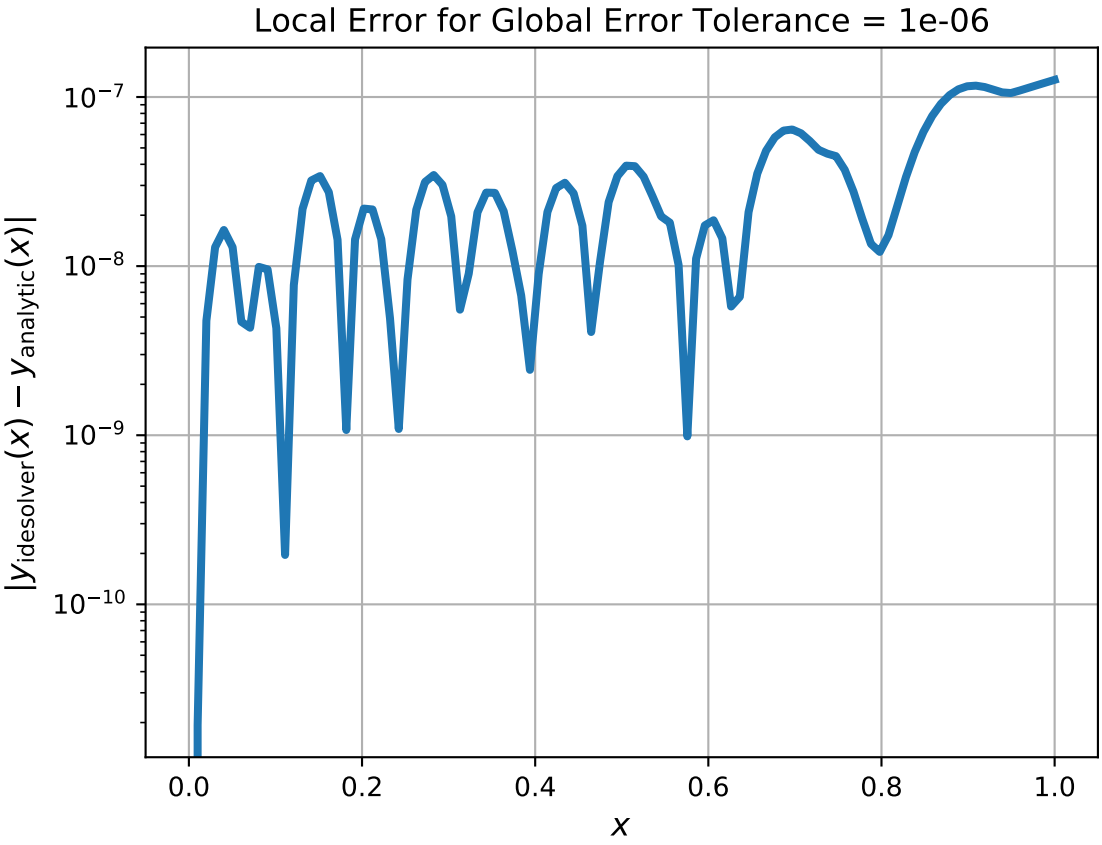
```
solver.solve()

solver.x  # whatever we passed in for x
solver.y  # the solution y(x)
```

The default global error tolerance is 10^{-6}, with no maximum number of iterations. For this IDE the algorithm converges in 40 iterations, resulting in a solution that closely approximates the analytic solution, as seen below.

```
import matplotlib.pyplot as plt

fig = plt.figure(dpi = 600)
ax = fig.add_subplot(111)

exact = np.log(1 + solver.x)

ax.plot(solver.x, solver.y, label = 'IDESolver Solution', linestyle = '-', linewidth␣
↪= 3)
ax.plot(solver.x, exact, label = 'Analytic Solution', linestyle = ':', linewidth = 3)

ax.legend(loc = 'best')
ax.grid(True)

ax.set_title(f'Solution for Global Error Tolerance = {solver.global_error_tolerance}')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y(x)$')

plt.show()
```




```
fig = plt.figure(dpi = 600)
ax = fig.add_subplot(111)

error = np.abs(solver.y - exact)

ax.plot(solver.x, error, linewidth = 3)

ax.set_yscale('log')
ax.grid(True)

ax.set_title(f'Local Error for Global Error Tolerance = {solver.global_error_
↪tolerance}')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$\left| y_{\mathrm{idesolver}}(x) - y_{\mathrm{analytic}}(x) \right|$
↪')

plt.show()
```


Local Error for Global Error Tolerance = 1e-06
10^{-7}
10^{-8}
10^{-9}
10^{-10}
$|y_{\text{idesolver}}(x) - y_{\text{analytic}}(x)|$
0.0
0.2
0.4
0.6
0.8
1.0
x

CHAPTER

TWO

MANUAL

IDESolver implements an iterative algorithm from this paper for solving general IDEs. The algorithm requires an ODE integrator and a quadrature integrator internally. IDESolver uses `scipy.integrate.solve_ivp()` as the ODE integrator. The quadrature integrator is either `scipy.integrate.quad()` or *complex_quad()*, a thin wrapper over `scipy.integrate.quad()` which handles splitting the real and imaginary parts of the integral.

2.1 The Algorithm

We want to find an approximate solution to

$$\frac{dy}{dx} = c(y, x) + d(x) \int_{\alpha(x)}^{\beta(x)} k(x, s)\, F(y(s))\, ds,$$
$$x \in [a, b], \quad y(a) = y_0.$$

The algorithm begins by creating an initial guess for y by using an ODE solver on

$$\frac{dy}{dx} = c(y, x)$$

Since there's no integral on the right-hand-side, standard ODE solvers can handle it easily. Call this guess $y^{(0)}$. We can then produce a better guess by seeing what we would get with the original IDE, but replacing y on the right-hand-side by $y^{(0)}$:

$$\frac{dy^{(1/2)}}{dx} = c(y^{(0)}, x) + d(x) \int_{\alpha(x)}^{\beta(x)} k(x, s)\, F(y^{(0)}(s))\, ds$$

Again, this is just an ODE, because $y^{(1/2)}$ does not appear on the right. At this point in the algorithm we check the global error between $y^{(0)}$ and $y^{(1/2)}$. If it's smaller than the tolerance, we stop iterating and take $y^{(1/2)}$ to be the solution. If it's larger than the tolerance, the iteration continues. To be conservative and to make sure we don't over-correct, we'll combine $y^{(1/2)}$ with $y^{(0)}$.

$$y^{(1)} = \alpha y^{(0)} + (1 - \alpha) y^{(1/2)}$$

The process then repeats: solve the IDE-turned-ODE with $y^{(1)}$ on the right-hand-side, see how different it is, maybe make a new guess, etc.

2.2 Stopping Conditions

IDESolver can operate in three modes: either a nonzero global error tolerance should be given, or a maximum number of iterations should be given, or both should be given. Nonzero global error tolerance is the standard mode, described in *The Algorithm*. If a maximum number of iterations is given with zero global error tolerance, the algorithm will iterate that many times and then stop. If both are given, the algorithm terminates if either condition is met.

2.3 Global Error Estimate

The default global error estimate G between two possible solutions y_1 and y_2 is

$$G = \sqrt{\sum_{x_i} |y_1(x_i) - y_2(x_i)|}$$

A different global error estimator can be passed in the constructor as the argument *global_error_function*.

2.4 Test Suite

First, get the entire IDESolver repository via `git clone https://github.com/JoshKarpel/idesolver.git`. Running the test suite requires some additional Python packages: run `pip install -r requirements-dev.txt` from the repository root to install them. Once installed, you can run the test suite by running `pytest` from the repository root.

CHAPTER

THREE

API

class idesolver.**IDESolver**(*x: numpy.ndarray, y_0: Union[float, numpy.float64, complex, numpy.complex128, numpy.ndarray, list], c: Optional[Callable] = None, d: Optional[Callable] = None, k: Optional[Callable] = None, f: Optional[Callable] = None, lower_bound: Optional[Callable] = None, upper_bound: Optional[Callable] = None, global_error_tolerance: float = 1e-06, max_iterations: Optional[int] = None, ode_method: str = 'RK45', ode_atol: float = 1e-08, ode_rtol: float = 1e-08, int_atol: float = 1e-08, int_rtol: float = 1e-08, interpolation_kind: str = 'cubic', smoothing_factor: float = 0.5, store_intermediate_y: bool = False, global_error_function: Callable = <function global_error>*)

A class that handles solving an integro-differential equation of the form

$$\frac{dy}{dx} = c(y, x) + d(x) \int_{\alpha(x)}^{\beta(x)} k(x, s)\, F(y(s))\, ds,$$
$$x \in [a, b], \quad y(a) = y_0.$$

x
The positions where the solution is calculated (i.e., where y is evaluated).

Type numpy.ndarray

y
The solution $y(x)$. None until *IDESolver.solve()* is finished.

Type numpy.ndarray

global_error
The final global error estimate. None until *IDESolver.solve()* is finished.

Type float

iteration
The current iteration. None until *IDESolver.solve()* starts.

Type int

y_intermediate
The intermediate solutions. Only exists if store_intermediate_y is True.

Parameters

- **x** (numpy.ndarray) – The array of x values to find the solution $y(x)$ at. Generally something like numpy.linspace(a, b, num_pts).
- **y_0** (float or complex or numpy.ndarray) – The initial condition, $y_0 = y(a)$ (can be multidimensional).

- **c** – The function $c(y, x)$. Defaults to $c(y, x) = 0$.
- **d** – The function $d(x)$. Defaults to $d(x) = 1$.
- **k** – The kernel function $k(x, s)$. Defaults to $k(x, s) = 1$.
- **f** – The function $F(y)$. Defaults to $f(y) = 0$.
- **lower_bound** – The lower bound function $\alpha(x)$. Defaults to the first element of `x`.
- **upper_bound** – The upper bound function $\beta(x)$. Defaults to the last element of `x`.
- **global_error_tolerance** (`float`) – The algorithm will continue until the global errors goes below this or uses more than *max_iterations* iterations. If `None`, the algorithm continues until hitting *max_iterations*.
- **max_iterations** (`int`) – The maximum number of iterations to use. If `None`, iteration will not stop unless the *global_error_tolerance* is satisfied. Defaults to `None`.
- **ode_method** (`str`) – The ODE solution method to use. As the *method* option of `scipy.integrate.solve_ivp()`. Defaults to `'RK45'`, which is good for non-stiff systems.
- **ode_atol** (`float`) – The absolute tolerance for the ODE solver. As the *atol* argument of `scipy.integrate.solve_ivp()`.
- **ode_rtol** (`float`) – The relative tolerance for the ODE solver. As the *rtol* argument of `scipy.integrate.solve_ivp()`.
- **int_atol** (`float`) – The absolute tolerance for the integration routine. As the *epsabs* argument of `scipy.integrate.quad()`.
- **int_rtol** (`float`) – The relative tolerance for the integration routine. As the *epsrel* argument of `scipy.integrate.quad()`.
- **interpolation_kind** (`str`) – The type of interpolation to use. As the *kind* argument of `scipy.interpolate.interp1d`. Defaults to `'cubic'`.
- **smoothing_factor** (`float`) – The smoothing factor used to combine the current guess with the new guess at each iteration. Defaults to `0.5`.
- **store_intermediate_y** (`bool`) – If `True`, the intermediate guesses for $y(x)$ at each iteration will be stored in the attribute *y_intermediate*.
- **global_error_function** – The function to use to calculate the global error. Defaults to *`global_error()`*.

solve(*callback: Optional[Callable] = None*) → numpy.ndarray

Compute the solution to the IDE.

Will emit a warning message if the global error increases on an iteration. This does not necessarily mean that the algorithm is not converging, but may indicate that it's having problems.

Will emit a warning message if the maximum number of iterations is used without reaching the global error tolerance.

Parameters **callback** – A function to call after each iteration. The function is passed the *`IDESolver`* instance, the current y guess, and the current global error.

Returns The solution to the IDE (i.e., $y(x)$).

Return type `numpy.ndarray`

idesolver.**global_error**(*y1: numpy.ndarray*, *y2: numpy.ndarray*) → float

The default global error function.

The estimate is the square root of the sum of squared differences between *y1* and *y2*.

Parameters

- **y1** (numpy.ndarray) – A guess of the solution.
- **y2** (numpy.ndarray) – Another guess of the solution.

Returns **error** – The global error estimate between *y1* and *y2*.

Return type float

idesolver.**complex_quad**(*integrand: Callable*, *lower_bound: float*, *upper_bound: float*, ***kwargs) -> (<class 'complex'>, <class 'float'>, <class 'float'>, <class 'tuple'>, <class 'tuple'>)*

A thin wrapper over scipy.integrate.quad() that handles splitting the real and complex parts of the integral and recombining them. Keyword arguments are passed to both of the internal quad calls.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS

4.1 How do I install IDESolver?

Installing IDESolver is easy, using pip:

```
$ pip install idesolver
```

4.2 Can I pickle an `IDESolver` instance?

Yes, with one caveat. You'll need to define the callables somewhere that Python can find them in the global namespace (i.e., top-level functions in a module, methods in a top-level class, etc.).

4.3 Can I parallelize IDESolver over multiple cores?

Not directly - the iterative algorithm is serial by nature. However, if you have lots of IDEs to solve, you can farm them out to individual cores using Python's `multiprocessing` module (multithreading won't provide any advantage). Here's an example of using a `multiprocessing.Pool` to solve several IDEs in parallel:

```
import multiprocessing
import numpy as np
from idesolver import IDESolver


def run(solver):
    solver.solve()

    return solver


def c(x, y):
    return y - (.5 * x) + (1 / (1 + x)) - np.log(1 + x)


def d(x):
    return 1 / (np.log(2)) ** 2


def k(x, s):
    return x / (1 + s)
```

(continues on next page)

(continued from previous page)

```
def lower_bound(x):
    return 0


def upper_bound(x):
    return 1


def f(y):
    return y


if __name__ == '__main__':
    ides = [
        IDESolver(
            x = np.linspace(0, 1, 100),
            y_0 = 0,
            c = c,
            d = d,
            k = k,
            lower_bound = lower_bound,
            upper_bound = upper_bound,
            f = f,
        )
        for y_0 in np.linspace(0, 1, 10)
    ]

    with multiprocessing.Pool(processes = 2) as pool:
        results = pool.map(run, ides)

    print(results)
```

Note that the callables all need to defined before the if-name-main so that they can be pickled.

CHAPTER

FIVE

CHANGE LOG

5.1 v1.1.0

- Add support for multidimensional IDEs (PR #35 resolves #28, thanks nbrucy!)

5.2 v1.0.5

- Relaxes dependency version restrictions in advance of changes to `pip`. There shouldn't be any impact on users.

5.3 v1.0.4

- Revision of packaging and CI flow. There shouldn't be any impact on users.

5.4 v1.0.3

- Revision of package structure and CI flow. There shouldn't be any impact on users.

5.5 v1.0.2

- IDESolver now explicitly requires Python 3.6+ on install. Dependencies on `numpy` and `scipy` are given as lower bounds.

5.6 v1.0.1

- Changed the name of `IDESolver.F` to `f`, as intended.
- The default global error function is now injected instead of hard-coded.

5.7 v1.0.0

Initial release.

PYTHON MODULE INDEX

i

INDEX