The Journal of Open Source Software

py-pde: A Python package for solving partial differential equations

David Zwicker[1]

1 Max Planck Institute for Dynamics and Self-Organization, Göttingen, Germany

DOI: 10.21105/joss.02158

Software

- Review
- Repository
- Archive

Editor: Juanjo Bazán

Reviewers:

- @celliern
- @mstimberg

Submitted: 02 March 2020
Published: 03 April 2020

Summary

Partial differential equations (PDEs) play a central role in describing the dynamics of physical systems in research and in practical applications. However, equations appearing in realistic scenarios are typically non-linear and analytical solutions rarely exist. Instead, such systems are solved by numerical integration to provide insight into their behavior. Moreover, such investigations can motivate approximative solutions, which might then lead to analytical insight.

The py-pde python package presented in this paper allows researchers to quickly and conveniently simulate and analyze PDEs of the general form

$$\partial_t u(\boldsymbol{x}, t) = \mathcal{D}[u(\boldsymbol{x}, t)] + \eta(u, \boldsymbol{x}, t) ,$$

where $\mathcal{D}$ is a (non-linear) differential operator that defines the time evolution of a (set of) physical fields u with possibly tensorial character, which depend on spatial coordinates $\boldsymbol{x}$ and time t. The framework also supports stochastic differential equations in the Itô representation, indicated by the noise term η in the equation above.

The main goal of the py-pde package is to provide a convenient way to analyze general PDEs, while at the same time allowing for enough flexibility to easily implement specialized code for particular cases. Since the code is written in pure Python, it can be easily installed via pip by simply calling `pip install py-pde`. However, the central parts are just-in-time compiled using `numba` (Lam, Pitrou, & Seibert, 2015) for computational efficiency. To improve user interaction further, some arguments accept mathematical expressions that are parsed by `sympy` (Meurer et al., 2017) and are compiled optionally. This approach lowers the barrier for new users while also providing speed and flexibility for advanced use cases. Moreover, the package provides convenience functions for creating suitable initial conditions, for controlling what is analyzed as well as stored during a simulation, and for visualizing the final results. The py-pde package thus serves as a toolbox for exploring PDEs for researchers as well as for students who want to delve into the fascinating world of dynamical systems.

Methods

The basic objects of the py-pde package are scalar and tensorial fields defined on various discretized grids. These grids can deal with periodic boundary conditions and they allow exploiting spatial symmetries that might be present in the physical problem. For instance, the scalar field $f(z, r) = \sqrt{z} * e^{-r^2}$ in cylindrical coordinates assuming azimuthal symmetry can be visualized using

```
grid = pde.CylindricalGrid(radius=5, bounds_z=[0, 1], shape=(32, 8))
field = pde.ScalarField.from_expression(grid, 'sqrt(z) * exp(-r**2)')
field.plot()
```

The package defines common differential operators that act directly on the fields. For instance, calling `field.gradient(bc='neumann')` returns a vector field on the same cylindrical grid where the components correspond to the gradient of `field` assuming Neumann boundary conditions. Here, differential operators are evaluated using the finite difference method (FDM) and the package supports various boundary conditions, which can be separately specified per field and boundary. The discretized fields are the foundation of the `py-pde` package and allow the comfortable construction of initial conditions, the visualization of final results, and the detailed investigation of intermediate data.

The main part of the `py-pde` package provides the infrastructure for solving partial differential equations. Here, we use the method of lines by explicitly discretizing space using the grid classes described above. This reduces the PDEs to a set of ordinary differential equations, which can be solved using standard methods. For instance, the diffusion equation $\partial_t u = \nabla^2 u$ on the cylindrical grid defined above can be solved by

```
eq = pde.DiffusionPDE()
result = eq.solve(field, t_range=[0, 10])
```

Note that the partial differential equation is defined independent of the grid, allowing use of the same implementation for various geometries. The package provides simple implementations of standard PDEs, but extensions are simple to realize. In particular, the differential operator $\mathcal{D}$ can be implemented in pure Python for initial testing, while a more specialized version compiled with `numba` (Lam et al., 2015) might be added later for speed. This approach allows fast testing of new PDEs while also providing an avenue for efficient calculations later.

The flexibility of `py-pde` is one of its key features. For instance, while the package implements forward and backward Euler methods as well as a Runge-Kutta scheme, users might require more sophisticated solvers. We already provide a wrapper for the excellent `scipy.integrate.solve_ivp` method from the SciPy package (Virtanen et al., 2020) and further additions are straightforward. Finally, the explicit Euler stepper provided by `py-pde` also supports stochastic differential equations in the Itô representation. The standard PDE classes support additive Gaussian white noise, but alternatives, including multiplicative noise, can be specified in user-defined classes. This feature allows users to quickly test the effect of noise on dynamical systems without in-depth knowledge of the associated numerical implementation.

Finally, the package provides many convenience methods that allow analyzing simulations on the fly, storing data persistently, and visualizing the temporal evolution of quantities of interest. These features might be helpful even when not dealing with PDEs. For instance, the result of applying differential operators on the discretized fields can be visualized directly. Here, the excellent integration of `matplotlib` (Hunter, 2007) into Jupyter notebooks (Pérez & Granger, 2007) allows for an efficient workflow.

The `py-pde` package employs a consistent object-oriented approach, where each component can be extended and some can even be used in isolation. For instance, the numba-compiled finite-difference operators, which support flexible boundary conditions, can be applied to `numpy.ndarrays` directly, e.g., in custom applications. Generally, the just-in-time compilation provided by numba (Lam et al., 2015) allows for numerically efficient code while making deploying code easy. In particular, the package can be distributed to a cluster using `pip` without worrying about setting paths or compiling source code.

The `py-pde` package joins a long list of software packages that aid researchers in analyzing PDEs. Lately, there have been several attempts at simplifying the process of translating the mathematical formulation of a PDE to a numerical implementation on the computer. Most notably, the finite-difference approach has been favored by the packages `scikit-finite-diff` (Cellier & Ruyer-Quil, 2019) and `Devito` (Louboutin et al., 2019). Conversely, finite-element and finite-volume methods provide more flexibility in the geometries considered and have been used in major packages, including `FEniCS` (Alnæs et al., 2015), `FiPy` (Guyer, Wheeler, & Warren, 2009), `pyclaw` (Ketcheson et al., 2012), and `SfePy` (Cimrman, Lukeš, & Rohan, 2019). Finally, spectral methods are another popular approach for calculating differentials of discretized fields, e.g., in the `dedalus project` (Burns, Vasil, Oishi, Lecoanet, & Brown, 2019). While these methods could in principle also be implemented in `py-pde`, they are limited to a small set of viable boundary conditions and are thus not the primary focus. Instead, `py-pde` aims at providing a full toolchain for creating, simulating, and analyzing PDEs and the associated fields. While being useful in research, `py-pde` might thus also suitable for education.

Acknowledgements

I am thankful to Jan Kirschbaum, Ajinkya Kulkarni, Estefania Vidal, and Noah Ziethen for discussions and critical testing of this package.

References

Alnæs, M. S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., et al. (2015). The FEniCS Project Version 1.5. *Archive of Numerical Software*, *3*(100). doi:10.11588/ans.2015.100.20553

Burns, K. J., Vasil, G. M., Oishi, J. S., Lecoanet, D., & Brown, B. P. (2019). Dedalus: A Flexible Framework for Numerical Simulations with Spectral Methods. *arXiv e-prints*, arXiv:1905.10388. Retrieved from http://arxiv.org/abs/1905.10388

Cellier, N., & Ruyer-Quil, C. (2019). scikit-finite-diff, a new tool for PDE solving. *Journal of Open Source Software*, *4*(38), 1356. doi:10.21105/joss.01356

Cimrman, R., Lukeš, V., & Rohan, E. (2019). Multiscale finite element calculations in Python using SfePy. *Advances in Computational Mathematics*. doi:10.1007/s10444-019-09666-0

Guyer, J. E., Wheeler, D., & Warren, J. A. (2009). FiPy: Partial differential equations with Python. *Computing in Science & Engineering*, *11*(3), 6–15. doi:10.1109/MCSE.2009.52

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Ketcheson, D. I., Mandli, K. T., Ahmadia, A. J., Alghamdi, A., Quezada de Luna, M., Parsani, M., Knepley, M. G., et al. (2012). PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. *SIAM Journal on Scientific Computing*, *34*(4), C210–C231. doi:10.1137/110856976

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the second workshop on the LLVM compiler infrastructure in HPC*, LLVM '15. New York, NY, USA: Association for Computing Machinery. doi:10.1145/2833157.2833162

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., et al. (2019). Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, *12*(3), 1165–1187. doi:10.5194/gmd-12-1165-2019

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., et al. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, *3*, e103. doi:10.7717/peerj-cs.103

Pérez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science & Engineering*, *9*(3), 21–29. doi:10.1109/MCSE.2007.53

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. doi:10.1038/s41592-019-0686-2