py-pde Documentation

Release unknown

David Zwicker

Jul 10, 2026

CONTENTS

1 Getting started **3**
- 1.1 When (not) to use the package . . . 3
- 1.2 Installation . . . 3
  - 1.2.1 Install using pip . . . 3
  - 1.2.2 Install using conda . . . 4
  - 1.2.3 Install from source . . . 4
    - Required prerequisites . . . 4
    - Optional packages . . . 4
    - Downloading *py-pde* . . . 5
- 1.3 A simple example . . . 5
- 1.4 Package overview . . . 5

2 Examples **7**
- 2.1 Grids and fields . . . 7
  - 2.1.1 Plot a polar grid . . . 7
  - 2.1.2 Plotting a vector field . . . 8
  - 2.1.3 Plotting a scalar field in cylindrical coordinates . . . 9
  - 2.1.4 Random scalar fields . . . 9
  - 2.1.5 Visualizing a scalar field . . . 11
  - 2.1.6 Finite differences approximation . . . 12
- 2.2 Simple PDEs . . . 14
  - 2.2.1 Solving Laplace's equation in 2d . . . 14
  - 2.2.2 Solving Poisson's equation in 1d . . . 15
  - 2.2.3 Simple diffusion equation . . . 15
  - 2.2.4 Kuramoto-Sivashinsky - Using *PDE* class . . . 17
  - 2.2.5 Spherically symmetric PDE . . . 18
  - 2.2.6 Diffusion on a Cartesian grid . . . 19
  - 2.2.7 Klein-Gordon equation . . . 20
  - 2.2.8 Stochastic simulation . . . 22
  - 2.2.9 Setting boundary conditions . . . 22
  - 2.2.10 Time-dependent boundary conditions . . . 24
  - 2.2.11 1D problem - Using *PDE* class . . . 25
  - 2.2.12 Brusselator - Using the *PDE* class . . . 26
  - 2.2.13 Diffusion equation with spatial dependence . . . 27
  - 2.2.14 Schrödinger's Equation . . . 28
- 2.3 Output and analysis . . . 29
  - 2.3.1 Storage examples . . . 29
  - 2.3.2 Logarithmic kymograph . . . 31
  - 2.3.3 Writing and reading trajectory data . . . 32
  - 2.3.4 Using simulation trackers . . . 32

2.4 Advanced PDEs . . . 34
2.4.1 Post-step hook function . . . 34
2.4.2 Heterogeneous boundary conditions . . . 36
2.4.3 Heterogeneous PDE . . . 37
2.4.4 Brusselator - Using the *ReactionDiffusionPDE* class . . . 39
2.4.5 Custom noise . . . 40
2.4.6 Kuramoto-Sivashinsky - Using custom class . . . 41
2.4.7 Custom Class for coupled PDEs . . . 43
2.4.8 SDE with Stratonovich interpretation . . . 44
2.4.9 1D problem - Using custom class . . . 46
2.4.10 Solver comparison . . . 48
2.4.11 Kuramoto-Sivashinsky - Compiled methods . . . 50
2.4.12 Post-step hook function in a custom class . . . 52
2.4.13 Custom PDE class: SIR model . . . 54
2.4.14 Brusselator - Using custom class . . . 56

3 User manual **59**
3.1 Mathematical basics . . . 59
3.1.1 Curvilinear coordinates . . . 59
Polar coordinates . . . 59
Spherical coordinates . . . 59
Cylindrical coordinates . . . 60
3.1.2 Spatial discretization . . . 60
3.1.3 Temporal evolution . . . 61
3.2 Basic usage . . . 61
3.2.1 Defining the geometry . . . 61
3.2.2 Initializing a field . . . 61
3.2.3 Specifying the PDE . . . 62
3.2.4 Running the simulation . . . 62
3.2.5 Analyzing the results . . . 62
3.3 Advanced usage . . . 62
3.3.1 Boundary conditions . . . 62
3.3.2 Expressions . . . 64
3.3.3 Custom PDE classes . . . 65
3.3.4 Low-level operators . . . 66
Differential operators . . . 66
Field integration . . . 67
Field interpolation . . . 67
Inner products . . . 68
3.3.5 Compiled implementations of PDEs . . . 68
3.3.6 Stochastic partial differential equation . . . 69
3.4 Configuration parameters . . . 71
3.5 Performance . . . 72
3.5.1 Measuring performance . . . 72
3.5.2 Improving performance . . . 74
3.5.3 Multiprocessing using MPI . . . 74
3.6 Contributing code . . . 75
3.6.1 Structure of the package . . . 75
3.6.2 Extending functionality . . . 75
3.6.3 Data layout . . . 76
3.6.4 Coding style . . . 77
3.6.5 Running unit tests . . . 77
3.7 Citing the package . . . 77
3.8 Code of Conduct . . . 78

3.8.1 Our Pledge . . . 78
3.8.2 Our Standards . . . 78
3.8.3 Our Responsibilities . . . 78
3.8.4 Scope . . . 78
3.8.5 Enforcement . . . 79
3.8.6 Attribution . . . 79

4 Reference manual **81**
4.1 pde.backends package . . . 81
4.1.1 pde.backends.jax package . . . 90
pde.backends.jax.operators package . . . 90
pde.backends.jax.operators.cartesian module . . . 90
pde.backends.jax.operators.cylindrical_sym module . . . 92
pde.backends.jax.operators.polar_sym module . . . 94
pde.backends.jax.operators.spherical_sym module . . . 95
pde.backends.jax.backend module . . . 98
pde.backends.jax.config module . . . 103
pde.backends.jax.typing module . . . 103
4.1.2 pde.backends.numba package . . . 104
pde.backends.numba.operators package . . . 104
pde.backends.numba.operators.cartesian module . . . 104
pde.backends.numba.operators.common module . . . 106
pde.backends.numba.operators.cylindrical_sym module . . . 107
pde.backends.numba.operators.polar_sym module . . . 109
pde.backends.numba.operators.spherical_sym module . . . 111
pde.backends.numba.backend module . . . 115
pde.backends.numba.config module . . . 120
pde.backends.numba.grids module . . . 120
pde.backends.numba.overloads module . . . 121
pde.backends.numba.utils module . . . 122
4.1.3 pde.backends.numba_mpi package . . . 124
pde.backends.numba_mpi.backend module . . . 125
pde.backends.numba_mpi.overloads module . . . 126
4.1.4 pde.backends.numpy package . . . 126
pde.backends.numpy.backend module . . . 126
4.1.5 pde.backends.scipy package . . . 130
pde.backends.scipy.operators package . . . 130
pde.backends.scipy.operators.cartesian module . . . 130
pde.backends.scipy.operators.common module . . . 132
pde.backends.scipy.operators.cylindrical_sym module . . . 133
pde.backends.scipy.operators.polar_sym module . . . 133
pde.backends.scipy.operators.spherical_sym module . . . 134
pde.backends.scipy.backend module . . . 134
4.1.6 pde.backends.torch package . . . 135
pde.backends.torch.operators package . . . 135
pde.backends.torch.operators.cartesian module . . . 135
pde.backends.torch.operators.common module . . . 139
pde.backends.torch.operators.cylindrical_sym module . . . 140
pde.backends.torch.operators.polar_sym module . . . 144
pde.backends.torch.operators.spherical_sym module . . . 148
pde.backends.torch.backend module . . . 152
pde.backends.torch.config module . . . 157
pde.backends.torch.typing module . . . 157
pde.backends.torch.utils module . . . 157

4.1.7 pde.backends.base module . . . 158
4.1.8 pde.backends.registry module . . . 166
4.2 pde.fields package . . . 168
4.2.1 pde.fields.base module . . . 190
4.2.2 pde.fields.collection module . . . 195
4.2.3 pde.fields.datafield_base module . . . 201
4.2.4 pde.fields.scalar module . . . 212
4.2.5 pde.fields.tensorial module . . . 217
4.2.6 pde.fields.vectorial module . . . 221
4.3 pde.grids package . . . 227
4.3.1 pde.grids.boundaries package . . . 228
Boundary conditions . . . 228
Boundaries overview . . . 230
pde.grids.boundaries.axes module . . . 231
pde.grids.boundaries.axis module . . . 235
pde.grids.boundaries.local module . . . 238
4.3.2 pde.grids.coordinates package . . . 260
pde.grids.coordinates.base module . . . 262
pde.grids.coordinates.bipolar module . . . 265
pde.grids.coordinates.bispherical module . . . 266
pde.grids.coordinates.cartesian module . . . 266
pde.grids.coordinates.cylindrical module . . . 266
pde.grids.coordinates.polar module . . . 267
pde.grids.coordinates.spherical module . . . 267
4.3.3 pde.grids.base module . . . 268
4.3.4 pde.grids.cartesian module . . . 279
4.3.5 pde.grids.cylindrical module . . . 284
4.3.6 pde.grids.spherical module . . . 288
4.4 pde.pdes package . . . 293
4.4.1 pde.pdes.allen_cahn module . . . 294
4.4.2 pde.pdes.base module . . . 295
4.4.3 pde.pdes.cahn_hilliard module . . . 302
4.4.4 pde.pdes.diffusion module . . . 303
4.4.5 pde.pdes.klein_gordon module . . . 304
4.4.6 pde.pdes.kpz_interface module . . . 306
4.4.7 pde.pdes.kuramoto_sivashinsky module . . . 307
4.4.8 pde.pdes.laplace module . . . 309
4.4.9 pde.pdes.pde module . . . 311
4.4.10 pde.pdes.reaction_diffusion module . . . 313
4.4.11 pde.pdes.swift_hohenberg module . . . 314
4.4.12 pde.pdes.wave module . . . 316
4.5 pde.solvers package . . . 317
4.5.1 pde.solvers.adams_bashforth module . . . 318
4.5.2 pde.solvers.base module . . . 318
4.5.3 pde.solvers.controller module . . . 321
4.5.4 pde.solvers.crank_nicolson module . . . 322
4.5.5 pde.solvers.euler module . . . 322
4.5.6 pde.solvers.explicit_mpi module . . . 323
4.5.7 pde.solvers.implicit module . . . 324
4.5.8 pde.solvers.milstein module . . . 325
4.5.9 pde.solvers.runge_kutta module . . . 325
4.5.10 pde.solvers.scipy module . . . 326
4.6 pde.storage package . . . 326
4.6.1 pde.storage.base module . . . 333

4.6.2 pde.storage.file module . . . 338
4.6.3 pde.storage.memory module . . . 340
4.6.4 pde.storage.modelrunner module . . . 341
4.6.5 pde.storage.movie module . . . 343
4.7 pde.tools package . . . 345
4.7.1 pde.tools.cache module . . . 346
4.7.2 pde.tools.config module . . . 351
4.7.3 pde.tools.cuboid module . . . 356
4.7.4 pde.tools.docstrings module . . . 358
4.7.5 pde.tools.expressions module . . . 359
4.7.6 pde.tools.ffmpeg module . . . 365
4.7.7 pde.tools.math module . . . 367
4.7.8 pde.tools.misc module . . . 368
4.7.9 pde.tools.modelrunner module . . . 372
4.7.10 pde.tools.mpi module . . . 372
4.7.11 pde.tools.nested_dict module . . . 374
4.7.12 pde.tools.numba module . . . 377
4.7.13 pde.tools.output module . . . 377
4.7.14 pde.tools.parse_duration module . . . 378
4.7.15 pde.tools.plotting module . . . 379
4.7.16 pde.tools.spectral module . . . 384
4.7.17 pde.tools.typing module . . . 385
4.8 pde.trackers package . . . 387
4.8.1 pde.trackers.base module . . . 388
4.8.2 pde.trackers.interactive module . . . 391
4.8.3 pde.trackers.interrupts module . . . 393
4.8.4 pde.trackers.trackers module . . . 397
4.9 pde.visualization package . . . 408
4.9.1 pde.visualization.movies module . . . 408
4.9.2 pde.visualization.plotting module . . . 411

Python Module Index **417**

The *py-pde* python package provides methods and classes useful for solving partial differential equations (PDEs) of the 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) operator containing spatial derivatives 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, where the noise is represented by η above.

The main audience for the package are researchers and students who want to investigate the behavior of a PDE and get an intuitive understanding of the role of the different terms and the boundary conditions. To support this, *py-pde* evaluates PDEs using the methods of lines with a finite-difference approximation of the differential operators. Consequently, the mathematical operator $\mathcal{D}$ can be naturally translated to a function evaluating the evolution rate of the PDE. Moreover, core computations can be compiled transparently using *numba*, *jax*, or *torch* for speed.

Contents

CHAPTER

ONE

GETTING STARTED

1.1 When (not) to use the package

py-pde provides a straight-forward way to simulate partial differential equations (PDEs) using a finite-difference scheme. Advantages of this approach include:

- Supports non-linear PDEs with complex boundary conditions.
- Direct specification of the evolution equations using a syntax that is similar to the underlying mathematical equations.
- Supports collections of multiple fields of vectorial or tensorial character.

However, there are of course also disadvantages to the approach taken by *py-pde*:

- Only suited for simple geometries (like rectangles, disks, or cylinders) with a fixed discretization.
- Optimized for PDEs that describe the time-evolution of a physical system. Neither time-independent systems nor integro-differential equations are fully supported.
- Finite-differences can lead to numerical instabilities.
- Provided generic solvers might not be most suitable choice for particular equations.

1.2 Installation

The *py-pde* package is developed for python 3.10 and has been tested up to version 3.14 under Linux, Windows, and macOS. Before you can start using the package, you need to install it using one of the following methods.

1.2.1 Install using pip

The package is available on pypi, so you should be able to install it by running

```
pip install py-pde
```

In order to have all features of the package available, you might also want to install the following optional packages:

```
pip install h5py pandas pyfftw tqdm
```

Moreover, **`ffmpeg`** needs to be installed for creating movies.

1.2.2 Install using conda

The *py-pde* package is also available on conda using the *conda-forge* channel. You can thus install it using

```
conda install -c conda-forge py-pde
```

This installation includes many dependencies to have most features of *py-pde*.

1.2.3 Install from source

Installing from source can be necessary if the pypi installation does not work or if the latest source code should be installed from github.

Required prerequisites

The code builds on other python packages, which need to be installed for *py-pde* to function properly. The required packages are listed in the table below:

Package	Minimal version	Usage
matplotlib	3.1	Visualizing results
numba	0.59	Just-in-time compilation to accelerate numerics
numpy	2	Handling numerical data
scipy	1.10	Miscellaneous scientific functions
sympy	1.9	Dealing with user-defined mathematical expressions
tqdm	4.66	Display progress bars during calculations
typing_extensions	4.10	Backports of typing features

The simplest way to install these packages is to use the `requirements.txt` in the base folder:

```
pip install -r requirements.txt
```

Alternatively, these package can be installed via your operating system's package manager, e.g. using **`macports`**, **`homebrew`**, or **`conda`**. The package versions given above are minimal requirements, although this is not tested systematically. Generally, it should help to install the latest version of the package.

Optional packages

The following packages should be installed to use some miscellaneous features:

Package	Minimal version	Usage
ffmpeg-python	0.2	Reading and writing videos
h5py	2.10	Storing data in the hierarchical file format
ipywidgets	8	Jupyter notebook support
jax	0.7	Using *jax* as a backend
mpi4py	3	Parallel processing using MPI
napari	0.4.8	Displaying images interactively
numba-mpi	0.22	Parallel processing using MPI+numba
pandas	2	Handling tabular data
py-modelrunner	0.19	Running simulations and handling I/O
rocket-fft	0.2.4	Numba-compiled fast Fourier transforms
torch	2.9	Using *torch* as a backend

For making movies, the **ffmpeg** should be available. Additional packages might be required for running the tests in the folder `tests` and to build the documentation in the folder `docs`. These packages are listed in the files `requirements.txt` in the respective folders.

Downloading *py-pde*

The package can be simply checked out from github.com/zwicker-group/py-pde. To import the package from any python session, it might be convenient to include the root folder of the package into the `PYTHONPATH` environment variable.

This documentation can be built by calling the **make html** in the `docs` folder. The final documentation will be available in `docs/build/html`. Note that a LaTeX documentation can be build using **make latexpdf**.

1.3 A simple example

After installation, the easiest way to get started is to run a diffusion simulation on a one-dimensional grid. The example below shows the full workflow: create a grid, initialize a field, define a PDE, solve it, and inspect the final result.

```
import pde

# 1. Define a one-dimensional domain with 64 support points
grid = pde.UnitGrid([64])

# 2. Create an initial scalar field with random values
state = pde.ScalarField.random_uniform(grid, vmin=0, vmax=1)

# 3. Define a simple PDE (diffusion equation)
eq = pde.DiffusionPDE(diffusivity=0.1)

# 4. Evolve the field in time
result = eq.solve(state, t_range=1, dt=1e-3)

# 5. Inspect the final state
result.plot()
```

This script runs with the default solver and uses only the core API. In practice, this is often enough to prototype a first model quickly. Running the script should open a line plot of the final one-dimensional field. Since the initial condition is random and diffusion smooths spatial variations, the final profile is typically less jagged than the initial state.

You can read more about the package in the next section, or you move on to the *Basic usage* section to learn about boundary conditions, custom equations, trackers, and result storage.

1.4 Package overview

The main aim of the *pde* package is to simulate partial differential equations in simple geometries. Here, the time evolution of a PDE is determined using the method of lines by explicitly discretizing space using fixed grids. The differential operators are implemented using the finite difference method. For simplicity, we consider only regular, orthogonal grids, where each axis has a uniform discretization and all axes are (locally) orthogonal. Currently, we support simulations on *CartesianGrid*, *PolarSymGrid*, *SphericalSymGrid*, and *CylindricalSymGrid*, with and without periodic boundaries where applicable.

Fields are defined by specifying values at the grid points using the classes *ScalarField*, *VectorField*, and *Tensor2Field*. These classes provide methods for applying differential operators to the fields, e.g., the result of applying the Laplacian to a scalar field is returned by calling the method *laplace()*, which returns another instance of *ScalarField*, whereas *gradient()* returns a *VectorField*. Combining these functions with ordinary arithmetics

on fields allows to represent the right hand side of many partial differential equations that appear in physics. Importantly, the differential operators work with flexible boundary conditions.

The PDEs to solve are represented as a separate class inheriting from *PDEBase*. One example defined in this package is the diffusion equation implemented as *DiffusionPDE*, but more specific situations need to be implemented by the user. Most notably, PDEs can be specified by their expression using the convenient *PDE* class.

The PDEs are solved using solver classes, where a simple explicit solver is implemented by *EulerSolver*, but more advanced implementations are available. Each solver object represents the integration strategy and configuration; during simulation it constructs an executable stepping function that advances the state in time. To obtain more details during the simulation, trackers can be supplied to the controller, which analyze intermediate states periodically. Typical trackers include *ProgressTracker* (display simulation progress), *PlotTracker* (display images of the simulation), and *SteadyStateTracker* (aborting simulation when a stationary state is reached). Others can be found in the *trackers* module. Moreover, we provide *MemoryStorage* and *FileStorage*, which can be used as trackers to store intermediate states in memory and in files, respectively.

CHAPTER

TWO

EXAMPLES

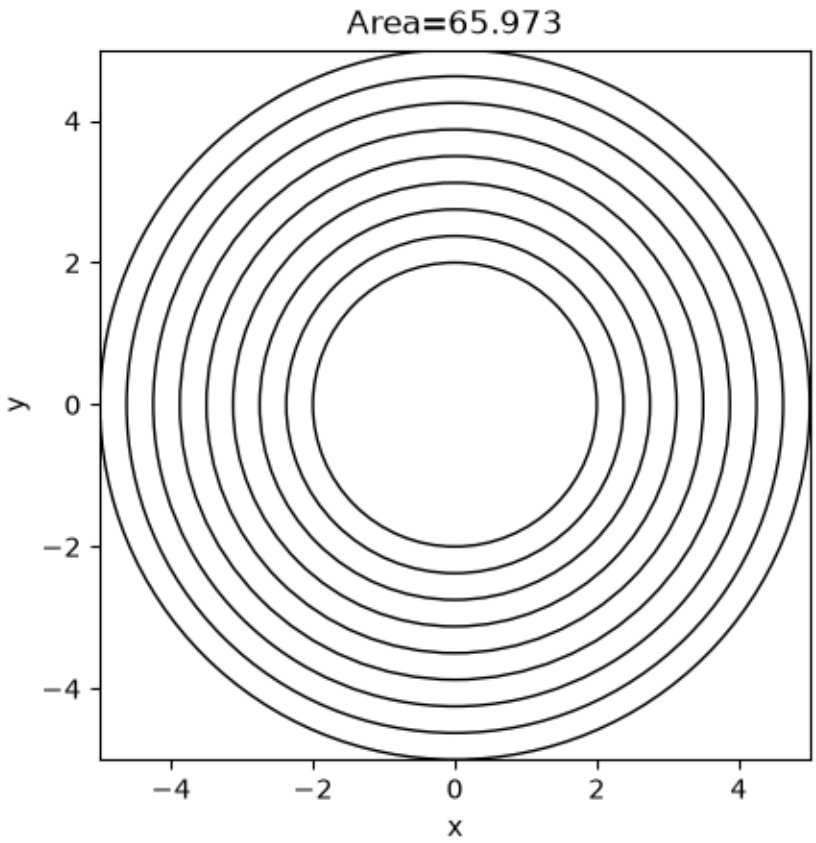
These are example scripts using the *py-pde* package, which illustrates some of the most important features of the package.

2.1 Grids and fields

These examples show how to define and manipulate discretized fields.

2.1.1 Plot a polar grid

This example shows how to initialize a polar grid with a hole inside and angular symmetry, so that fields only depend on the radial coordinate.

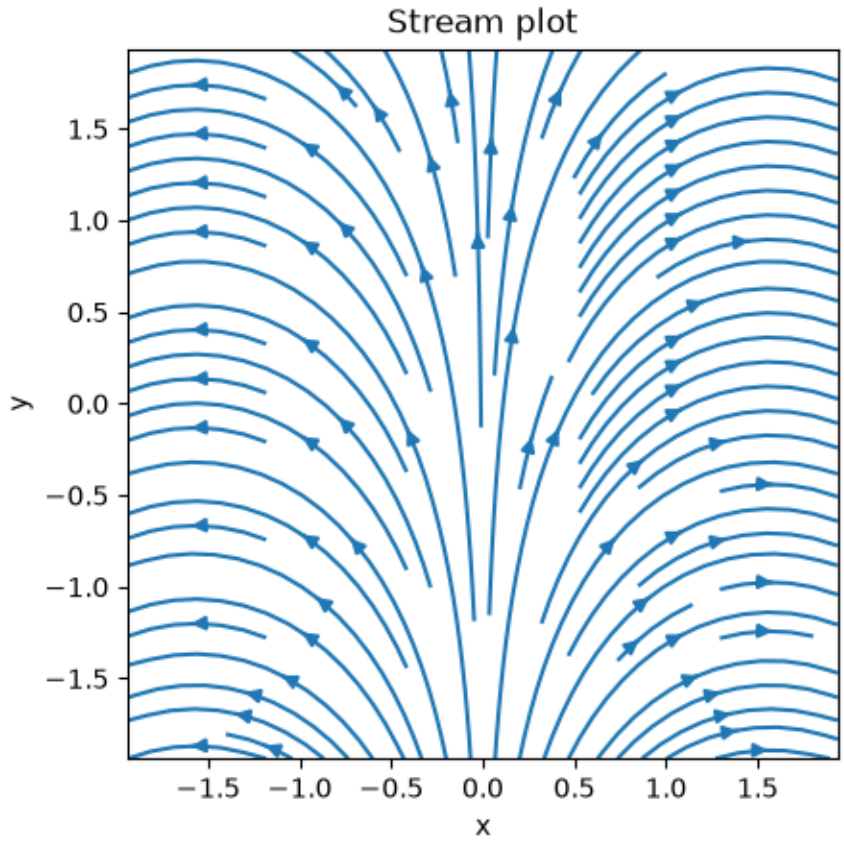


```
from pde import PolarSymGrid

grid = PolarSymGrid((2, 5), 8)
grid.plot(title=f"Area={grid.volume:.5g}")
```

Total running time of the script: (0 minutes 0.054 seconds)

2.1.2 Plotting a vector field

This example shows how to initialize and visualize the vector field $\boldsymbol{u} = \big(\sin(x), \cos(x)\big)$.

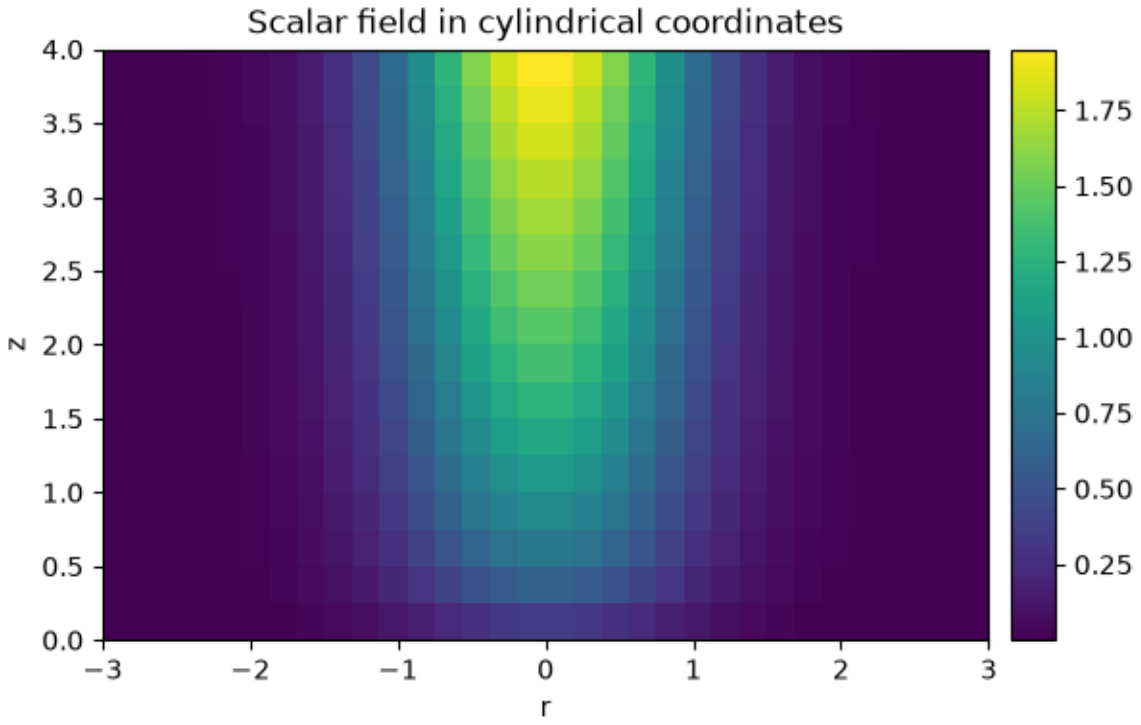



```
from pde import CartesianGrid, VectorField

grid = CartesianGrid([[-2, 2], [-2, 2]], 32)
field = VectorField.from_expression(grid, ["sin(x)", "cos(x)"])
field.plot(method="streamplot", title="Stream plot")
```

Total running time of the script: (0 minutes 0.300 seconds)

2.1.3 Plotting a scalar field in cylindrical coordinates

This example shows how to initialize and visualize the scalar field $u = \sqrt{z}\,\exp(-r^2)$ in cylindrical coordinates.

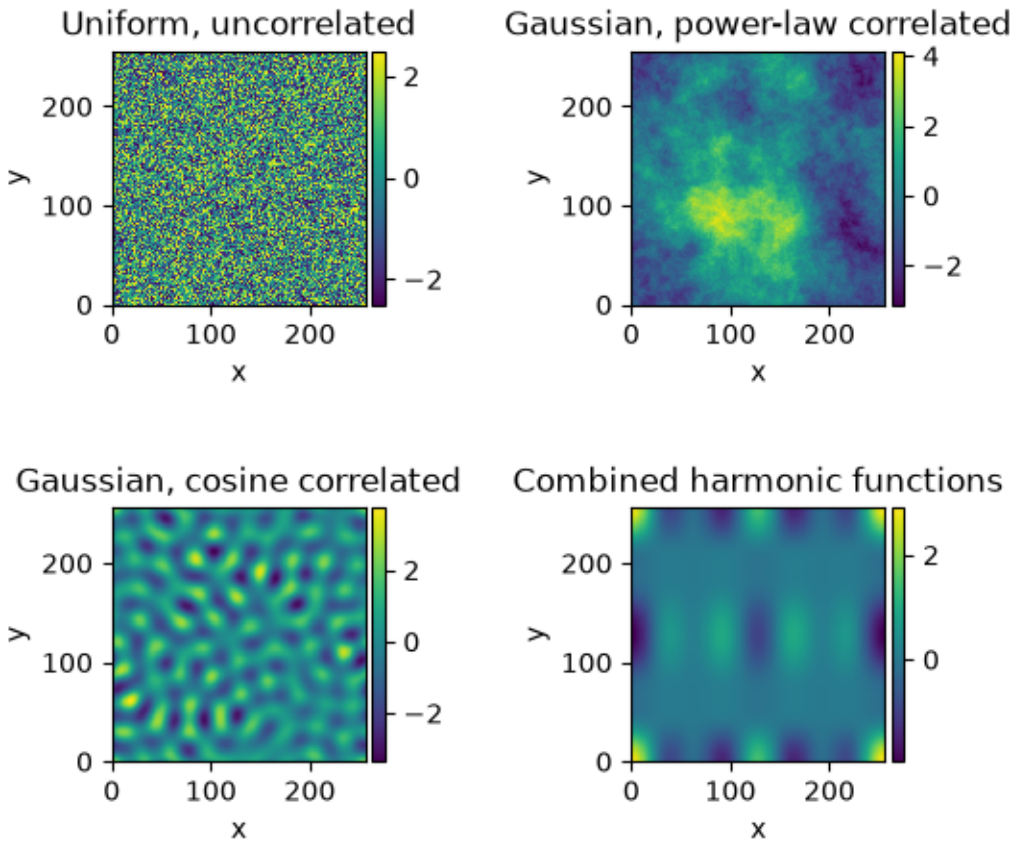



```
from pde import CylindricalSymGrid, ScalarField

grid = CylindricalSymGrid(radius=3, bounds_z=[0, 4], shape=16)
field = ScalarField.from_expression(grid, "sqrt(z) * exp(-r**2)")
field.plot(title="Scalar field in cylindrical coordinates")
```

Total running time of the script: (0 minutes 0.137 seconds)

2.1.4 Random scalar fields

This example showcases several random fields

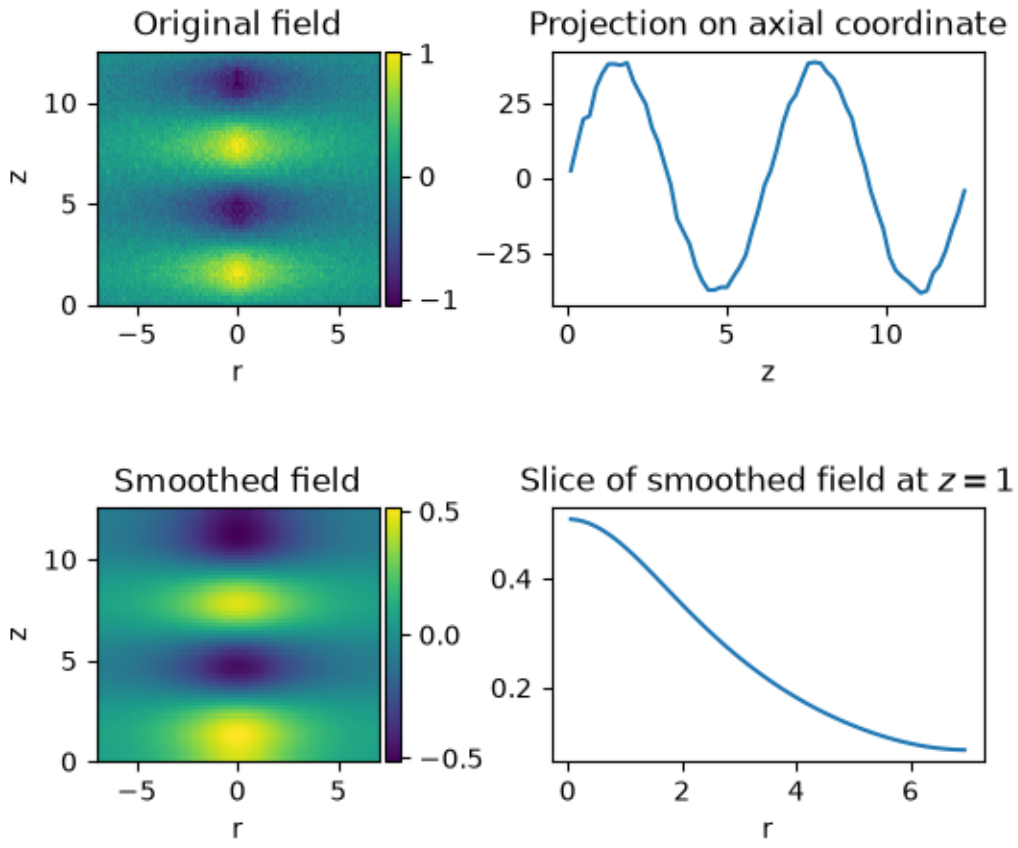



```
import matplotlib.pyplot as plt

from pde import ScalarField, UnitGrid

# initialize grid and plot figure
grid = UnitGrid([256, 256], periodic=True)
fig, axes = plt.subplots(nrows=2, ncols=2)

f1 = ScalarField.random_uniform(grid, -2.5, 2.5)
f1.plot(ax=axes[0, 0], title="Uniform, uncorrelated")

f2 = ScalarField.random_normal(grid, correlation="power law", exponent=-6)
f2.plot(ax=axes[0, 1], title="Gaussian, power-law correlated")

f3 = ScalarField.random_normal(grid, correlation="cosine", length_scale=30)
f3.plot(ax=axes[1, 0], title="Gaussian, cosine correlated")

f4 = ScalarField.random_harmonic(grid, modes=4)
f4.plot(ax=axes[1, 1], title="Combined harmonic functions")

plt.subplots_adjust(hspace=0.8)
plt.show()
```

Total running time of the script: (0 minutes 0.230 seconds)

2.1.5 Visualizing a scalar field

This example displays methods for visualizing scalar fields.




```
import matplotlib.pyplot as plt
import numpy as np

from pde import CylindricalSymGrid, ScalarField

# create a scalar field with some noise
grid = CylindricalSymGrid(7, [0, 4 * np.pi], 64)
data = ScalarField.from_expression(grid, "sin(z) * exp(-r / 3)")
data += 0.05 * ScalarField.random_normal(grid)

# manipulate the field
smoothed = data.smooth()  # Gaussian smoothing to get rid of the noise
projected = data.project("r")  # integrate along the radial direction
sliced = smoothed.slice({"z": 1})  # slice the smoothed data

# create four plots of the field and the modifications
fig, axes = plt.subplots(nrows=2, ncols=2)
data.plot(ax=axes[0, 0], title="Original field")
smoothed.plot(ax=axes[1, 0], title="Smoothed field")
projected.plot(ax=axes[0, 1], title="Projection on axial coordinate")
```

(continues on next page)

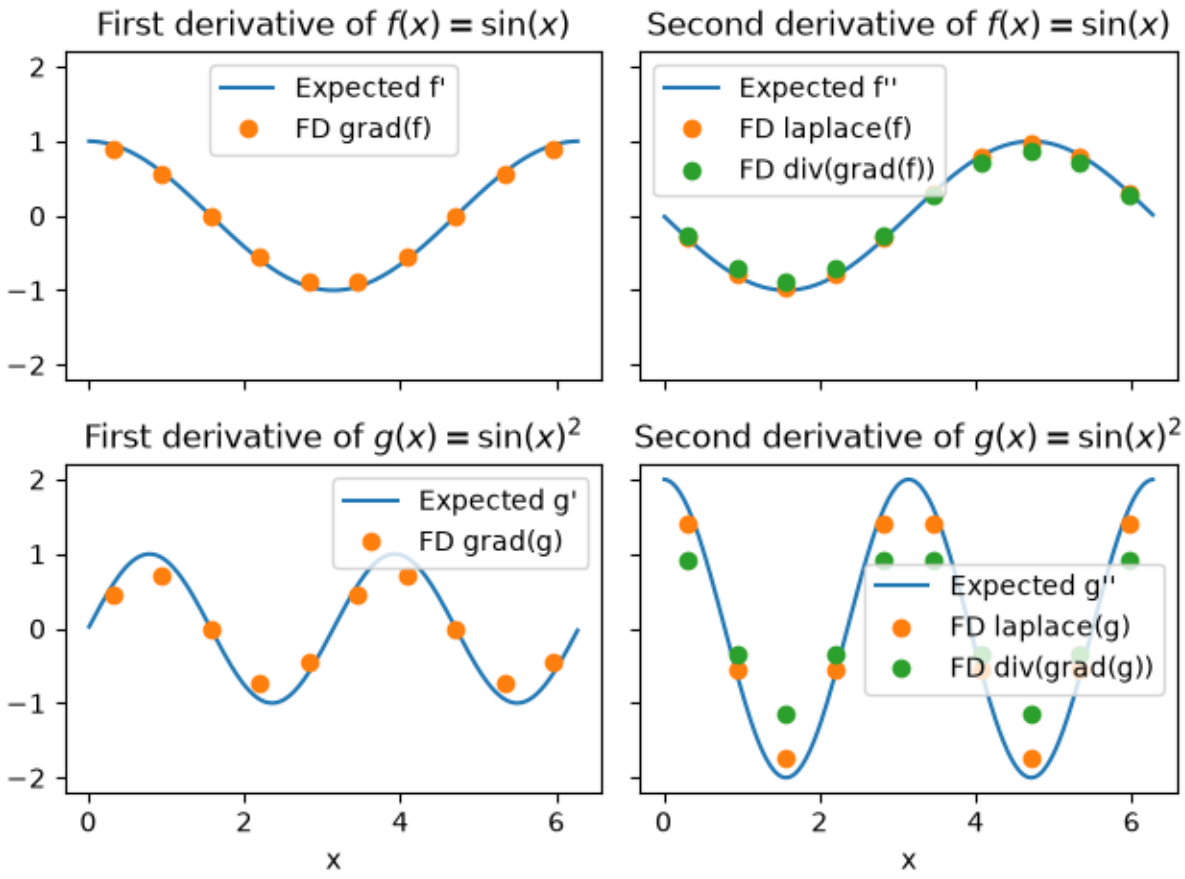
(continued from previous page)

```
sliced.plot(ax=axes[1, 1], title="Slice of smoothed field at $z=1$")
plt.subplots_adjust(hspace=0.8)
plt.show()
```

Total running time of the script: (0 minutes 0.389 seconds)

2.1.6 Finite differences approximation

This example displays various finite difference (FD) approximations of derivatives of simple harmonic function.




```
import matplotlib.pyplot as plt
import numpy as np

from pde import CartesianGrid, ScalarField
from pde.tools.expressions import evaluate

# create grids with different resolution to emphasize finite difference approximation
grid_fine = CartesianGrid([(0, 2 * np.pi)], 256, periodic=True)
grid_coarse = CartesianGrid([(0, 2 * np.pi)], 10, periodic=True)

# create figure to present plots of the derivative
fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True)
```

(continues on next page)

(continued from previous page)

```
# plot first derivatives of sin(x)
f = ScalarField.from_expression(grid_coarse, "sin(x)")
f_grad = f.gradient("periodic")  # first derivative (from gradient vector field)
ScalarField.from_expression(grid_fine, "cos(x)").plot(
    ax=axes[0, 0], label="Expected f'"
)
f_grad.plot(ax=axes[0, 0], label="FD grad(f)", ls="", marker="o")
plt.legend(frameon=True)
plt.ylabel("")
plt.xlabel("")
plt.title(r"First derivative of $f(x) = \sin(x)$")

# plot second derivatives of sin(x)
f_laplace = f.laplace("periodic")  # second derivative
f_grad2 = f_grad.divergence("periodic")  # second derivative using composition
ScalarField.from_expression(grid_fine, "-sin(x)").plot(
    ax=axes[0, 1], label="Expected f''"
)
f_laplace.plot(ax=axes[0, 1], label="FD laplace(f)", ls="", marker="o")
f_grad2.plot(ax=axes[0, 1], label="FD div(grad(f))", ls="", marker="o")
plt.legend(frameon=True)
plt.xlabel("")
plt.title(r"Second derivative of $f(x) = \sin(x)$")

# plot first derivatives of sin(x)**2
g_fine = ScalarField.from_expression(grid_fine, "sin(x)**2")
g = g_fine.interpolate_to_grid(grid_coarse)
expected = evaluate("2 * cos(x) * sin(x)", {"g": g_fine})
fd_1 = evaluate("d_dx(g)", {"g": g})  # first derivative (from directional derivative)
expected.plot(ax=axes[1, 0], label="Expected g'")
fd_1.plot(ax=axes[1, 0], label="FD grad(g)", ls="", marker="o")
plt.legend(frameon=True)
plt.title(r"First derivative of $g(x) = \sin(x)^2$")

# plot second derivatives of sin(x)**2
expected = evaluate("2 * cos(2 * x)", {"g": g_fine})
fd_2 = evaluate("d2_dx2(g)", {"g": g})  # second derivative
fd_11 = evaluate("d_dx(d_dx(g))", {"g": g})  # composition of first derivatives
expected.plot(ax=axes[1, 1], label="Expected g''")
fd_2.plot(ax=axes[1, 1], label="FD laplace(g)", ls="", marker="o")
fd_11.plot(ax=axes[1, 1], label="FD div(grad(g))", ls="", marker="o")
plt.legend(frameon=True)
plt.title(r"Second derivative of $g(x) = \sin(x)^2$")

# finalize plot
plt.tight_layout()
plt.show()
```

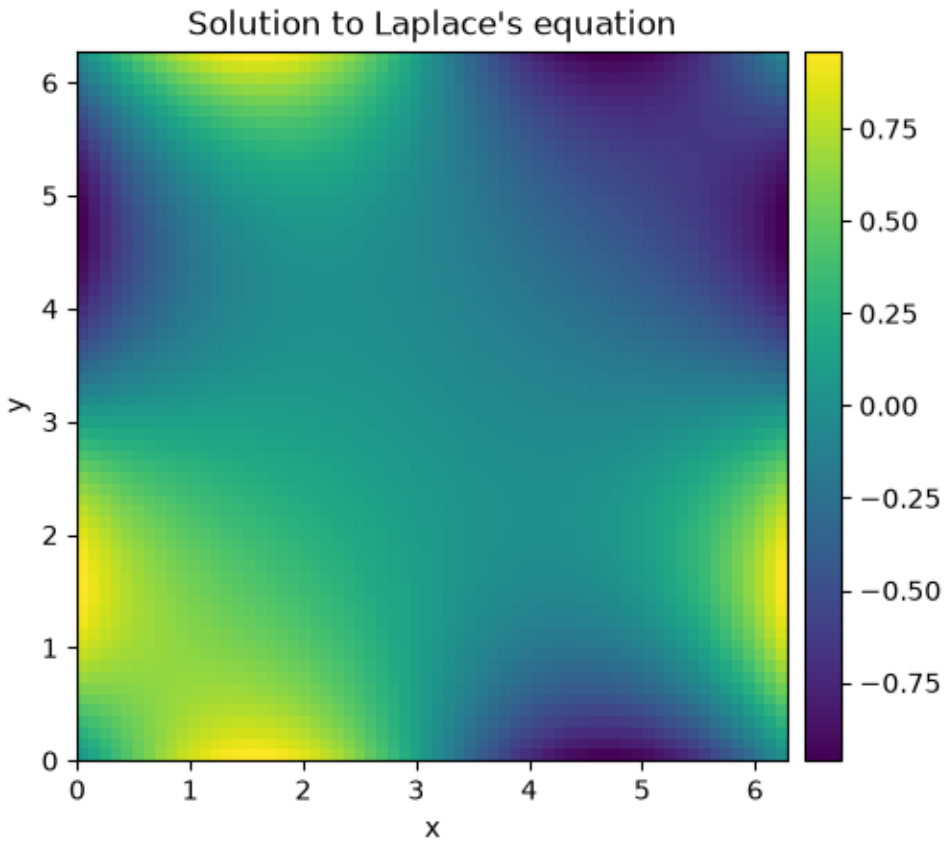
Total running time of the script: (0 minutes 6.266 seconds)

2.2 Simple PDEs

These examples demonstrate basic usage of the package to solve PDEs.

2.2.1 Solving Laplace's equation in 2d

This example shows how to solve a 2d Laplace equation with spatially varying boundary conditions.

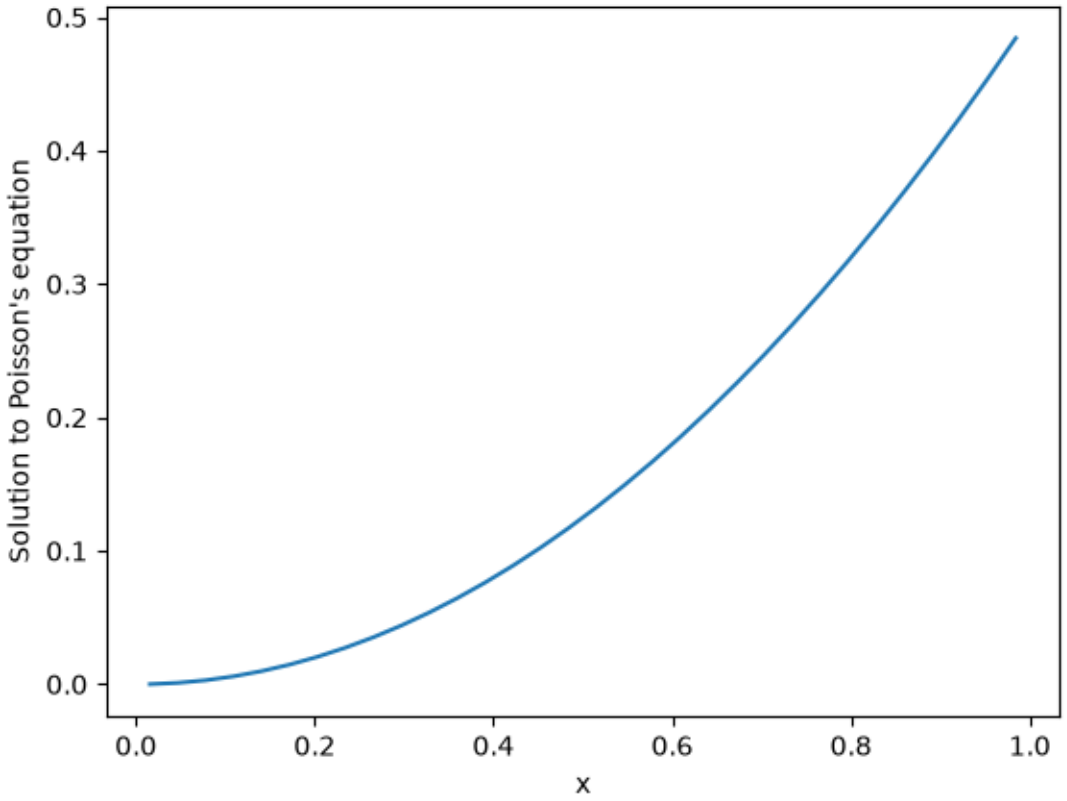



```
import numpy as np

from pde import CartesianGrid, solve_laplace_equation

grid = CartesianGrid([[0, 2 * np.pi], [0, 2 * np.pi]], 64)
bcs = {"x": {"value": "sin(y)"}, "y": {"value": "sin(x)"}}

res = solve_laplace_equation(grid, bcs)
res.plot()
```

Total running time of the script: (0 minutes 0.622 seconds)

2.2.2 Solving Poisson's equation in 1d

This example shows how to solve a 1d Poisson equation with boundary conditions.

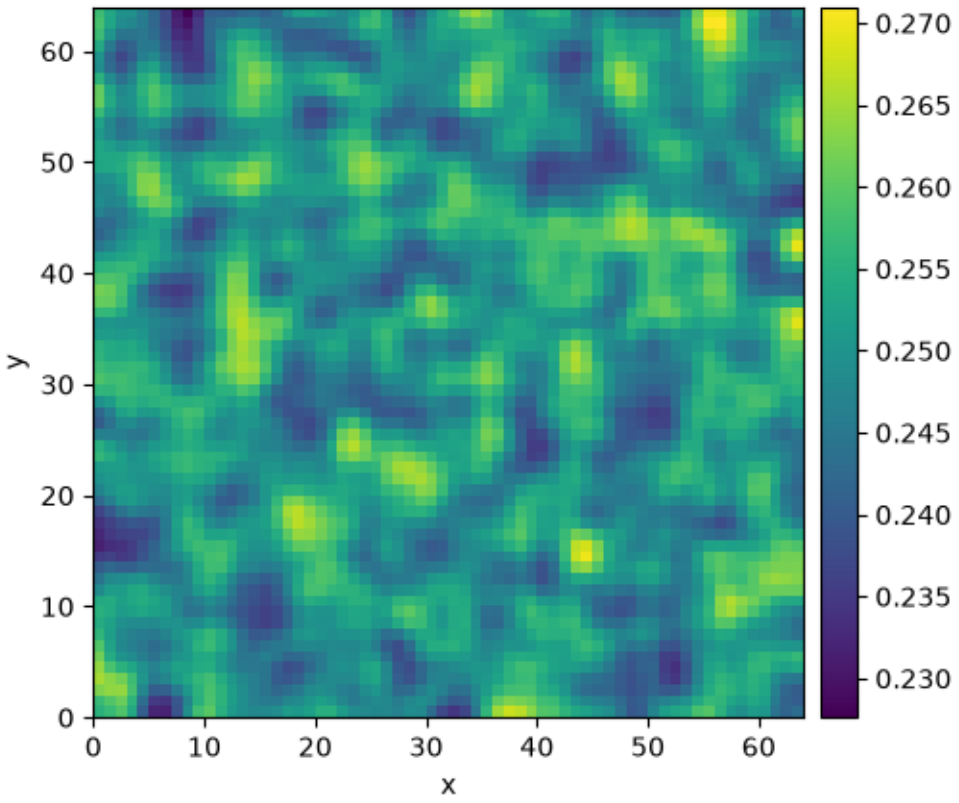



```
from pde import CartesianGrid, ScalarField, solve_poisson_equation

grid = CartesianGrid([[0, 1]], 32, periodic=False)
field = ScalarField(grid, 1)
result = solve_poisson_equation(field, bc={"x-": {"value": 0}, "x+": {"derivative": 1}
↪})

result.plot()
```

Total running time of the script: (0 minutes 0.058 seconds)

2.2.3 Simple diffusion equation

This example solves a simple diffusion equation in two dimensions.




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:11<?, ?it/s]
  0%|          | 0.00316/10.0 [00:11<10:01:49, 3612.05s/it]
  2%|▏         | 0.1541/10.0 [00:11<12:09, 74.07s/it]
 80%|████████  | 7.99595/10.0 [00:11<00:02,  1.43s/it]
 80%|████████  | 7.99595/10.0 [00:11<00:02,  1.43s/it]
100%|██████████| 10.0/10.0 [00:11<00:00,  1.14s/it]
100%|██████████| 10.0/10.0 [00:11<00:00,  1.14s/it]
```

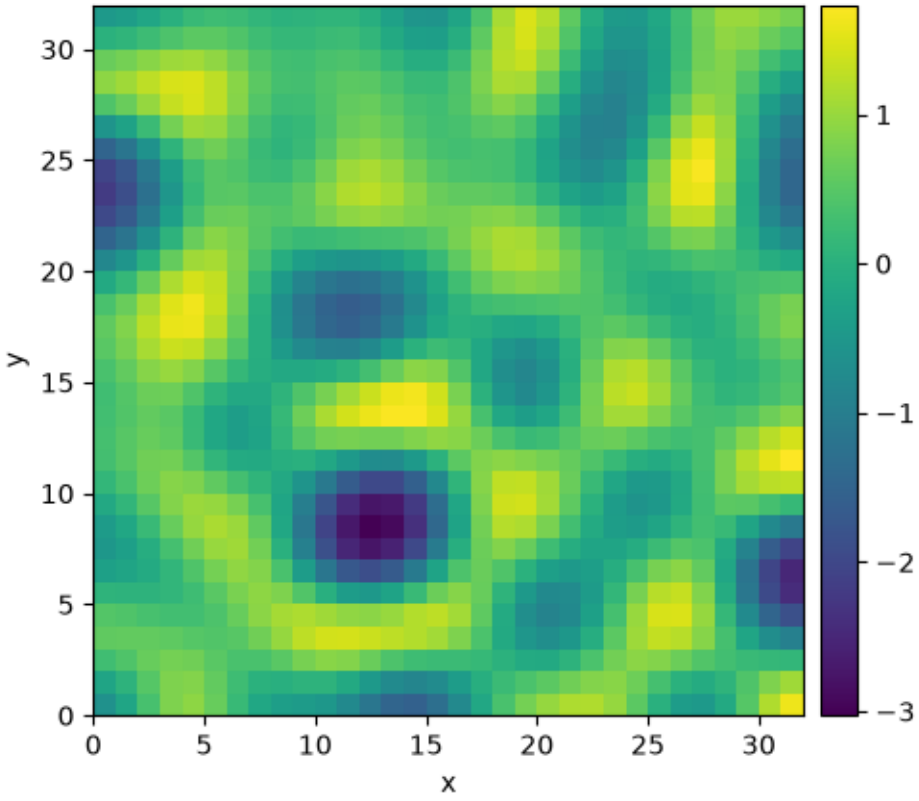
```python
from pde import DiffusionPDE, ScalarField, UnitGrid

grid = UnitGrid([64, 64])  # generate grid
state = ScalarField.random_uniform(grid, 0.2, 0.3)  # generate initial condition

eq = DiffusionPDE(diffusivity=0.1)  # define the pde
result = eq.solve(state, t_range=10)
result.plot()
```

Total running time of the script: (0 minutes 11.513 seconds)

2.2.4 Kuramoto-Sivashinsky - Using *PDE* class

This example implements a scalar PDE using the *PDE*. We here consider the Kuramoto–Sivashinsky equation, which for instance describes the dynamics of flame fronts:

$$\partial_t u = -\frac{1}{2}|\nabla u|^2 - \nabla^2 u - \nabla^4 u$$



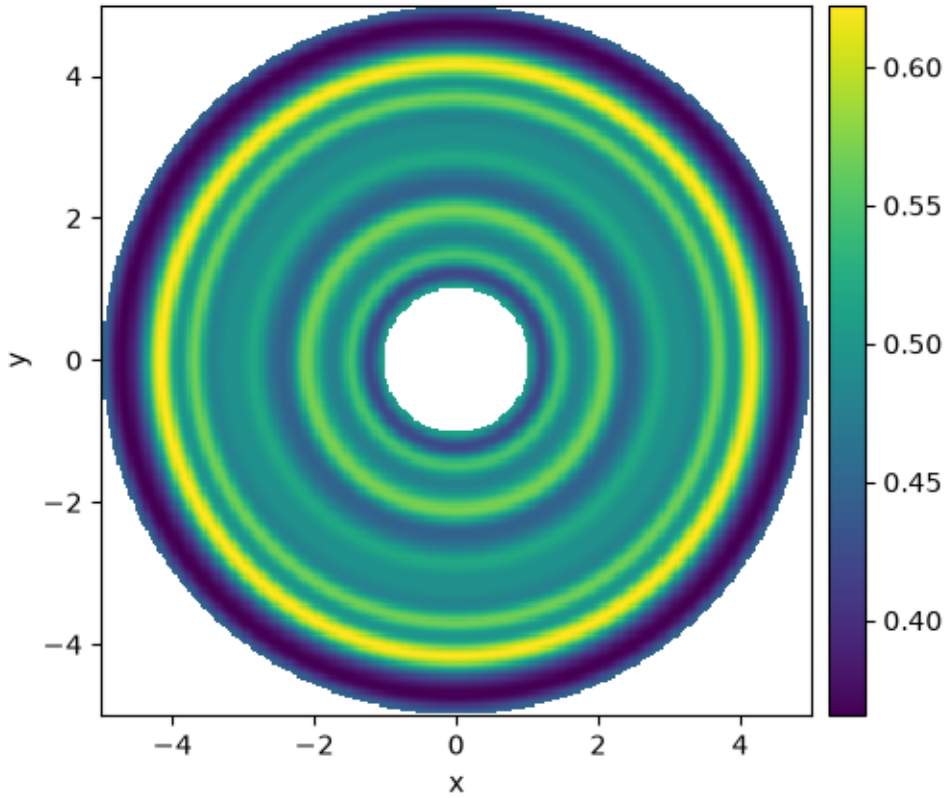

```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:11<?, ?it/s]
  0%|          | 0.01/10.0 [00:11<3:07:15, 1124.64s/it]
  2%|▏         | 0.21/10.0 [00:11<08:44, 53.56s/it]
  2%|▏         | 0.21/10.0 [00:11<08:44, 53.61s/it]
100%|██████████| 10.0/10.0 [00:11<00:00,  1.13s/it]
100%|██████████| 10.0/10.0 [00:11<00:00,  1.13s/it]
```

```
from pde import PDE, ScalarField, UnitGrid

grid = UnitGrid([32, 32])  # generate grid
state = ScalarField.random_uniform(grid)  # generate initial condition

eq = PDE({"u": "-gradient_squared(u) / 2 - laplace(u + laplace(u))"})  # define the␣
↪pde
result = eq.solve(state, t_range=10, dt=0.01)
result.plot()
```

Total running time of the script: (0 minutes 11.367 seconds)

2.2.5 Spherically symmetric PDE

This example illustrates how to solve a PDE in a spherically symmetric geometry.



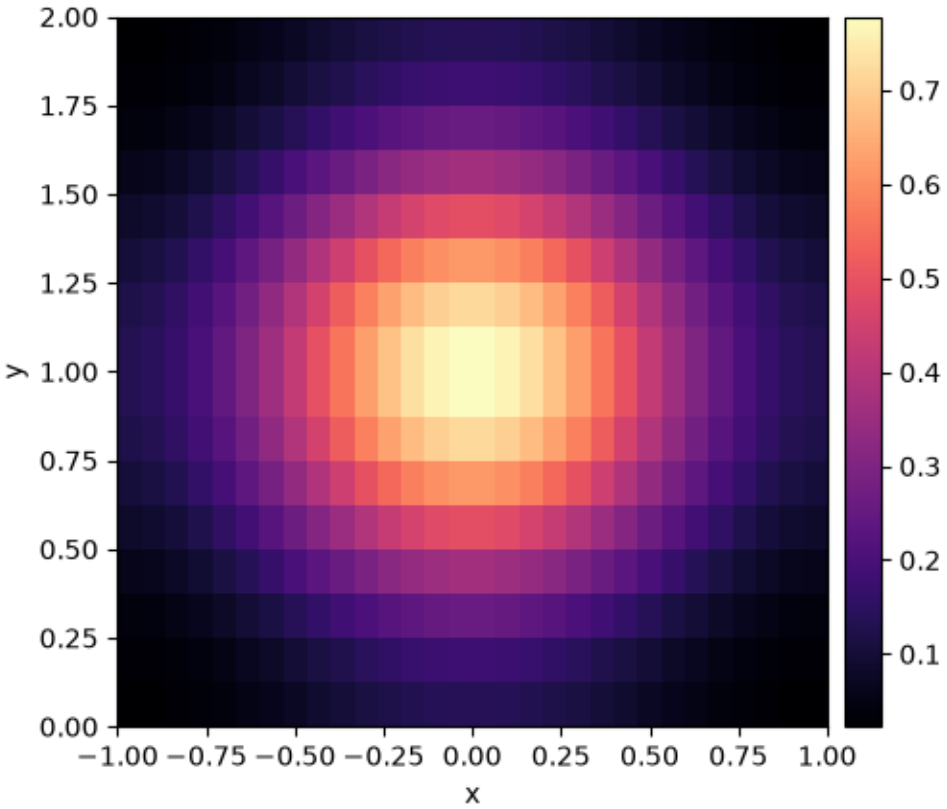

```
  0%|          | 0/0.1 [00:00<?, ?it/s]
Initializing:   0%|          | 0/0.1 [00:00<?, ?it/s]
  0%|          | 0/0.1 [00:03<?, ?it/s]
  6%|▌         | 0.006/0.1 [00:03<00:47, 505.56s/it]
  6%|▌         | 0.006/0.1 [00:03<00:47, 505.60s/it]
100%|██████████| 0.1/0.1 [00:03<00:00, 30.34s/it]
100%|██████████| 0.1/0.1 [00:03<00:00, 30.34s/it]
```

```
from pde import DiffusionPDE, ScalarField, SphericalSymGrid

grid = SphericalSymGrid(radius=[1, 5], shape=128)  # generate grid
state = ScalarField.random_uniform(grid)  # generate initial condition

eq = DiffusionPDE(0.1)  # define the PDE
result = eq.solve(state, t_range=0.1, dt=0.001)

result.plot(kind="image")
```

Total running time of the script: (0 minutes 3.322 seconds)

2.2.6 Diffusion on a Cartesian grid

This example shows how to solve the diffusion equation on a Cartesian grid.




```
  0%|          | 0/1.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/1.0 [00:00<?, ?it/s]
  0%|          | 0/1.0 [00:06<?, ?it/s]
  1%|          | 0.01/1.0 [00:06<10:40, 646.70s/it]
```

(continues on next page)

(continued from previous page)

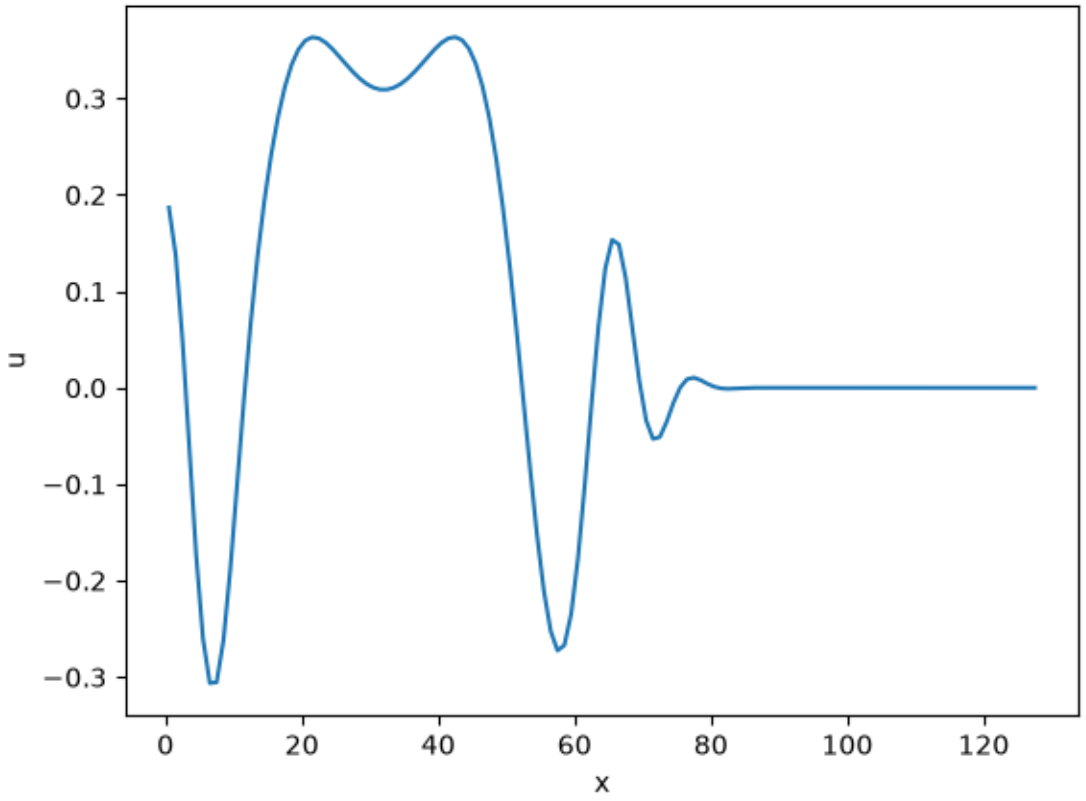
```
 27%|███        | 0.27/1.0 [00:06<00:17, 23.95s/it]
 27%|███        | 0.27/1.0 [00:06<00:17, 23.95s/it]
100%|███████████| 1.0/1.0 [00:06<00:00,  6.47s/it]
100%|███████████| 1.0/1.0 [00:06<00:00,  6.47s/it]
```

```
from pde import CartesianGrid, DiffusionPDE, ScalarField

grid = CartesianGrid([[-1, 1], [0, 2]], [30, 16])  # generate grid
state = ScalarField(grid)  # generate initial condition
state.insert([0, 1], 1)

eq = DiffusionPDE(0.1)  # define the pde
result = eq.solve(state, t_range=1, dt=0.01)
result.plot(cmap="magma")
```

Total running time of the script: (0 minutes 6.570 seconds)

2.2.7 Klein-Gordon equation

This example solves the Klein-Gordon equation in one dimension, showing the effect of the mass term on wave propagation. With mass=0 the equation reduces to the standard wave equation; increasing the mass introduces dispersion.



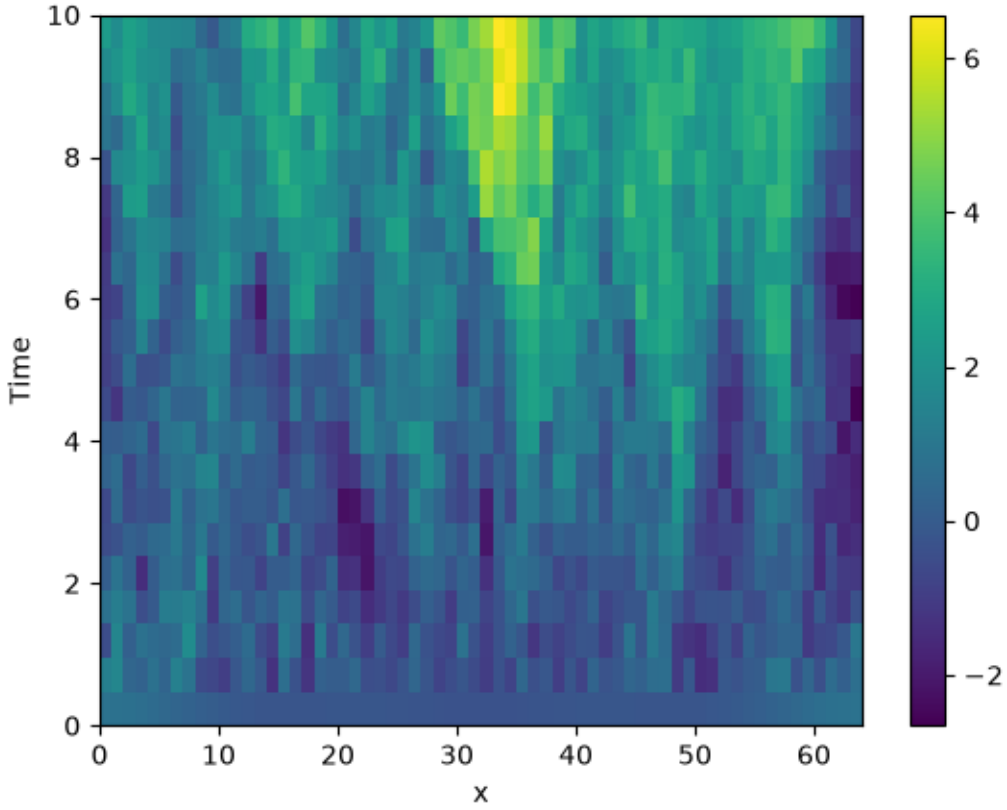

```
  0%|          | 0/50.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/50.0 [00:00<?, ?it/s]
  0%|          | 0/50.0 [00:03<?, ?it/s]
  0%|          | 0.01/50.0 [00:03<4:19:18, 311.23s/it]
  1%|          | 0.39/50.0 [00:03<06:35,  7.98s/it]
 53%|█████     | 26.66/50.0 [00:03<00:02,  8.56it/s]
 53%|█████     | 26.66/50.0 [00:03<00:02,  8.55it/s]
100%|██████████| 50.0/50.0 [00:03<00:00, 16.04it/s]
100%|██████████| 50.0/50.0 [00:03<00:00, 16.04it/s]
```

```python
from pde import KleinGordonPDE, ScalarField, UnitGrid

grid = UnitGrid([128])  # generate grid
u = ScalarField.from_expression(grid, "exp(-((x - 32) / 5) ** 2)")

eq = KleinGordonPDE(speed=1, mass=0.5)  # define the pde
state = eq.get_initial_condition(u)
result = eq.solve(state, t_range=50, dt=0.01)
result[0].plot()
```

Total running time of the script: (0 minutes 3.239 seconds)

2.2.8 Stochastic simulation

This example illustrates how a stochastic simulation can be done.

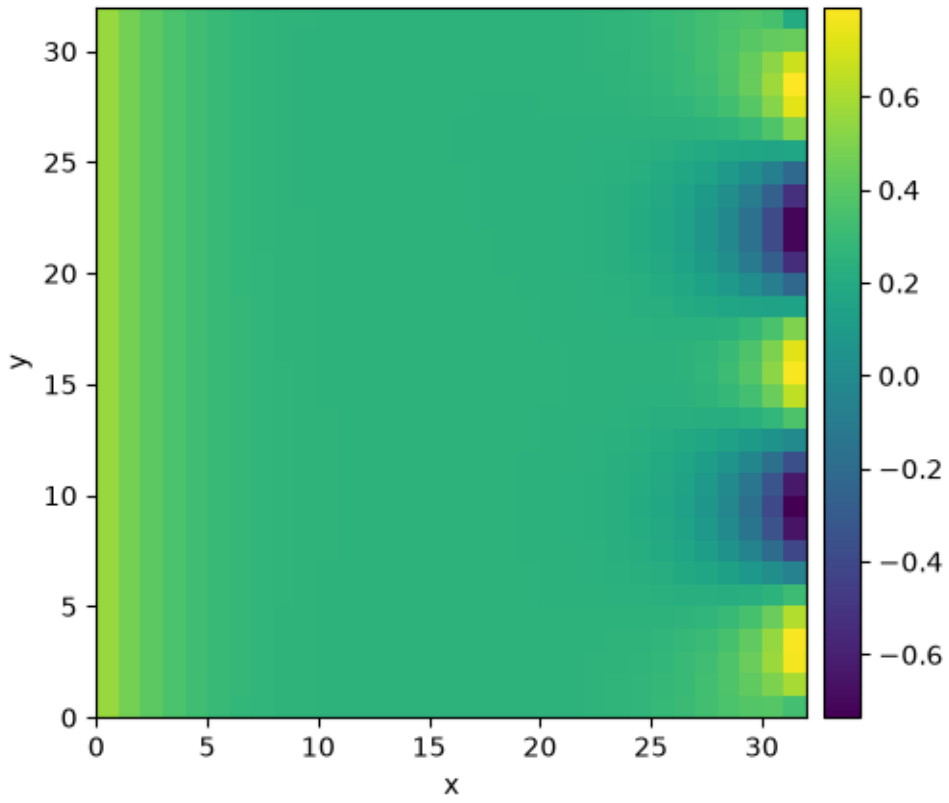



```
from pde import KPZInterfacePDE, MemoryStorage, ScalarField, UnitGrid, plot_kymograph

grid = UnitGrid([64])  # generate grid
state = ScalarField.random_harmonic(grid)  # generate initial condition

eq = KPZInterfacePDE(noise=1)  # define the SDE
storage = MemoryStorage()
eq.solve(state, t_range=10, dt=0.01, tracker=storage.tracker(0.5))
plot_kymograph(storage)
```

Total running time of the script: (0 minutes 5.715 seconds)

2.2.9 Setting boundary conditions

This example shows how different boundary conditions can be specified.




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:06<?, ?it/s]
  0%|          | 0.005/10.0 [00:06<3:41:17, 1328.46s/it]
  3%|▎         | 0.255/10.0 [00:06<04:13, 26.05s/it]
  3%|▎         | 0.255/10.0 [00:06<04:14, 26.07s/it]
100%|██████████| 10.0/10.0 [00:06<00:00,  1.50it/s]
100%|██████████| 10.0/10.0 [00:06<00:00,  1.50it/s]
```

```python
from pde import DiffusionPDE, ScalarField, UnitGrid

grid = UnitGrid([32, 32], periodic=[False, True])  # generate grid
state = ScalarField.random_uniform(grid, 0.2, 0.3)  # generate initial condition

# set boundary conditions `bc` for all axes
eq = DiffusionPDE(
    bc={"x-": {"derivative": 0.1}, "x+": {"value": "sin(y / 2)"}, "y": "periodic"}
)
```

(continues on next page)

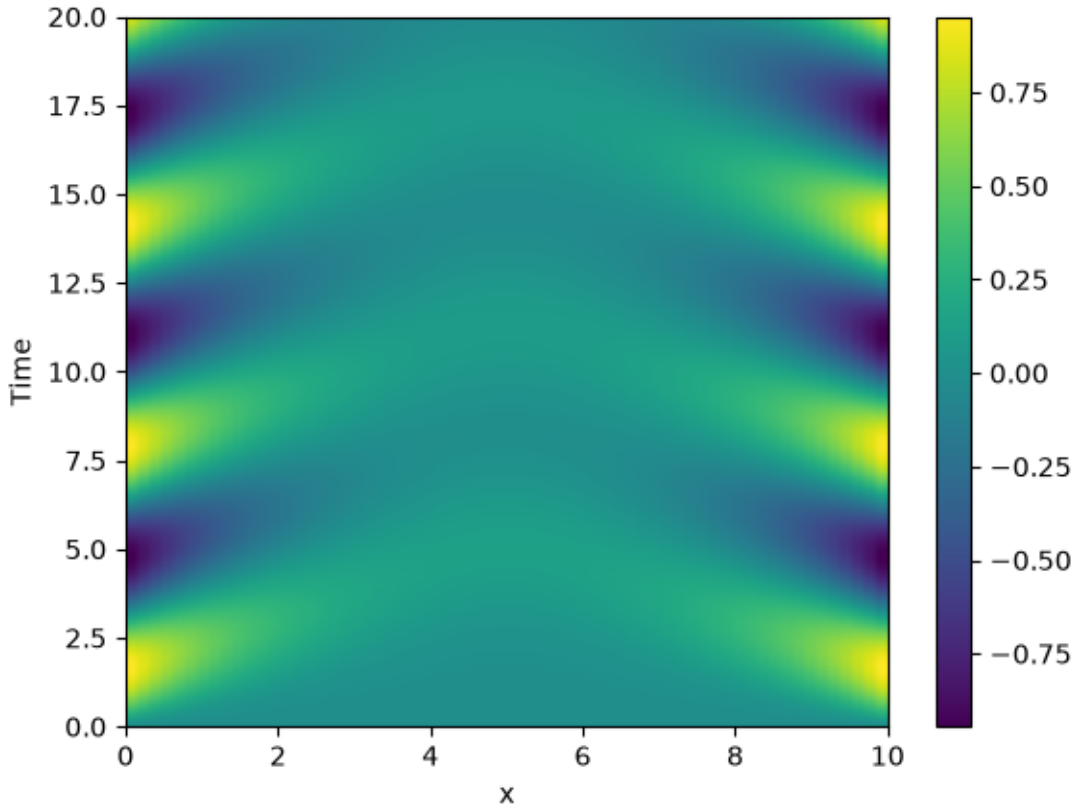
(continued from previous page)

```
result = eq.solve(state, t_range=10, dt=0.005)
result.plot()
```

Total running time of the script: (0 minutes 6.741 seconds)

2.2.10 Time-dependent boundary conditions

This example solves a simple diffusion equation in one dimensions with time-dependent boundary conditions.




```
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

grid = CartesianGrid([[0, 10]], [64])  # generate grid
state = ScalarField(grid)  # generate initial condition

eq = PDE({"c": "laplace(c)"}, bc={"value_expression": "sin(t)"})

storage = MemoryStorage()
eq.solve(state, t_range=20, dt=1e-4, tracker=storage.tracker(0.1))

# plot the trajectory as a space-time plot
plot_kymograph(storage)
```

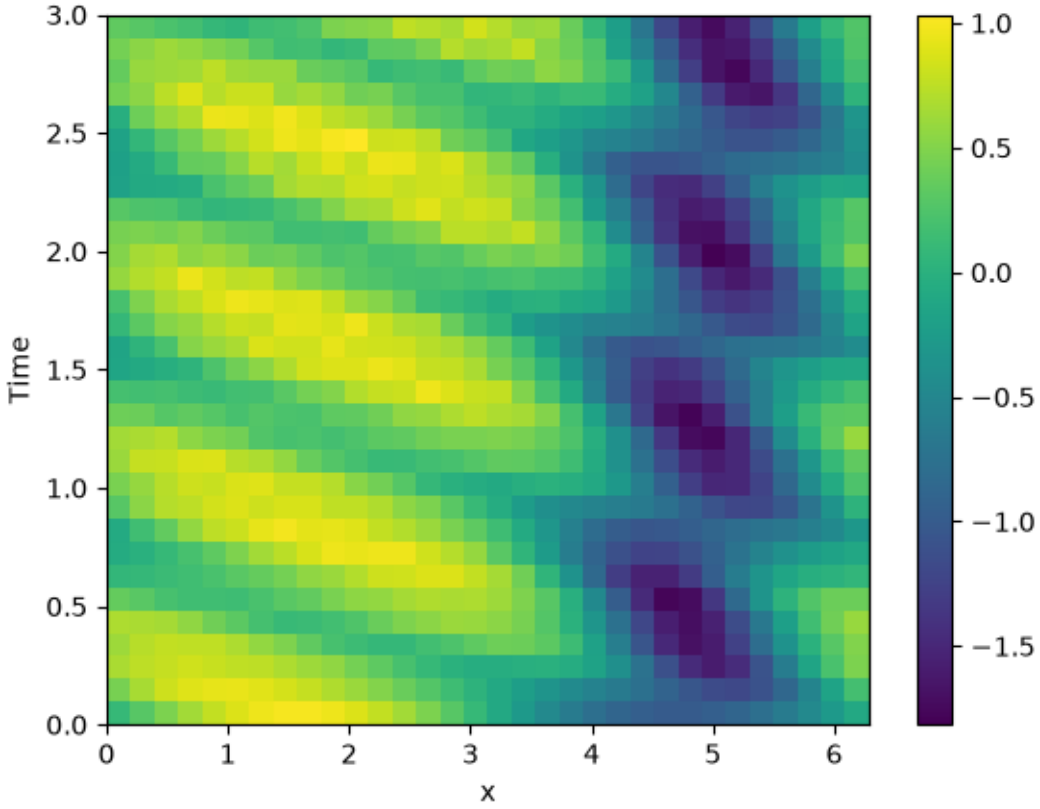
Total running time of the script: (0 minutes 5.615 seconds)

2.2.11 1D problem - Using *PDE* class

This example implements a PDE that is only defined in one dimension. Here, we chose the Korteweg-de Vries equation, given by

$$\partial_t \phi = 6\phi\partial_x\phi - \partial_x^3\phi$$

which we implement using the *PDE*.




```
from math import pi

from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

# initialize the equation and the space
eq = PDE({"ψ": "6 * ψ * d_dx(ψ) - laplace(d_dx(ψ))"})
grid = CartesianGrid([[0, 2 * pi]], [32], periodic=True)
state = ScalarField.from_expression(grid, "sin(x)")

# solve the equation and store the trajectory
storage = MemoryStorage()
eq.solve(state, t_range=3, solver="scipy", tracker=storage.tracker(0.1))

# plot the trajectory as a space-time plot
plot_kymograph(storage)
```

Total running time of the script: (0 minutes 4.907 seconds)

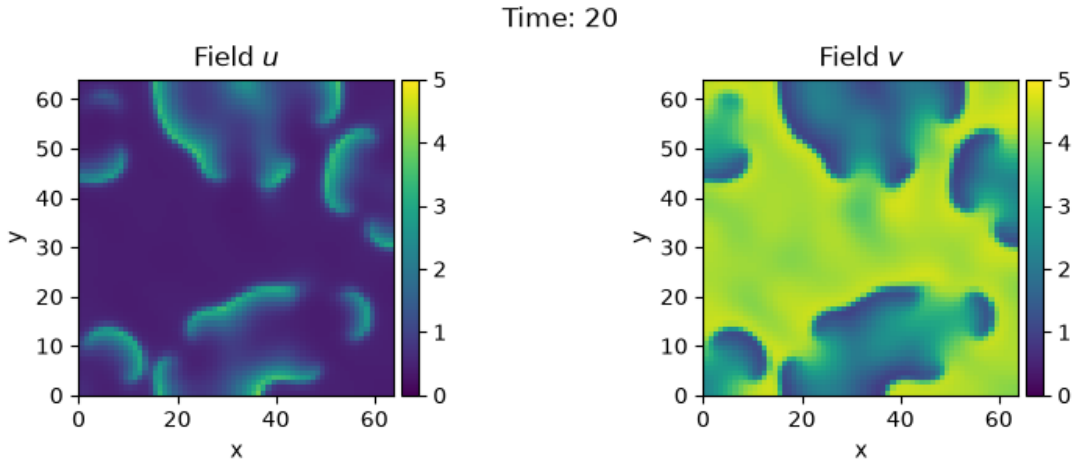
2.2.12 Brusselator - Using the *PDE* class

This example uses the *PDE* class to implement the Brusselator with spatial coupling,

$$\partial_t u = D_0 \nabla^2 u + a - (1 + b)u + vu^2$$
$$\partial_t v = D_1 \nabla^2 v + bu - vu^2$$

Here, D_0 and D_1 are the respective diffusivity and the parameters a and b are related to reaction rates.

Note that the same result can also be achieved with a *full implementation of a custom class*, which allows for more flexibility at the cost of code complexity.




```
from pde import PDE, FieldCollection, PlotTracker, ScalarField, UnitGrid

# define the PDE
a, b = 1, 3
d0, d1 = 1, 0.1
eq = PDE(
    {
        "u": f"{d0} * laplace(u) + {a} - ({b} + 1) * u + u**2 * v",
        "v": f"{d1} * laplace(v) + {b} * u - u**2 * v",
    }
)

# initialize state
grid = UnitGrid([64, 64])
u = ScalarField(grid, a, label="Field $u$")
v = b / a + 0.1 * ScalarField.random_normal(grid, label="Field $v$")
state = FieldCollection([u, v])

# simulate the pde
tracker = PlotTracker(interrupts=1, plot_args={"vmin": 0, "vmax": 5})
sol = eq.solve(state, t_range=20, dt=1e-3, tracker=tracker)
```

Total running time of the script: (0 minutes 14.133 seconds)

2.2.13 Diffusion equation with spatial dependence

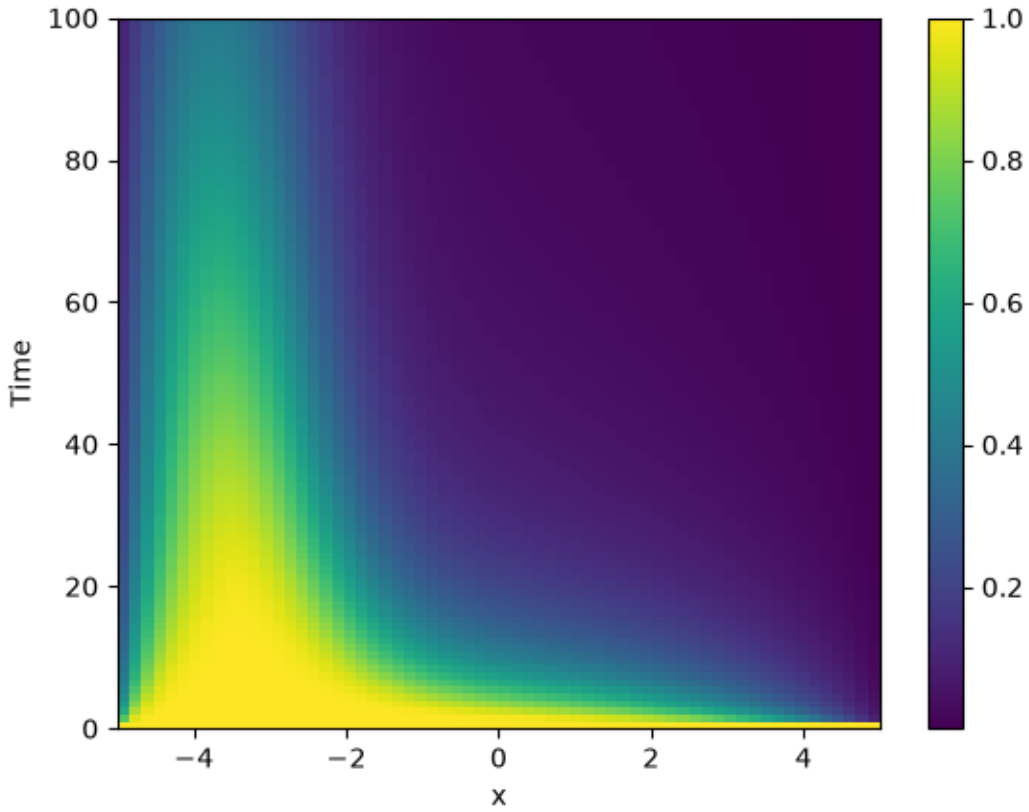
This example solve the Diffusion equation with a heterogeneous diffusivity:

$$\partial_t c = \nabla\big(D(\boldsymbol{r})\nabla c\big)$$

using the *PDE* class. In particular, we consider $D(x) = 1.01 + \tanh(x)$, which gives a low diffusivity on the left side of the domain.

Note that the naive implementation, `PDE({"c": "divergence((1.01 + tanh(x)) * gradient(c))"})`, has numerical instabilities. This is because two finite difference approximations are nested. To arrive at a more stable numerical scheme, it is advisable to expand the divergence,

$$\partial_t c = D\nabla^2 c + \nabla D.\nabla c$$




```
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

# Expanded definition of the PDE
diffusivity = "1.01 + tanh(x)"
term_1 = f"({diffusivity}) * laplace(c)"
term_2 = f"dot(gradient({diffusivity}), gradient(c))"
eq = PDE({"c": f"{term_1} + {term_2}"}, bc={"value": 0})


grid = CartesianGrid([[-5, 5]], 64)  # generate grid
```

(continues on next page)

(continued from previous page)

```
field = ScalarField(grid, 1)  # generate initial condition

storage = MemoryStorage()  # store intermediate information of the simulation
res = eq.solve(field, 100, dt=1e-3, tracker=storage.tracker(1))  # solve the PDE

plot_kymograph(storage)  # visualize the result in a space-time plot
```

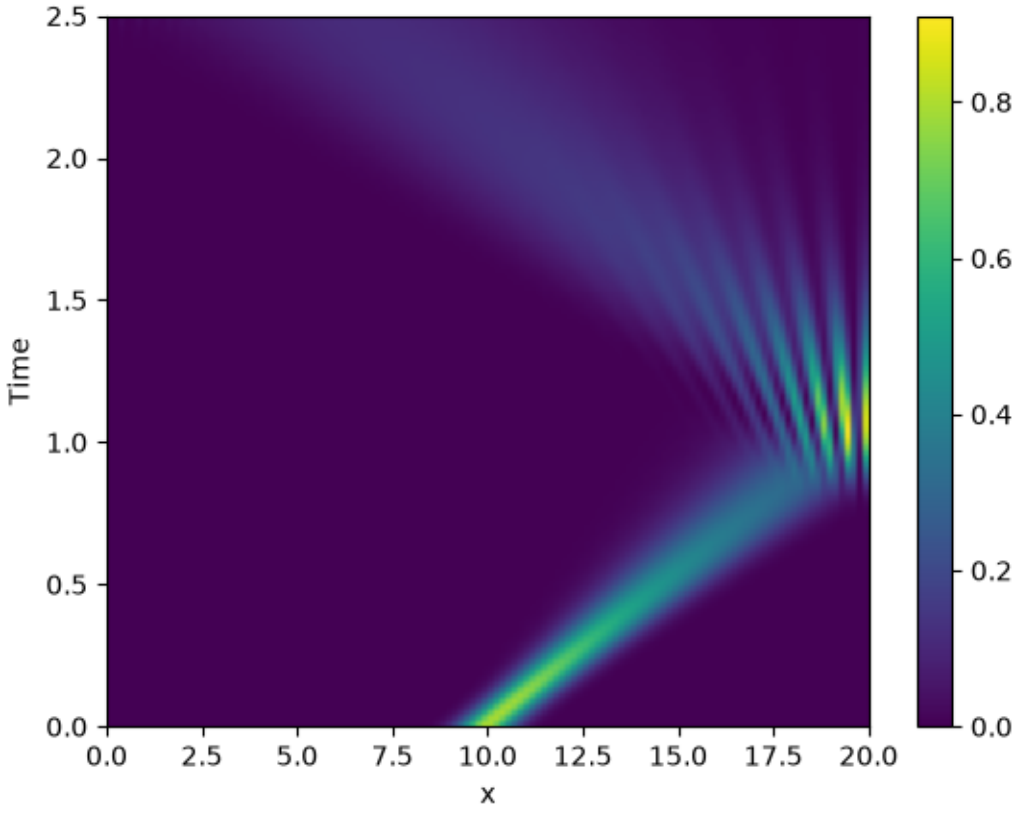
Total running time of the script: (0 minutes 9.980 seconds)

2.2.14 Schrödinger's Equation

This example implements a complex PDE using the *PDE*. We here chose the Schrödinger equation without a spatial potential in non-dimensional form:

$$i\partial_t \psi = -\nabla^2 \psi$$

Note that the example imposes Neumann conditions at the wall, so the wave packet is expected to reflect off the wall.




```
from math import sqrt

from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

grid = CartesianGrid([[0, 20]], 128, periodic=False)  # generate grid
```

(continues on next page)

(continued from previous page)

```
# create a (normalized) wave packet with a certain form as an initial condition
initial_state = ScalarField.from_expression(grid, "exp(I * 5 * x) * exp(-(x - 10)**2)
↪")
initial_state /= sqrt(initial_state.to_scalar("norm_squared").integral.real)

eq = PDE({"ψ": "I * laplace(ψ)"})  # define the pde

# solve the pde and store intermediate data
storage = MemoryStorage()
eq.solve(initial_state, t_range=2.5, dt=1e-5, tracker=[storage.tracker(0.02)])

# visualize the results as a space-time plot
plot_kymograph(storage, scalar="norm_squared")
```

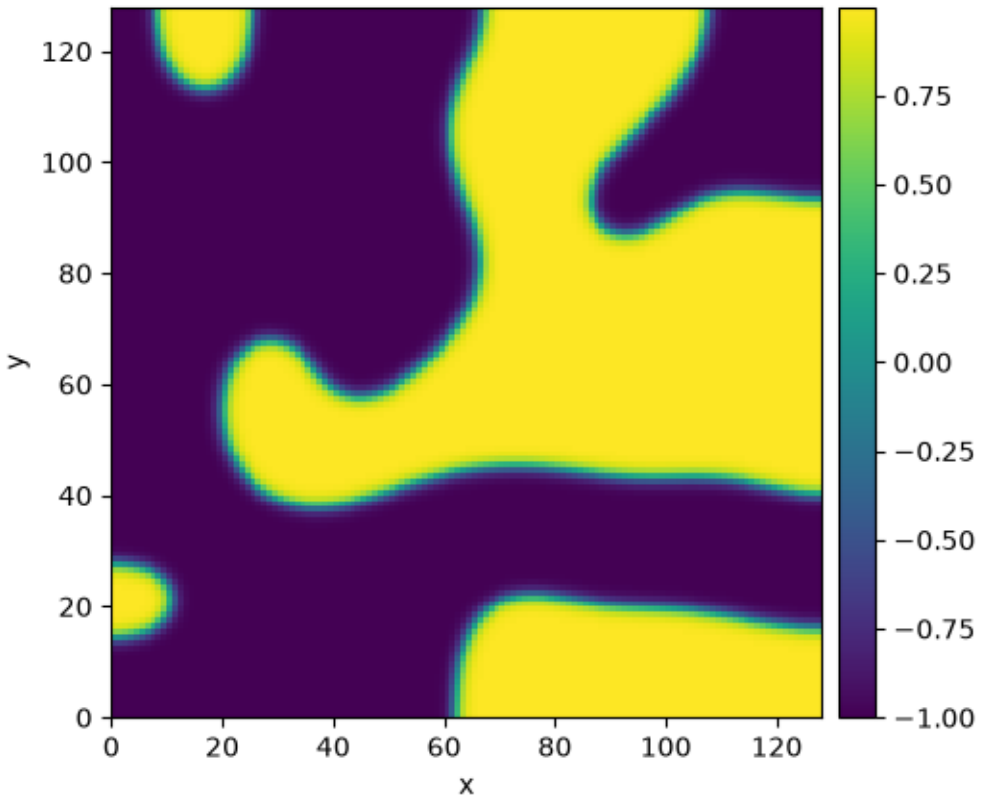
Total running time of the script: (0 minutes 4.166 seconds)

2.3 Output and analysis

These examples demonstrate how to store and analyze output.

2.3.1 Storage examples

This example shows how to use *storage* to store data persistently.

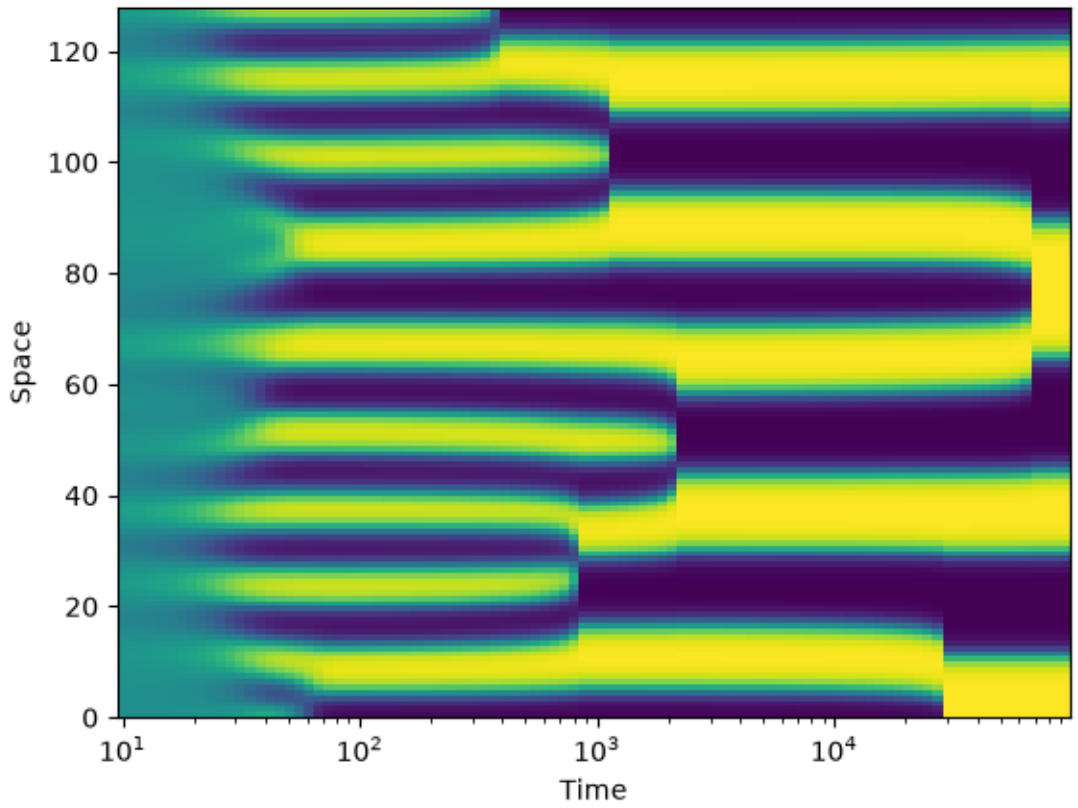



```
from pde import AllenCahnPDE, FileStorage, MovieStorage, ScalarField, UnitGrid

# initialize the model
state = ScalarField.random_uniform(UnitGrid([128, 128]), -0.01, 0.01)
eq = AllenCahnPDE()

# initialize empty storages
file_write = FileStorage("allen_cahn.hdf")
movie_write = MovieStorage("allen_cahn.avi", vmin=-1, vmax=1)

# store trajectory in storage
final_state = eq.solve(
    state,
    t_range=100,
    adaptive=True,
    tracker=[file_write.tracker(2), movie_write.tracker(1)],
)

# read storage and plot last frame
movie_read = MovieStorage("allen_cahn.avi")
movie_read[-1].plot()
```

Total running time of the script: (0 minutes 20.775 seconds)

2.3.2 Logarithmic kymograph

This example demonstrates a space-time plot with a logarithmic time axis (using `utilitiez.densityplot()`), which is useful to analyze coarsening processes.




```
import matplotlib.pyplot as plt
from utilitiez import densityplot

import pde

# define grid, initial field, and the PDE
grid = pde.UnitGrid([128])
field = pde.ScalarField.random_uniform(grid, -0.1, 0.1)
eq = pde.CahnHilliardPDE(interface_width=2)

# run the simulation and store data in logarithmically spaced time intervals
storage = pde.MemoryStorage()
res = eq.solve(
    field, t_range=1e5, adaptive=True, tracker=[storage.tracker("geometric(10, 1.1)")]
)

# create the density plot, which detects the logarithmically scaled time
densityplot(storage.data, storage.times, grid.axes_coords[0])
plt.xlabel("Time")
```

(continues on next page)

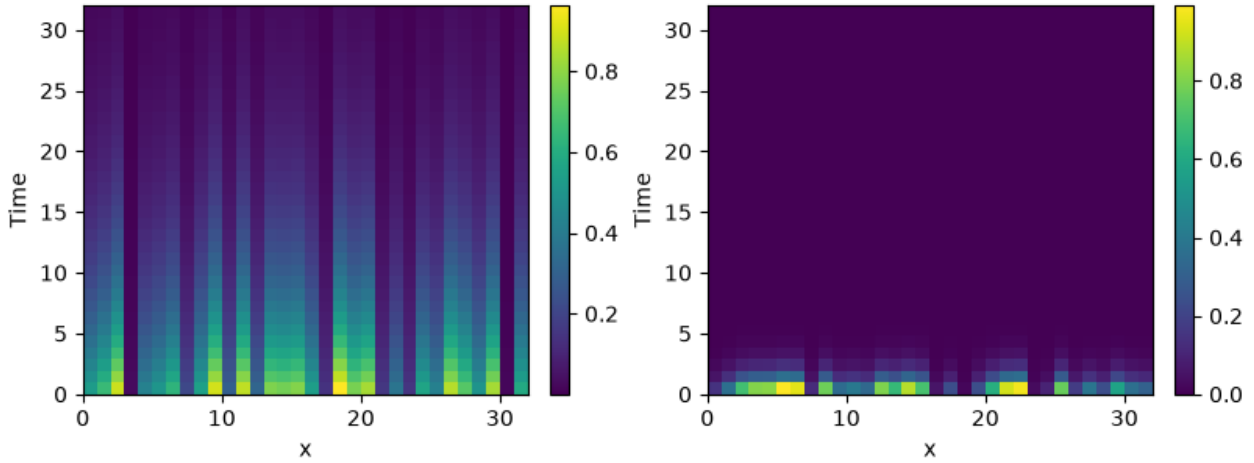
(continued from previous page)

```
plt.ylabel("Space")
```

Total running time of the script: (0 minutes 7.898 seconds)

2.3.3 Writing and reading trajectory data

This example illustrates how to store intermediate data to a file for later post-processing. The storage frequency is an argument to the tracker.

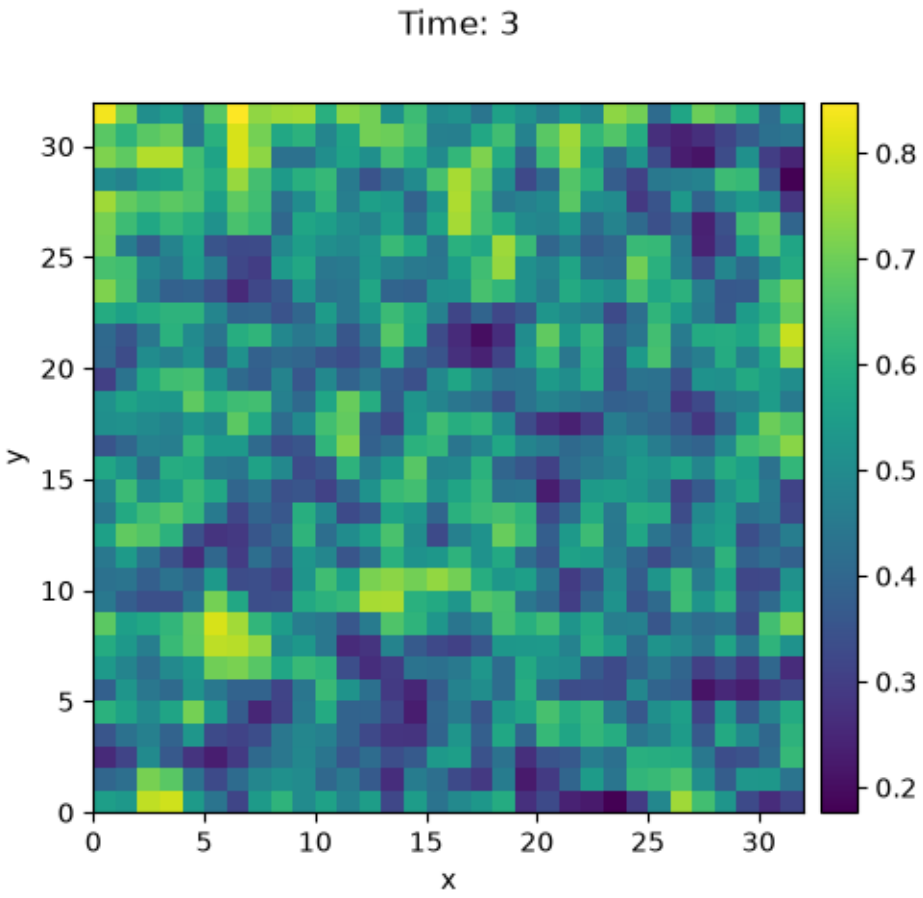



```
from tempfile import NamedTemporaryFile

import pde

# define grid, state and pde
grid = pde.UnitGrid([32])
state = pde.FieldCollection(
    [pde.ScalarField.random_uniform(grid), pde.VectorField.random_uniform(grid)]
)
eq = pde.PDE({"s": "-0.1 * s", "v": "-v"})

# get a temporary file to write data to
with NamedTemporaryFile(suffix=".hdf5") as path:
    # run a simulation and write the results
    writer = pde.FileStorage(path.name, write_mode="truncate")
    eq.solve(state, t_range=32, dt=0.01, tracker=writer.tracker(1))

    # read the simulation back in again
    reader = pde.FileStorage(path.name, write_mode="read_only")
    pde.plot_kymographs(reader)
```

Total running time of the script: (0 minutes 4.398 seconds)

2.3.4 Using simulation trackers

This example illustrates how trackers can be used to analyze simulations.




```
  0%|          | 0/3.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/3.0 [00:00<?, ?it/s]
  0%|          | 0/3.0 [00:02<?, ?it/s]
  3%|▌         | 0.1/3.0 [00:02<01:17, 26.87s/it]
  7%|▊         | 0.2/3.0 [00:02<00:37, 13.44s/it]
 50%|█████     | 1.5/3.0 [00:02<00:02,  1.83s/it]
 50%|█████     | 1.5/3.0 [00:02<00:02,  1.89s/it]
100%|██████████| 3.0/3.0 [00:02<00:00,  1.06it/s]
100%|██████████| 3.0/3.0 [00:02<00:00,  1.06it/s]
503.6885582029231
503.6885582029231
503.6885582029231
503.6885582029231
```

```
import pde

grid = pde.UnitGrid([32, 32])  # generate grid
state = pde.ScalarField.random_uniform(grid)  # generate initial condition
```

(continues on next page)

(continued from previous page)

```
storage = pde.MemoryStorage()

trackers = [
    "progress",  # show progress bar during simulation
    "steady_state",  # abort when steady state is reached
    storage.tracker(interrupts=1),  # store data every simulation time unit
    pde.PlotTracker(show=True),  # show images during simulation
    # print some output every 5 real seconds:
    pde.PrintTracker(interrupts=pde.RealtimeInterrupts(duration=5)),
]

eq = pde.DiffusionPDE(0.1)  # define the PDE
eq.solve(state, 3, dt=0.1, tracker=trackers)

for field in storage:
    print(field.integral)
```

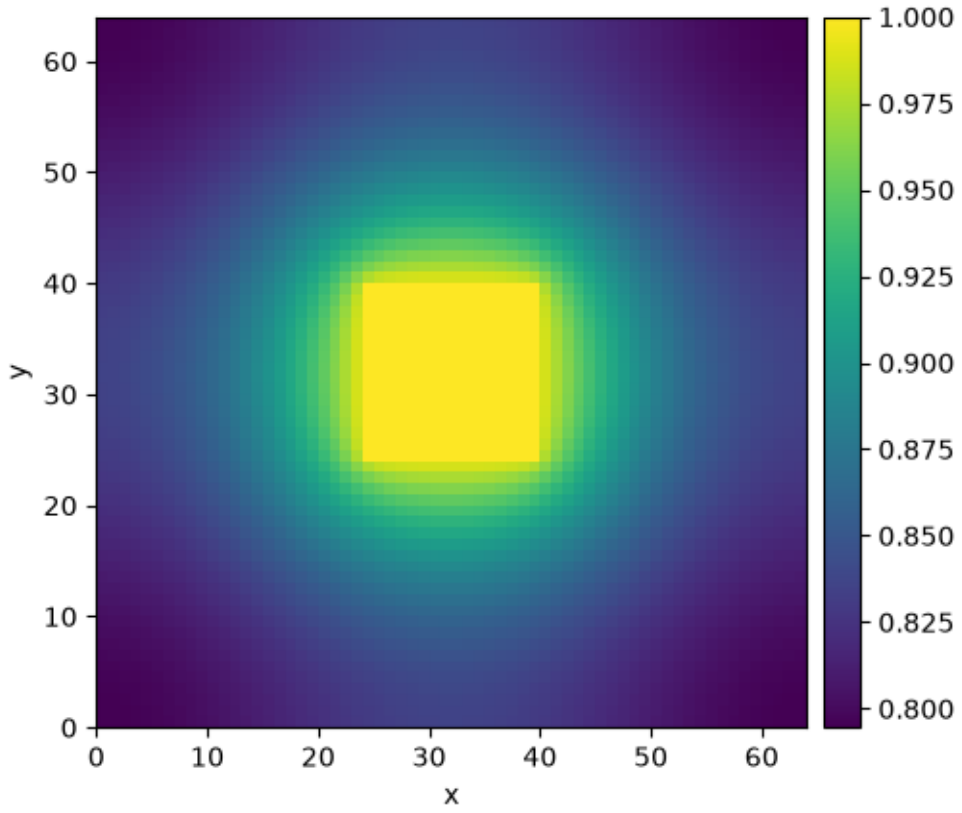
Total running time of the script: (0 minutes 2.894 seconds)

2.4 Advanced PDEs

These examples demonstrate more advanced usage of the package.

2.4.1 Post-step hook function

Demonstrate the simple hook function in `PDE`, which is called after each time step and may modify the state and abort the simulation.




```
  0%|          | 0/10000.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10000.0 [00:00<?, ?it/s]
  0%|          | 0/10000.0 [00:14<?, ?it/s]
  0%|          | 0.1/10000.0 [00:14<401:34:56, 144.57s/it]
  0%|          | 0.2/10000.0 [00:14<200:47:31, 72.29s/it]
  0%|          | 13.0/10000.0 [00:14<3:05:07,  1.11s/it]
  3%|▎         | 335.8/10000.0 [00:14<06:57, 23.17it/s]
  3%|▎         | 335.8/10000.0 [00:14<06:59, 23.05it/s]
  3%|▎         | 335.8/10000.0 [00:14<06:59, 23.05it/s]
```

```python
from pde import PDE, ScalarField, UnitGrid


def post_step_hook(state_data, t):
    """Helper function called after every time step."""
    state_data[24:40, 24:40] = 1  # set central region to given value

    if t > 1e3:
        raise StopIteration  # abort simulation at given time
```

(continues on next page)

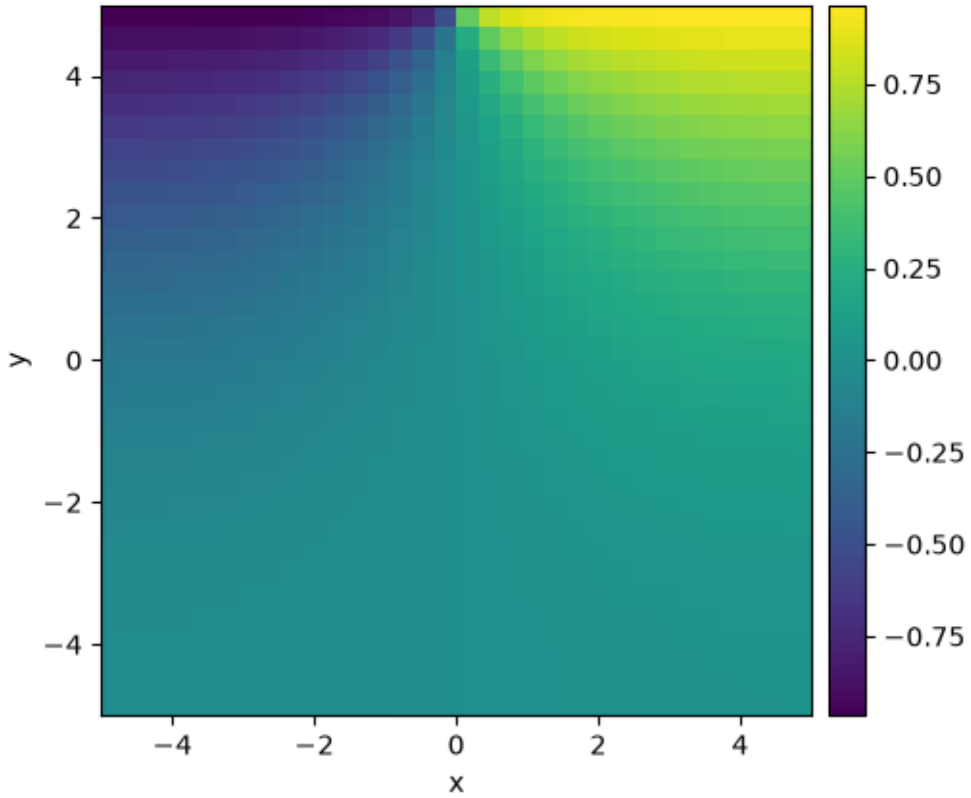
(continued from previous page)

```
    return state_data


eq = PDE({"c": "laplace(c)"}, post_step_hook=post_step_hook)
state = ScalarField(UnitGrid([64, 64]))
result = eq.solve(state, dt=0.1, t_range=1e4)
result.plot()
```

Total running time of the script: (0 minutes 14.707 seconds)

2.4.2 Heterogeneous boundary conditions

This example implements a diffusion equation with a boundary condition specified by a function, which can in principle depend on time.




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:00<?, ?it/s]
  3%|▌         | 0.31/10.0 [00:00<00:04,  2.41it/s]
 12%|▉         | 1.19/10.0 [00:00<00:01,  7.26it/s]
 58%|█████     | 5.83/10.0 [00:00<00:00, 15.42it/s]
 58%|█████     | 5.83/10.0 [00:00<00:00, 10.72it/s]
```

(continues on next page)

(continued from previous page)

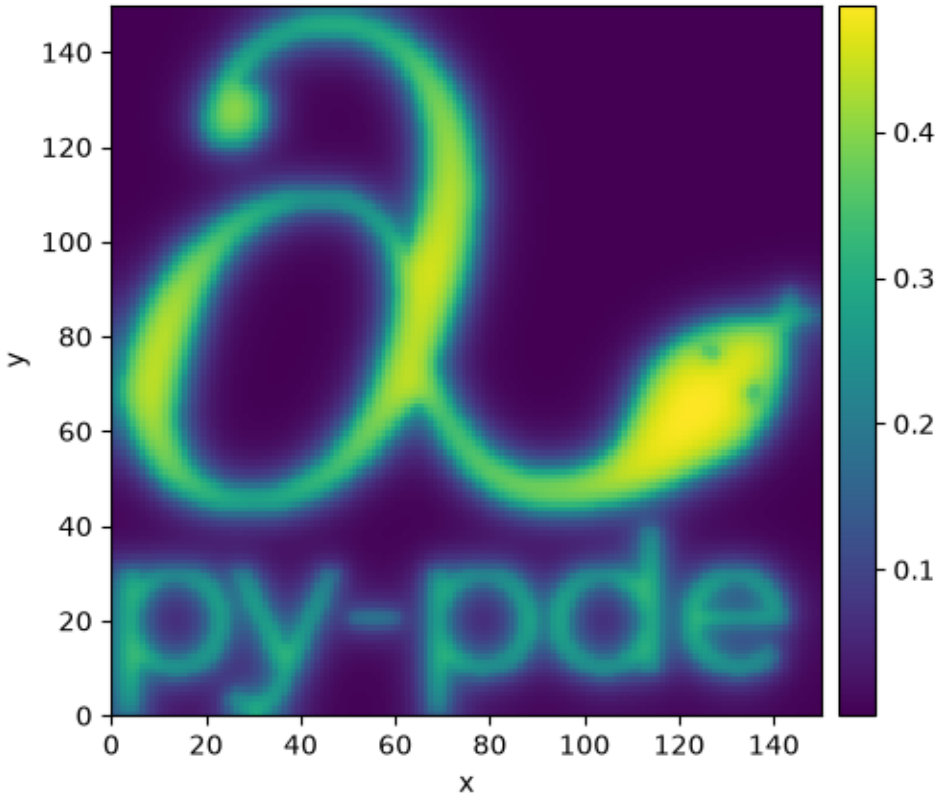
```
100%|██████████| 10.0/10.0 [00:00<00:00, 18.38it/s]
100%|██████████| 10.0/10.0 [00:00<00:00, 18.38it/s]
```

```python
import numpy as np

from pde import CartesianGrid, DiffusionPDE, ScalarField

# define grid and an initial state
grid = CartesianGrid([[-5, 5], [-5, 5]], 32)
field = ScalarField(grid)


# define the boundary conditions, which here are calculated from a function
def bc_value(adjacent_value, dx, x, y, t):
    """Return boundary value."""
    return np.sign(x)


# define and solve a simple diffusion equation
eq = DiffusionPDE(bc={"*": {"derivative": 0}, "y+": {"value_expression": bc_value}})
res = eq.solve(field, t_range=10, dt=0.01, backend="numpy")
res.plot()
```

Total running time of the script: (0 minutes 0.620 seconds)

2.4.3 Heterogeneous PDE

This example loads an example image and uses it as the source field for a simple reaction-diffusion equation.




```
  0%|          | 0/100.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/100.0 [00:00<?, ?it/s]
  0%|          | 0/100.0 [00:30<?, ?it/s]
  0%|          | 0.00316/100.0 [00:30<267:42:54, 9638.05s/it]
  0%|          | 0.10355/100.0 [00:30<8:09:43, 294.14s/it]
  2%|▏         | 2.4514/100.0 [00:30<20:12, 12.43s/it]
 32%|███       | 31.83796/100.0 [00:30<01:05,  1.04it/s]
 32%|███       | 31.83796/100.0 [00:30<01:05,  1.04it/s]
100%|██████████| 100.0/100.0 [00:30<00:00,  3.28it/s]
100%|██████████| 100.0/100.0 [00:30<00:00,  3.28it/s]
```

```python
import inspect
from pathlib import Path

from pde import PDE, ScalarField

# load a field relative to the current file
package_path = Path(inspect.getfile(lambda: None)).parents[2]
img_path = package_path / "docs" / "source" / "_images" / "logo_small.png"
```

(continues on next page)

(continued from previous page)

```
background = ScalarField.from_image(img_path)  # create source field from image
state = ScalarField(background.grid)  # generate initial condition

# define the pde
eq = PDE({"c": "laplace(c) + 0.2 * source - 0.1 * c"}, consts={"source": background})
result = eq.solve(state, t_range=100, adaptive=True)
result.plot()
```

Total running time of the script: (0 minutes 33.848 seconds)

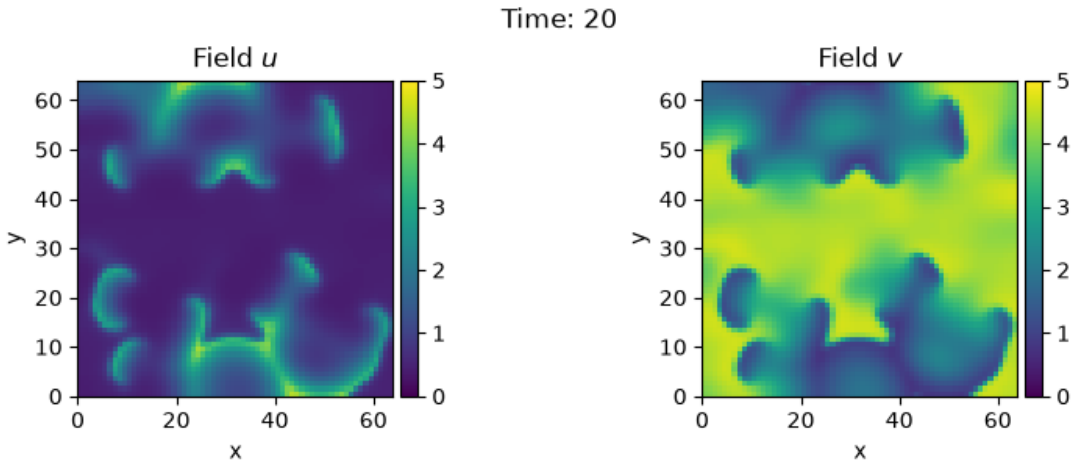
2.4.4 Brusselator - Using the *ReactionDiffusionPDE* class

This example uses the `ReactionDiffusionPDE` class to implement the Brusselator with spatial coupling,

$$\partial_t u = D_0 \nabla^2 u + a - (1 + b)u + vu^2$$
$$\partial_t v = D_1 \nabla^2 v + bu - vu^2$$

Here, D_0 and D_1 are the respective diffusivity and the parameters a and b are related to reaction rates.

Note that the PDE can also be implemented using the `PDE` class; see *the example*.




```
from pde import (
    FieldCollection,
    PlotTracker,
    ReactionDiffusionPDE,
    ScalarField,
    UnitGrid,
)

# define the PDE
a, b = 1, 3
d0, d1 = 1, 0.1
eq = ReactionDiffusionPDE(
    variables=["u", "v"],
    diffusivity=[d0, d1],
    sources=[f"{a} - ({b} + 1) * u + u**2 * v", f"{b} * u - u**2 * v"],
)
```

(continues on next page)

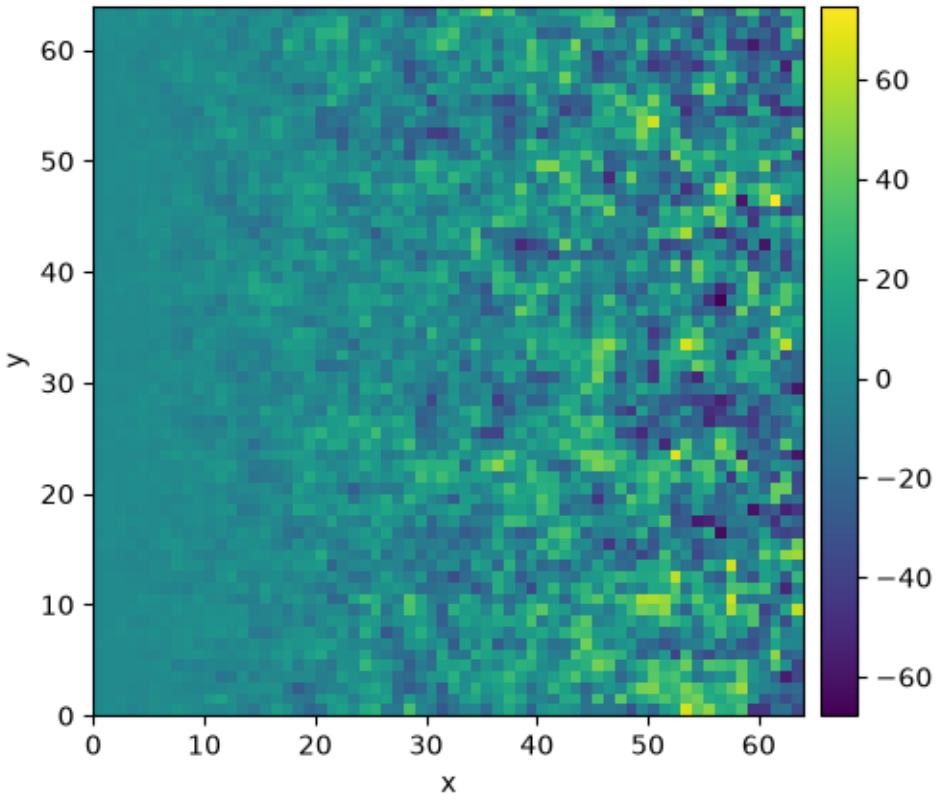
(continued from previous page)

```
# initialize state
grid = UnitGrid([64, 64])
u = ScalarField(grid, a, label="Field $u$")
v = b / a + 0.1 * ScalarField.random_normal(grid, label="Field $v$")
state = FieldCollection([u, v])

# simulate the pde
tracker = PlotTracker(interrupts=1, plot_args={"vmin": 0, "vmax": 5})
sol = eq.solve(state, t_range=20, dt=1e-3, tracker=tracker)
```

Total running time of the script: (0 minutes 15.199 seconds)

2.4.5 Custom noise

This example solves a diffusion equation with a custom noise.




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:04<?, ?it/s]
  0%|          | 0.01/10.0 [00:04<1:08:38, 412.27s/it]
  2%|          | 0.24/10.0 [00:04<02:47, 17.19s/it]
 40%|████      | 3.99/10.0 [00:04<00:06,  1.05s/it]
```

(continues on next page)

(continued from previous page)

```
 40%|████      | 3.99/10.0 [00:04<00:06,  1.07s/it]
100%|██████████| 10.0/10.0 [00:04<00:00,  2.33it/s]
100%|██████████| 10.0/10.0 [00:04<00:00,  2.33it/s]
```

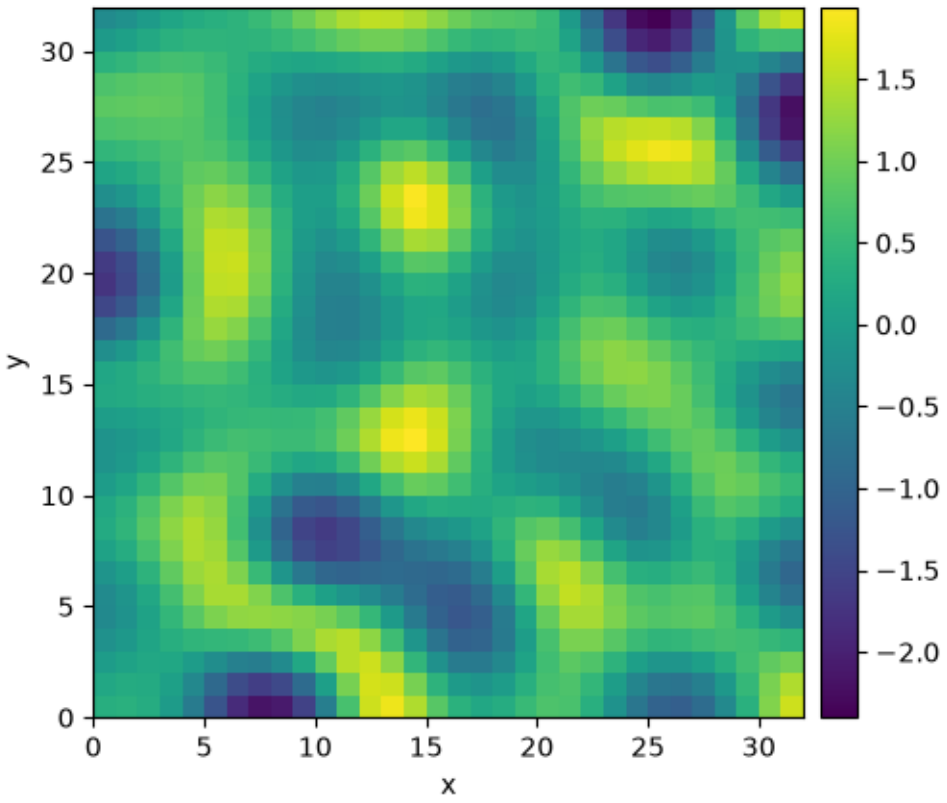
```python
from pde import DiffusionPDE, ScalarField, UnitGrid


class DiffusionCustomNoisePDE(DiffusionPDE):
    """Diffusion PDE with custom noise implementation."""

    use_noise_variance = True

    def make_noise_variance(self, state, *, backend, ret_diff=False):
        """Make function that calculates noise variance."""
        noise = float(self.noise)
        x_values = state.grid.cell_coords[..., 0]

        def noise_variance(state_data, t):
            return noise * x_values**2

        return noise_variance


eq = DiffusionCustomNoisePDE(diffusivity=0.1, noise=0.1)  # define the pde
state = ScalarField.random_uniform(UnitGrid([64, 64]))  # generate initial condition
result = eq.solve(state, t_range=10, dt=0.01)
result.plot()
```

Total running time of the script: (0 minutes 4.367 seconds)

2.4.6 Kuramoto-Sivashinsky - Using custom class

This example implements a scalar PDE using a custom class. We here consider the Kuramoto–Sivashinsky equation, which for instance describes the dynamics of flame fronts:

$$\partial_t u = -\frac{1}{2}|\nabla u|^2 - \nabla^2 u - \nabla^4 u$$




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:00<?, ?it/s]
  3%|▎         | 0.3/10.0 [00:00<00:08,  1.16it/s]
  9%|▉         | 0.9/10.0 [00:00<00:03,  2.97it/s]
 37%|███▋      | 3.7/10.0 [00:00<00:00,  7.25it/s]
 99%|█████████▉| 9.86/10.0 [00:00<00:00, 10.22it/s]
 99%|█████████▉| 9.86/10.0 [00:00<00:00, 10.11it/s]
100%|██████████| 10.0/10.0 [00:00<00:00, 10.25it/s]
100%|██████████| 10.0/10.0 [00:00<00:00, 10.25it/s]
```

```python
from pde import PDEBase, ScalarField, UnitGrid


class KuramotoSivashinskyPDE(PDEBase):
    """Implementation of the normalized Kuramoto-Sivashinsky equation."""

    def evolution_rate(self, state, t=0):
        """Implement the python version of the evolution equation."""
```

(continues on next page)

(continued from previous page)

```
        state_lap = state.laplace(bc="auto_periodic_neumann")
        state_lap2 = state_lap.laplace(bc="auto_periodic_neumann")
        state_grad = state.gradient(bc="auto_periodic_neumann")
        return -state_grad.to_scalar("squared_sum") / 2 - state_lap - state_lap2


grid = UnitGrid([32, 32])  # generate grid
state = ScalarField.random_uniform(grid)  # generate initial condition

eq = KuramotoSivashinskyPDE()  # define the pde
result = eq.solve(state, t_range=10, dt=0.01)
result.plot()
```

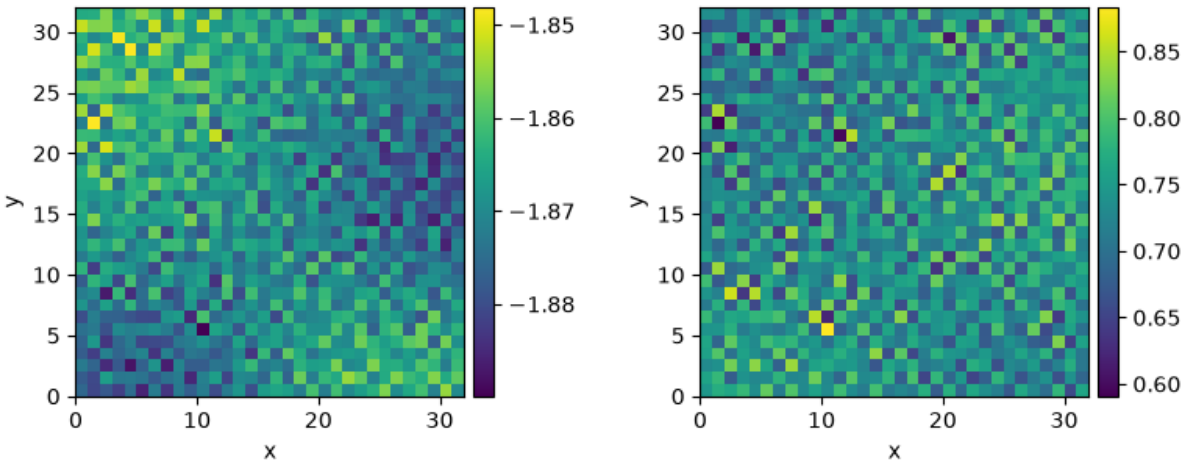
Total running time of the script: (0 minutes 1.057 seconds)

2.4.7 Custom Class for coupled PDEs

This example shows how to solve a set of coupled PDEs, the spatially coupled FitzHugh–Nagumo model, which is a simple model for the excitable dynamics of coupled Neurons:

$$\partial_t u = \nabla^2 u + u(u-\alpha)(1-u) + w$$
$$\partial_t w = \epsilon u$$

Here, α denotes the external stimulus and ϵ defines the recovery time scale. We implement this as a custom PDE class below.




```
  0%|          | 0/100.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/100.0 [00:00<?, ?it/s]
  0%|          | 0/100.0 [00:00<?, ?it/s]
  0%|          | 0.29/100.0 [00:00<00:43,  2.30it/s]
  1%|          | 1.09/100.0 [00:00<00:15,  6.57it/s]
  5%|▌         | 5.16/100.0 [00:00<00:06, 14.27it/s]
 14%|█▍        | 14.35/100.0 [00:00<00:04, 17.91it/s]
 28%|██▊       | 28.2/100.0 [00:01<00:03, 19.06it/s]
 45%|████▌     | 45.02/100.0 [00:02<00:02, 19.51it/s]
 64%|██████▍   | 63.5/100.0 [00:03<00:01, 19.82it/s]
 83%|████████▎ | 83.02/100.0 [00:04<00:00, 19.90it/s]
```

(continues on next page)

(continued from previous page)

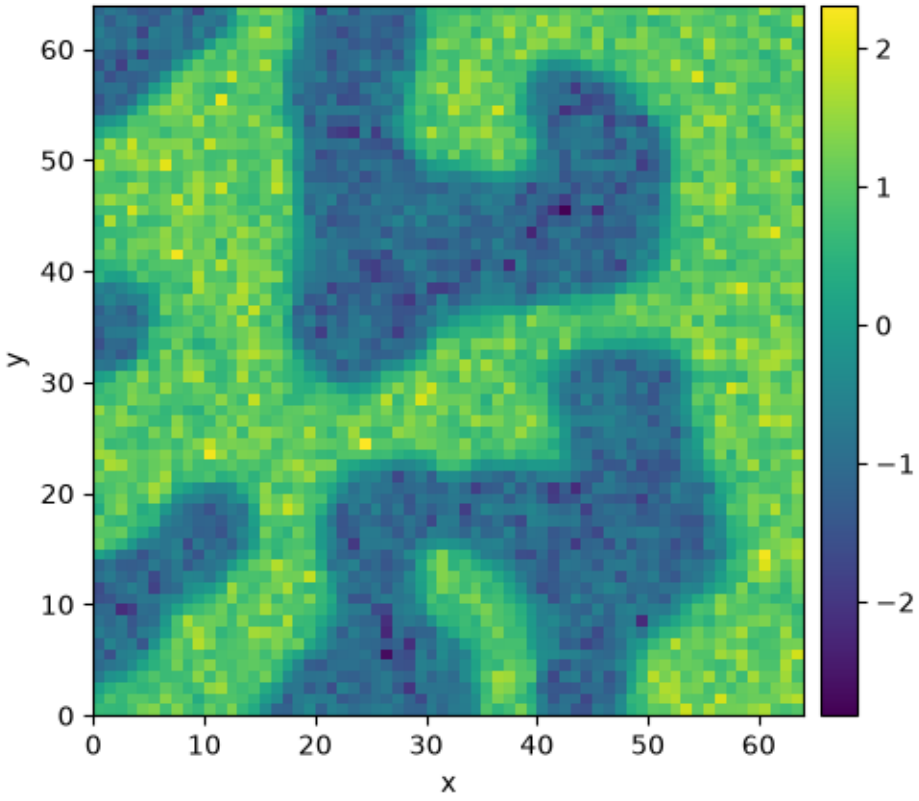
```
 83%|████████▎  | 83.02/100.0 [00:04<00:01, 16.61it/s]
100%|██████████| 100.0/100.0 [00:04<00:00, 20.01it/s]
100%|██████████| 100.0/100.0 [00:04<00:00, 20.01it/s]
```

```python
from pde import FieldCollection, PDEBase, UnitGrid


class FitzhughNagumoPDE(PDEBase):
    """FitzHugh-Nagumo model with diffusive coupling."""

    def __init__(self, stimulus=0.5, τ=10, a=0, b=0, bc="auto_periodic_neumann"):
        super().__init__()
        self.bc = bc
        self.stimulus = stimulus
        self.τ = τ
        self.a = a
        self.b = b

    def evolution_rate(self, state, t=0):
        v, w = state  # membrane potential and recovery variable

        v_t = v.laplace(bc=self.bc) + v - v**3 / 3 - w + self.stimulus
        w_t = (v + self.a - self.b * w) / self.τ

        return FieldCollection([v_t, w_t])


grid = UnitGrid([32, 32])
state = FieldCollection.scalar_random_uniform(2, grid)

eq = FitzhughNagumoPDE()
result = eq.solve(state, t_range=100, dt=0.01)
result.plot()
```

Total running time of the script: (0 minutes 5.192 seconds)

2.4.8 SDE with Stratonovich interpretation

This example solves a stochastic diffusion equation with Stratonovich interpretation




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:05<?, ?it/s]
  0%|          | 0.004/10.0 [00:05<3:28:22, 1250.72s/it]
  2%|▏         | 0.151/10.0 [00:05<05:27, 33.30s/it]
 11%|▉         | 1.078/10.0 [00:05<00:42,  4.81s/it]
 34%|███       | 3.429/10.0 [00:05<00:10,  1.63s/it]
 72%|██████▍   | 7.173/10.0 [00:06<00:02,  1.15it/s]
 72%|██████▍   | 7.173/10.0 [00:06<00:02,  1.07it/s]
100%|█████████| 10.0/10.0 [00:06<00:00,  1.50it/s]
100%|█████████| 10.0/10.0 [00:06<00:00,  1.50it/s]
```

```python
from pde import PDE, ScalarField, UnitGrid


class AllenCahnNoisePDE(PDE):
    """Allen-Cahn PDE with custom noise implementation."""

    use_noise_variance = True
```

(continues on next page)

(continued from previous page)

```
    def make_noise_variance(self, state, *, backend, ret_diff=False):
        """Make function that calculates noise variance."""
        noise = float(self.noise)

        if ret_diff:

            def noise_variance(state_data, t):
                return noise * state_data**2, 2 * noise * state_data

        else:

            def noise_variance(state_data, t):
                return noise * state_data**2

        return noise_variance


eq = AllenCahnNoisePDE(
    rhs={"c": "laplace(c) + c - c**3"}, noise=1.0, noise_interpretation="stratonovich"
)
state = ScalarField.random_uniform(UnitGrid([64, 64]), -1, 1)
result = eq.solve(state, t_range=10, dt=1e-3, solver="milstein")
result.plot()
```

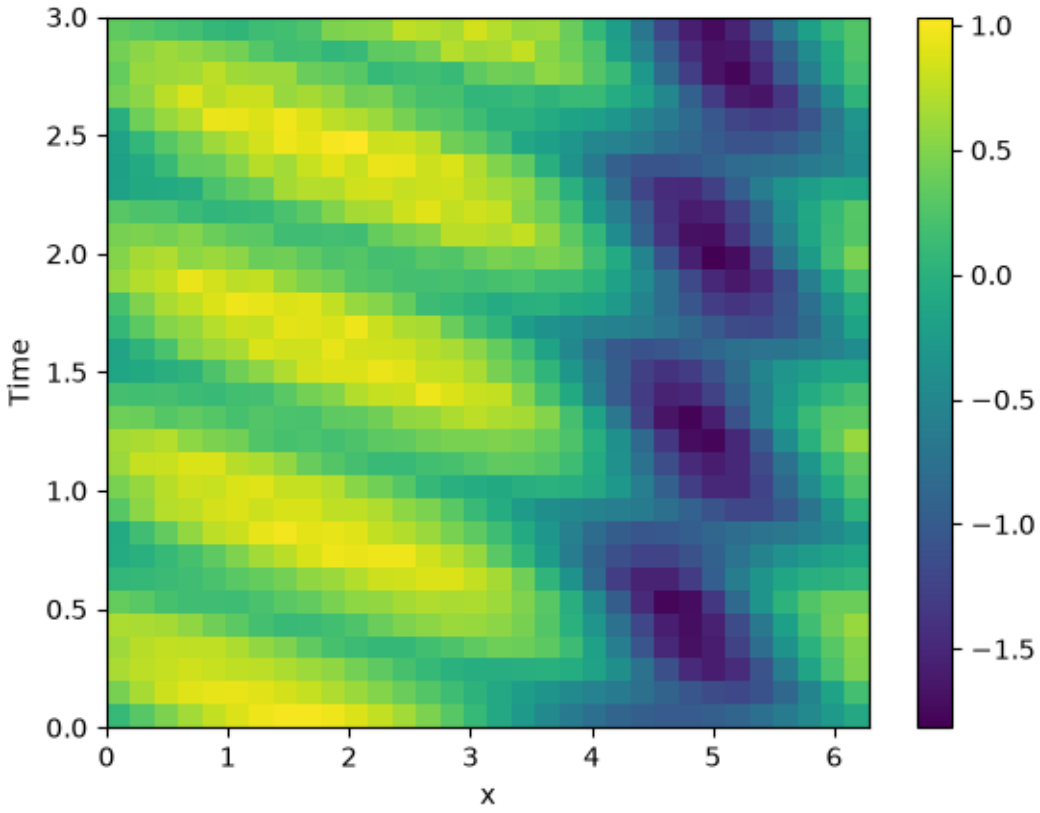
Total running time of the script: (0 minutes 6.783 seconds)

2.4.9 1D problem - Using custom class

This example implements a PDE that is only defined in one dimension. Here, we chose the Korteweg-de Vries equation, given by

$$\partial_t \phi = 6\phi \partial_x \phi - \partial_x^3 \phi$$

which we implement using a custom PDE class below.




```
from math import pi

from pde import CartesianGrid, MemoryStorage, PDEBase, ScalarField, plot_kymograph


class KortewegDeVriesPDE(PDEBase):
    """Korteweg-de Vries equation."""

    def evolution_rate(self, state, t=0):
        """Implement the python version of the evolution equation."""
        assert state.grid.dim == 1  # ensure the state is one-dimensional
        grad_x = state.gradient("auto_periodic_neumann")[0]
        return 6 * state * grad_x - grad_x.laplace("auto_periodic_neumann")


# initialize the equation and the space
grid = CartesianGrid([[0, 2 * pi]], [32], periodic=True)
state = ScalarField.from_expression(grid, "sin(x)")

# solve the equation and store the trajectory
storage = MemoryStorage()
eq = KortewegDeVriesPDE()
eq.solve(state, t_range=3, solver="scipy", tracker=storage.tracker(0.1))
```

(continues on next page)

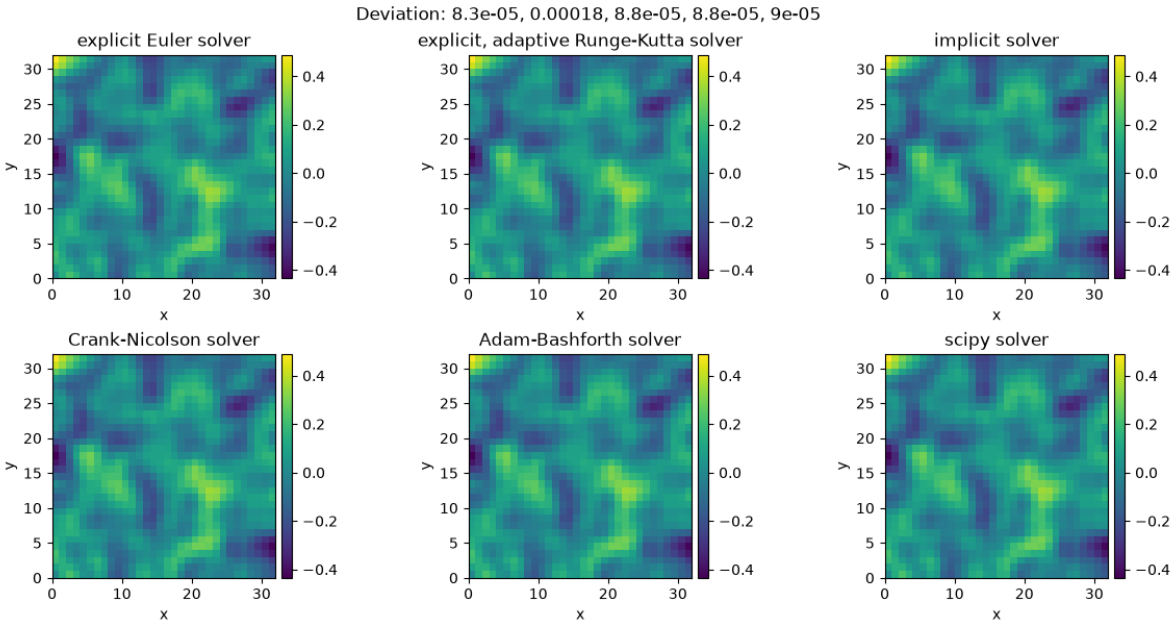
(continued from previous page)

```
# plot the trajectory as a space-time plot
plot_kymograph(storage)
```

Total running time of the script: (0 minutes 1.882 seconds)

2.4.10 Solver comparison

This example shows how to set up solvers explicitly and how to extract diagnostic information.




```
Diagnostic information for explicit Euler solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪0.0032848430000029794, 'tracker': 4.882999999722415e-05, 'compilation': 6.343449393}
↪, 'solver_start': '2026-07-10 12:19:42.601435+00:00', 'successful': True, 'stop_
↪reason': 'Reached final time', 'solver_duration': '0:00:00.003325', 't_final': 1.0},
↪ 'package_version': 'unknown', 'solver': {'class': 'EulerSolver', 'pde_class':
↪'DiffusionPDE', 'dt': 0.001, 'steps': 1000, 'dt_adaptive': False, 'stochastic':␣
↪False, 'backend': {'name': 'numba', 'implementation': 'numba'}, 'post_step_data':␣
↪None}}

Diagnostic information for explicit, adaptive Runge-Kutta solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪0.0005906709999976556, 'tracker': 4.924899999991794e-05, 'compilation': 7.
↪4390137370000105}, 'solver_start': '2026-07-10 12:19:50.050060+00:00', 'successful
↪': True, 'stop_reason': 'Reached final time', 'solver_duration': '0:00:00.000632',
↪'t_final': 1.0}, 'package_version': 'unknown', 'solver': {'class': 'RungeKuttaSolver
↪', 'pde_class': 'DiffusionPDE', 'dt': 0.19644766182541057, 'steps': 12, 'dt_adaptive
↪': True, 'stochastic': False, 'backend': {'name': 'numba', 'implementation': 'numba
↪'}, 'dt_statistics': {'min': 0.001, 'max': 0.16648687186135858, 'mean': 0.
↪08333333333333334, 'std': 0.05173057989539766, 'count': 12.0}, 'post_step_data':␣
↪None}}
```

(continues on next page)

(continued from previous page)

```
Diagnostic information for implicit solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪0.014146550999996066, 'tracker': 5.4359000003501023e-05, 'compilation': 4.
↪328175918000014}, 'solver_start': '2026-07-10 12:19:54.387547+00:00', 'successful':␣
↪True, 'stop_reason': 'Reached final time', 'solver_duration': '0:00:00.014194', 't_
↪final': 1.0}, 'package_version': 'unknown', 'solver': {'class': 'ImplicitSolver',
↪'pde_class': 'DiffusionPDE', 'dt': 0.001, 'steps': 1000, 'dt_adaptive': False,
↪'stochastic': False, 'backend': {'name': 'numba', 'implementation': 'numba'},
↪'function_evaluations': 0, 'post_step_data': None}}

Diagnostic information for Crank-Nicolson solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪0.0200661249999996, 'tracker': 6.106100001090908e-05, 'compilation': 6.487102604},
↪'solver_start': '2026-07-10 12:20:00.897003+00:00', 'successful': True, 'stop_reason
↪': 'Reached final time', 'solver_duration': '0:00:00.020124', 't_final': 1.0},
↪'package_version': 'unknown', 'solver': {'class': 'CrankNicolsonSolver', 'pde_class
↪': 'DiffusionPDE', 'dt': 0.001, 'steps': 1000, 'dt_adaptive': False, 'stochastic':␣
↪False, 'backend': {'name': 'numba', 'implementation': 'numba'}, 'function_
↪evaluations': 0, 'post_step_data': None}}

Diagnostic information for Adam-Bashforth solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪4.032563752999991, 'tracker': 6.150100000468228e-05, 'compilation': 1.
↪9956892520000054}, 'solver_start': '2026-07-10 12:20:02.922220+00:00', 'successful
↪': True, 'stop_reason': 'Reached final time', 'solver_duration': '0:00:04.033041',
↪'t_final': 1.0}, 'package_version': 'unknown', 'solver': {'class':
↪'AdamsBashforthSolver', 'pde_class': 'DiffusionPDE', 'dt': 0.001, 'steps': 1000,
↪'dt_adaptive': False, 'stochastic': False, 'backend': {'name': 'numba',
↪'implementation': 'numba'}, 'post_step_data': None}}

Diagnostic information for scipy solver:
{'controller': {'mpi_run': False, 't_start': 0, 't_end': 1.0, 'profiler': {'solver':␣
↪0.647358257999997, 'tracker': 4.7220000013226127e-05, 'compilation': 0.
↪0017100309999875662}, 'solver_start': '2026-07-10 12:20:06.960001+00:00',
↪'successful': True, 'stop_reason': 'Reached final time', 'solver_duration':
↪'0:00:00.647465', 't_final': np.float64(1.0)}, 'package_version': 'unknown', 'solver
↪': {'class': 'ScipySolver', 'pde_class': 'DiffusionPDE', 'dt': 0.001, 'steps': 61,
↪'stochastic': False, 'backend': {'name': 'numba', 'implementation': 'numba'}}}
```

```python
import pde

# initialize the grid, an initial condition, and the PDE
grid = pde.UnitGrid([32, 32])
field = pde.ScalarField.random_uniform(grid, -1, 1)
eq = pde.DiffusionPDE()
```

(continues on next page)

(continued from previous page)

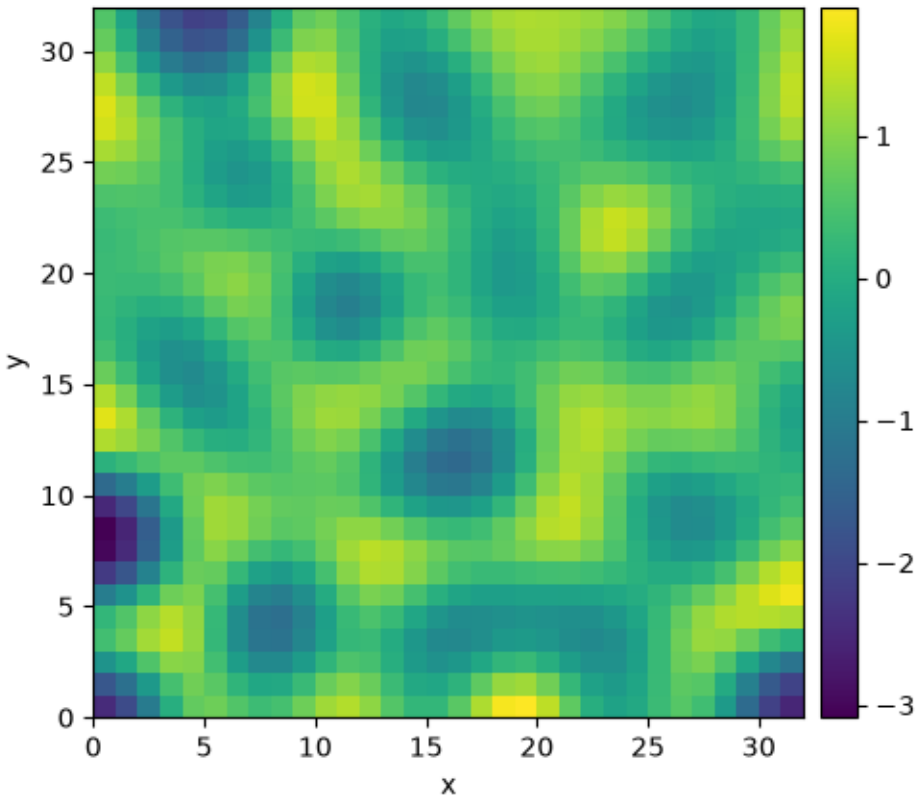
```
def run_solver(solver, label):
    """Helper function testing the solver."""
    controller = pde.Controller(solver, t_range=1, tracker=None)
    sol = controller.run(field, dt=1e-3)
    sol.label = label + " solver"
    print(f"Diagnostic information for {sol.label}:")
    print(controller.diagnostics)
    print()
    return sol


# try different solvers
solutions = [
    run_solver(pde.EulerSolver(eq), "explicit Euler"),
    run_solver(
        pde.RungeKuttaSolver(eq, adaptive=True), "explicit, adaptive Runge-Kutta"
    ),
    run_solver(pde.ImplicitSolver(eq), "implicit"),
    run_solver(pde.CrankNicolsonSolver(eq), "Crank-Nicolson"),
    run_solver(pde.AdamsBashforthSolver(eq), "Adam-Bashforth"),
    run_solver(pde.ScipySolver(eq), "scipy"),
]

# plot both fields and give the deviation as the title
deviations = [(solutions[0] - sol).fluctuations for sol in solutions]
title = "Deviation: " + ", ".join(f"{deviation:.2g}" for deviation in deviations[1:])
pde.FieldCollection(solutions).plot(title=title, arrangement=(2, 3))
```

Total running time of the script: (0 minutes 32.011 seconds)

2.4.11 Kuramoto-Sivashinsky - Compiled methods

This example implements a scalar PDE using a custom class with a numba-compiled method for accelerated calculations. We here consider the Kuramoto–Sivashinsky equation, which for instance describes the dynamics of flame fronts:

$$\partial_t u = -\frac{1}{2}|\nabla u|^2 - \nabla^2 u - \nabla^4 u$$




```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:13<?, ?it/s]
  0%|          | 0.01/10.0 [00:13<3:37:40, 1307.36s/it]
  2%|▏         | 0.22/10.0 [00:13<09:41, 59.43s/it]
  2%|▏         | 0.22/10.0 [00:13<09:41, 59.47s/it]
100%|██████████| 10.0/10.0 [00:13<00:00,  1.31s/it]
100%|██████████| 10.0/10.0 [00:13<00:00,  1.31s/it]
```

```python
from pde import PDEBase, ScalarField, UnitGrid


class KuramotoSivashinskyPDE(PDEBase):
    """Implementation of the normalized Kuramoto-Sivashinsky equation."""

    def __init__(self, bc="auto_periodic_neumann"):
        super().__init__()
        self.bc = bc
```

(continues on next page)

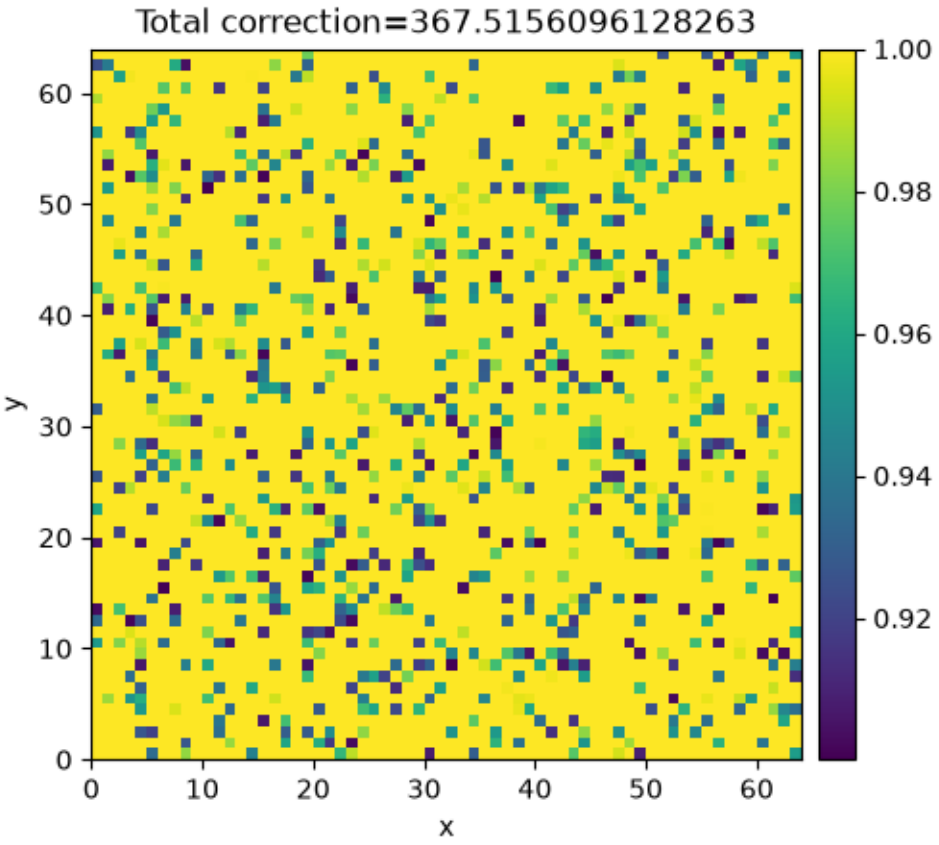
(continued from previous page)

```
    def evolution_rate(self, state, t=0):
        """Implement the python version of the evolution equation."""
        state_lap = state.laplace(bc=self.bc)
        state_lap2 = state_lap.laplace(bc=self.bc)
        state_grad_sq = state.gradient_squared(bc=self.bc)
        return -state_grad_sq / 2 - state_lap - state_lap2

    def make_evolution_rate(self, state, backend):
        """Compilable implementation of the PDE."""
        gradient_squared = state.grid.make_operator(
            "gradient_squared", bc=self.bc, backend=backend, dtype=state.dtype
        )
        laplace = state.grid.make_operator(
            "laplace", bc=self.bc, backend=backend, dtype=state.dtype
        )

        def pde_rhs(data, t):
            return -0.5 * gradient_squared(data) - laplace(data + laplace(data))

        return pde_rhs


grid = UnitGrid([32, 32])  # generate grid
state = ScalarField.random_uniform(grid)  # generate initial condition

eq = KuramotoSivashinskyPDE()  # define the pde
result = eq.solve(state, t_range=10, dt=0.01)
result.plot()
```

Total running time of the script: (0 minutes 13.165 seconds)

2.4.12 Post-step hook function in a custom class

The hook function created by `make_post_step_hook()` is called after each time step. The function can modify the state, keep track of additional information, and abort the simulation.




```
  0%|          | 0/10000.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10000.0 [00:00<?, ?it/s]
  0%|          | 0/10000.0 [00:00<?, ?it/s]
  0%|          | 0.3/10000.0 [00:00<00:28, 356.66it/s]
  0%|          | 0.3/10000.0 [00:00<00:49, 200.17it/s]
  0%|          | 0.3/10000.0 [00:00<00:53, 186.03it/s]
```

```python
from pde import PDEBase, ScalarField, UnitGrid


class CustomPDE(PDEBase):
    def make_post_step_hook(self, state, backend):
        """Create a hook function that is called after every time step."""

        def post_step_hook(state_data, t, post_step_data):
            """Limit state 1 and abort when standard deviation exceeds 1."""
            i = state_data > 1  # get violating entries
            overshoot = (state_data[i] - 1).sum()  # get total correction
            state_data[i] = 1  # limit data entries
```

(continues on next page)

(continued from previous page)

```
            post_step_data += overshoot  # accumulate total correction
            if post_step_data > 400:
                # Abort simulation when correction exceeds 400
                # Note that the `post_step_data` of the previous step will be␣
↪returned.
                raise StopIteration
            return state_data, post_step_data

        return post_step_hook, 0.0  # hook function and initial value for data

    def evolution_rate(self, state, t=0):
        """Evaluate the right hand side of the evolution equation."""
        return state.__class__(state.grid, data=1)  # constant growth


grid = UnitGrid([64, 64])  # generate grid
state = ScalarField.random_uniform(grid, 0.0, 0.5)  # generate initial condition

eq = CustomPDE()
result = eq.solve(state, dt=0.1, t_range=1e4)
result.plot(title=f"Total correction={eq.diagnostics['solver']['post_step_data']}")
```

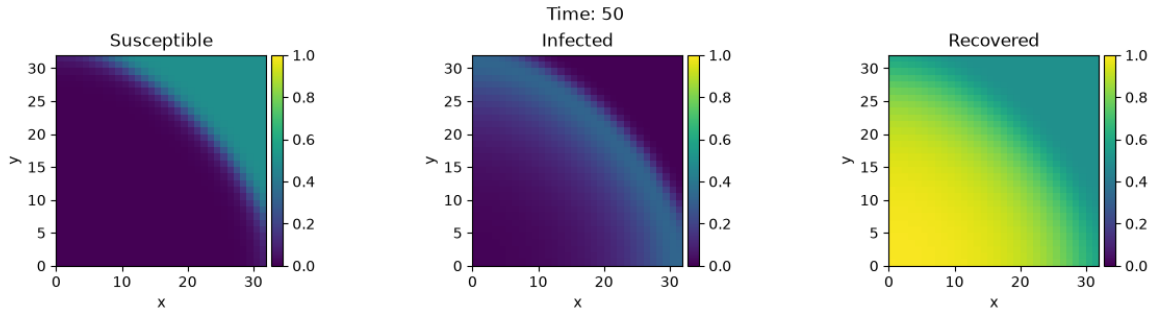
Total running time of the script: (0 minutes 0.085 seconds)

2.4.13 Custom PDE class: SIR model

This example implements a spatially coupled SIR model with the following dynamics for the density of susceptible, infected, and recovered individuals:

$$\partial_t s = D\nabla^2 s - \beta i s$$
$$\partial_t i = D\nabla^2 i + \beta i s - \gamma i$$
$$\partial_t r = D\nabla^2 r + \gamma i$$

Here, D is the diffusivity, β the infection rate, and γ the recovery rate.




```
  0%|          | 0/50.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/50.0 [00:00<?, ?it/s]
  0%|          | 0/50.0 [00:00<?, ?it/s]
  0%|          | 0.02/50.0 [00:00<23:43, 28.48s/it]
  0%|          | 0.05/50.0 [00:00<09:32, 11.46s/it]
  1%|          | 0.66/50.0 [00:00<00:47,  1.05it/s]
  6%|▌         | 3.17/50.0 [00:00<00:12,  3.65it/s]
```

(continues on next page)

(continued from previous page)

```
 17%|▉         | 8.35/50.0 [00:01<00:06,  6.16it/s]
 32%|▌         | 15.75/50.0 [00:02<00:04,  7.01it/s]
 47%|▊         | 23.6/50.0 [00:03<00:03,  7.43it/s]
 63%|▉         | 31.74/50.0 [00:04<00:02,  7.69it/s]
 80%|▊         | 40.08/50.0 [00:05<00:01,  7.85it/s]
 97%|▉         | 48.52/50.0 [00:05<00:00,  8.22it/s]
 97%|▉         | 48.52/50.0 [00:06<00:00,  7.79it/s]
100%|▉         | 50.0/50.0 [00:06<00:00,  8.03it/s]
100%|▉         | 50.0/50.0 [00:06<00:00,  8.03it/s]
```

```python
from pde import FieldCollection, PDEBase, PlotTracker, ScalarField, UnitGrid


class SIRPDE(PDEBase):
    """SIR-model with diffusive mobility."""

    def __init__(
        self, beta=0.3, gamma=0.9, diffusivity=0.1, bc="auto_periodic_neumann"
    ):
        super().__init__()
        self.beta = beta  # transmission rate
        self.gamma = gamma  # recovery rate
        self.diffusivity = diffusivity  # spatial mobility
        self.bc = bc  # boundary condition

    def get_state(self, s, i):
        """Generate a suitable initial state."""
        norm = (s + i).data.max()  # maximal density
        if norm > 1:
            s /= norm
            i /= norm
        s.label = "Susceptible"
        i.label = "Infected"

        # create recovered field
        r = ScalarField(s.grid, data=1 - s - i, label="Recovered")
        return FieldCollection([s, i, r])

    def evolution_rate(self, state, t=0):
        s, i, r = state
        diff = self.diffusivity
        ds_dt = diff * s.laplace(self.bc) - self.beta * i * s
        di_dt = diff * i.laplace(self.bc) + self.beta * i * s - self.gamma * i
        dr_dt = diff * r.laplace(self.bc) + self.gamma * i
        return FieldCollection([ds_dt, di_dt, dr_dt])
```

(continues on next page)

(continued from previous page)

```
eq = SIRPDE(beta=2, gamma=0.1)

# initialize state
grid = UnitGrid([32, 32])
s = ScalarField(grid, 1)
i = ScalarField(grid, 0)
i.data[0, 0] = 1
state = eq.get_state(s, i)

# simulate the pde
tracker = PlotTracker(interrupts=10, plot_args={"vmin": 0, "vmax": 1})
sol = eq.solve(state, t_range=50, dt=1e-2, tracker=["progress", tracker])
```

Total running time of the script: (0 minutes 6.482 seconds)

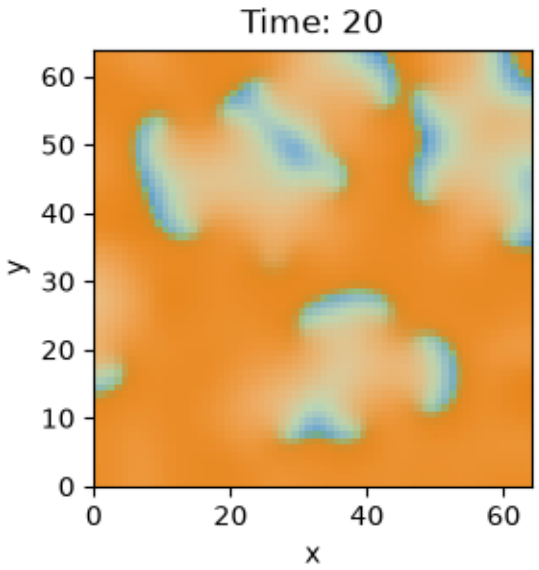
2.4.14 Brusselator - Using custom class

This example implements the Brusselator with spatial coupling,

$$\partial_t u = D_0 \nabla^2 u + a - (1 + b)u + vu^2$$
$$\partial_t v = D_1 \nabla^2 v + bu - vu^2$$

Here, D_0 and D_1 are the respective diffusivity and the parameters a and b are related to reaction rates.

Note that the PDE can also be implemented using the `PDE` class; see *the example*. However, that implementation is less flexible and might be more difficult to extend later.




```
import numpy as np

from pde import FieldCollection, PDEBase, PlotTracker, ScalarField, UnitGrid


class BrusselatorPDE(PDEBase):
    """Brusselator with diffusive mobility."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, a=1, b=3, diffusivity=None, bc="auto_periodic_neumann"):
        super().__init__()
        self.a = a
        self.b = b
        self.diffusivity = [1, 0.1] if diffusivity is None else diffusivity
        self.bc = bc  # boundary condition

    def get_initial_state(self, grid):
        """Prepare a useful initial state."""
        u = ScalarField(grid, self.a, label="Field $u$")
        v = self.b / self.a + 0.1 * ScalarField.random_normal(grid, label="Field $v$")
        return FieldCollection([u, v])

    def evolution_rate(self, state, t=0):
        """Pure python implementation of the PDE."""
        u, v = state
        rhs = state.copy()
        d0, d1 = self.diffusivity
        rhs[0] = d0 * u.laplace(self.bc) + self.a - (self.b + 1) * u + u**2 * v
        rhs[1] = d1 * v.laplace(self.bc) + self.b * u - u**2 * v
        return rhs

    def make_evolution_rate(self, state, backend):
        """Compilable implementation of the PDE."""
        d0, d1 = self.diffusivity
        a, b = self.a, self.b
        laplace = state.grid.make_operator(
            "laplace", bc=self.bc, backend=backend, dtype=state.dtype
        )

        def pde_rhs(state_data, t):
            u = state_data[0]
            v = state_data[1]

            rate_u = d0 * laplace(u) + a - (1 + b) * u + v * u**2
            rate_v = d1 * laplace(v) + b * u - v * u**2
            return np.stack((rate_u, rate_v))

        return pde_rhs


# initialize state
grid = UnitGrid([64, 64])
eq = BrusselatorPDE(diffusivity=[1, 0.1])
state = eq.get_initial_state(grid)

# simulate the pde
tracker = PlotTracker(interrupts=1, plot_args={"kind": "merged", "vmin": 0, "vmax": 5}
↪)
sol = eq.solve(state, t_range=20, dt=1e-3, tracker=tracker)
```

Total running time of the script: (0 minutes 10.177 seconds)

CHAPTER

THREE

USER MANUAL

3.1 Mathematical basics

To solve partial differential equations (PDEs), the *py-pde* package provides differential operators to express spatial derivatives. These operators are implemented using the finite difference method to support various boundary conditions. The time evolution of the PDE is then calculated using the method of lines by explicitly discretizing space using the grid classes. This reduces the PDEs to a set of ordinary differential equations, which can be solved using standard methods as described below.

3.1.1 Curvilinear coordinates

The package supports multiple curvilinear coordinate systems through `pde.grids.coordinates`. They permit exploiting symmetries present in physical systems. Consequently, many grids implemented in *py-pde* inherently assume symmetry of the described fields. However, a drawback of curvilinear coordinates is the fact that the basis vectors now depend on position, which makes tensor fields less intuitive and complicates the expression of differential operators. To avoid confusion, we specify the used coordinate systems explicitly:

Polar coordinates

Polar coordinates describe points by a radius r and an angle ϕ in a two-dimensional Cartesian coordinate system. They are defined by the transformation

$$\begin{cases} x = r\cos(\phi) \\ y = r\sin(\phi) \end{cases} \quad \text{for } r \in [0, \infty] \text{ and } \phi \in [0, 2\pi)$$

The associated symmetric grid `PolarSymGrid` assumes that fields depend only on the radial coordinate r. Note that vector and tensor fields can still have components in the angular direction. In particular, vector fields still have two components: $\vec{v}(r) = v_r(r)\vec{e}_r + v_\phi(r)\vec{e}_\phi$.

Spherical coordinates

Spherical coordinates describe points by a radius r, an azimuthal angle θ, and a polar angle ϕ. The conversion to ordinary Cartesian coordinates reads

$$\begin{cases} x = r\sin(\theta)\cos(\phi) \\ y = r\sin(\theta)\sin(\phi) \\ z = r\cos(\theta) \end{cases} \quad \text{for } r \in [0, \infty],\ \theta \in [0, \pi],\ \text{and } \phi \in [0, 2\pi)$$

The associated symmetric grid `SphericalSymGrid` assumes that fields depend only on the radial coordinate r. Note that vector and tensor fields can still have components in the two angular directions.

Warning

Not all results of differential operators on vectorial and tensorial fields can be expressed in terms of fields that only depend on the radial coordinate r. In particular, the gradient of a vector field can only be calculated if the azimuthal component of the vector field vanishes. Similarly, the divergence of a tensor field can only be taken in special situations.

Cylindrical coordinates

Cylindrical coordinates describe points by a radius r, an axial coordinate z, and a polar angle ϕ. The conversion to ordinary Cartesian coordinates reads

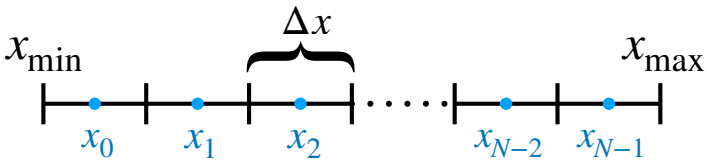
$$\begin{cases} x = r\cos(\phi) \\ y = r\sin(\phi) \\ z = z \end{cases} \quad \text{for } r \in [0, \infty], z \in \mathbb{R}, \text{ and } \phi \in [0, 2\pi)$$

The associated symmetric grid `CylindricalSymGrid` assumes that fields depend only on the coordinates r and z. Vector and tensor fields still specify all components in the three-dimensional space.

Warning

The order of components in the vector and tensor fields defined on cylindrical grids is different than in ordinary math. While it is common to use (r, ϕ, z), we here use the order (r, z, ϕ). It might thus be best to access components by name instead of index.

3.1.2 Spatial discretization



The finite differences scheme used by *py-pde* is currently restricted to orthogonal coordinate systems with uniform discretization. Because of the orthogonality, each axis of the grid can be discretized independently. For simplicity, we only consider uniform grids, where the support points are spaced equidistantly along a given axis, i.e., the discretization Δx is constant. If a given axis covers values in a range $[x_{\text{min}}, x_{\text{max}}]$, a discretization with N support points can then be thought of as covering the axis with N equal-sized boxes; see inset. Field values are then specified for each box, i.e., the support points lie at the centers of the box:

$$x_i = x_{\text{min}} + \left(i + \frac{1}{2}\right)\Delta x \quad \text{for} \quad i = 0, \ldots, N-1$$
$$\Delta x = \frac{x_{\text{max}} - x_{\text{min}}}{N}$$

which is also indicated in the inset.

Differential operators are implemented using the usual second-order central differences. This requires introducing virtual support points at x_{-1} and x_N, which can be determined from the boundary conditions at $x = x_{\text{min}}$ and $x = x_{\text{max}}$, respectively. The field classes automate this transparently. However, if you need more control over boundary conditions, you can access the full underlying data using the `field._data_full`, which will have $N + 2$ entries along an axis that has N support points. In this case, the first and last entries (`data_full[0]` and `data_full[N + 1]`) denote the lower and upper virtual point, respectively. The actual field data can be obtained using `data_full[1:-1]` or the `field.data` attribute for convenience. Note that functions evaluating differential operators generally expect the full data as input while they return only valid data.

3.1.3 Temporal evolution

Once the fields have been discretized, the PDE reduces to a set of coupled ordinary differential equations (ODEs), which can be solved using standard methods. This reduction is also known as the method of lines. The *py-pde* package implements the simple Euler scheme in the *EulerSolver* class, and a more advanced Runge-Kutta scheme scheme in the *RungeKuttaSolver* class. Conceptually, solver objects store the strategy and configuration, while the actual state updates are performed by stepping functions created from these solver objects. For the simple implementations of these explicit methods, the user typically specifies a fixed time step, although adaptive methods, which adjust the time step automatically, are also often used and available in the package. One problem with explicit solvers is that they require small time steps to stably evolve some PDEs; such PDEs are then often called 'stiff'. Stiff PDEs can sometimes be solved more efficiently by using implicit methods. This package provides a simple implementation of the Backward Euler method in the *ImplicitSolver* class. Finally, more advanced methods are available by wrapping the `scipy.integrate.solve_ivp()` in the *ScipySolver* class.

3.2 Basic usage

We describe the typical workflow to solve a PDE using *py-pde*. Throughout this section, we assume that the package has been imported using `import pde`.

3.2.1 Defining the geometry

The state of the system is described in a discretized geometry, also known as a *grid*. The package focuses on simple geometries, which work well for the employed finite difference scheme. Grids are defined by instances of various classes that capture the symmetries of the underlying space. In particular, the package offers Cartesian grids of *1* to *3* dimensions via *CartesianGrid*, as well as curvilinear coordinates for spherically symmetric systems in two dimension (*PolarSymGrid*) and three dimensions (*SphericalSymGrid*), as well as the special class *CylindricalSymGrid* for a cylindrical geometry which is symmetric in the angle.

All grids allow to set the size of the underlying geometry and the number of support points along each axis, which determines the spatial resolution. Moreover, most grids support periodic boundary conditions. For example, a rectangular grid with one periodic boundary condition can be specified as

```
grid = pde.CartesianGrid([[0, 10], [0, 5]], [20, 10], periodic=[True, False])
```

This grid will have a rectangular shape of 10x5 with square unit cells of side length *0.5*. Note that the grid will only be periodic in the *x*-direction.

3.2.2 Initializing a field

Fields specify the values at the discrete points of the grid defined in the previous section. Most PDEs in the package involve scalar variables, which are represented using the class *ScalarField*. However, tensors with rank 1 (vectors) and rank 2 are also supported using *VectorField* and *Tensor2Field*, respectively. In any case, a field is initialized using a pre-defined grid, e.g., `field = pde.ScalarField(grid)`. Optional arguments allow setting field values as well as a label that is later used in plotting, e.g., `field1 = pde.ScalarField(grid, data=1, label="Ones")`. Moreover, fields can be initialized randomly (`field2 = pde.ScalarField.random_normal(grid, mean=0.5)`) or from a mathematical expression, which may depend on the coordinates of the grid (`field3 = pde.ScalarField.from_expression(grid, "x * y")`).

All field classes support basic arithmetic operations and can be used much like numpy arrays. Moreover, they have methods for applying differential operators, e.g., the result of applying the Laplacian to a scalar field is returned by calling the method *laplace()*, which returns another instance of *ScalarField*, whereas *gradient()* returns a *VectorField*. Combining these functions with ordinary arithmetic on fields allows represent the right hand side of many partial differential equations that appear in physics. Importantly, the differential operators work with flexible boundary conditions.

3.2.3 Specifying the PDE

PDEs are also instances of special classes and a number of classical PDEs are already pre-defined in the module *pde.pdes*. Moreover, the special class *PDE* allows defining PDEs by simply specifying the expression on their right hand side. To see how this works in practice, let us consider the Kuramoto–Sivashinsky equation, $\partial_t u = -\nabla^4 u - \nabla^2 u - \frac{1}{2}|\nabla u|^2$, which describes the time evolution of a scalar field u. A simple implementation of this equation reads

```
eq = pde.PDE({"u": "-gradient_squared(u) / 2 - laplace(u + laplace(u))"})
```

Here, the argument defines the evolution rate for all fields (in this case only u). The expression on the right hand side can contain typical mathematical functions and the operators defined by the package.

3.2.4 Running the simulation

To solve the PDE, we use the generated initial condition, i.e., the initial field `field`, which are evolved forward in time by the PDE. This field also defines the geometry on which the PDE is solved. In the simplest case, the solution is then obtain by running

```
result = eq.solve(field, t_range=10, dt=1e-2)
```

Here, *t_range* specifies the duration over which the PDE is considered and *dt* specifies the time step. The *result* field will be defined on the same grid as the initial condition *field*, but instead contain the data value at the final time. Note that all intermediate states are discarded in the simulation above and no information about the dynamical evolution is retained. To study the dynamics, one can either analyze the evolution on the fly or store its state for subsequent analysis. Both these tasks are achieved using *trackers*, which analyze the simulation periodically. For instance, to store the state for some time points in memory, one uses

```
storage = pde.MemoryStorage()
result = eq.solve(field, t_range=10, dt=1e-3, tracker=["progress", storage.
↪tracker(1)])
```

Note that we also included the special identifier `"progress"` in the list of trackers, which shows a progress bar during the simulation. Another useful tracker is `"plot"` which displays the state on the fly.

3.2.5 Analyzing the results

Sometimes is suffices to plot the final result, which can be done using `result.plot()`. The final result can of course also be analyzed quantitatively, e.g., using `result.average` to obtain its mean value. If the intermediate states have been saved as indicated above, they can be analyzed subsequently:

```
for time, field in storage.items():
    print(f"t={time}, field={field.magnitude}")
```

Moreover, a movie of the simulation can be created using *movie()*, i.e., by calling `pde.movie(storage, filename=FILE)`, where *FILE* determines where the movie is written.

3.3 Advanced usage

3.3.1 Boundary conditions

A crucial aspect of partial differential equations are boundary conditions, which need to be specified at the domain boundaries. For the simple domains contained in *py-pde*, all boundaries are orthogonal to one of the axes in the domain, so boundary conditions need to be applied to both sides of each axis. Here, the lower side of an axis can have a different condition than the upper side. For instance, one can enforce the value of a field to be *4* at the lower side and its derivative (in the outward direction) to be *2* on the upper side using the following code:

```
bc = {"x-": {"value": 4}, "x+": {"derivative": 2}}

grid = pde.UnitGrid([16])
field = pde.ScalarField(grid)
field.laplace(bc)
```

Here, the Laplace operator applied to the field in the last line will respect the boundary conditions. Note that both sides of the axis can be specified together if their conditions are the same. For instance, to enforce a value of *3* on both side, one could simply use `bc = {"x": {"value": 3}}`. Vectorial boundary conditions, e.g., to calculate the vector gradient or tensor divergence, can have vectorial values for the boundary condition. Generally, only the normal components at a boundary need to be specified if an operator reduces the rank of a field, e.g., for divergences. Otherwise, e.g., for gradients and Laplacians, the full field needs to be specified at the boundary.

Boundary values that depend on space can be set by specifying a mathematical expression, which may depend on the coordinates of all axes:

```
# two different conditions for lower and upper end of x-axis
bc_left = {"derivative": 0.1}
bc_right = {"value": "sin(y / 2)"}
# the same condition on the lower and upper end of the y-axis
bc_y = {"value": "sqrt(1 + cos(x))"}

grid = UnitGrid([32, 32])
field = pde.ScalarField(grid)
field.laplace(bc={"x-": bc_left, "x+": bc_right, "y": bc_y})
```

Warning

To interpret arbitrary expressions, the package uses `exec()`. It should therefore not be used in a context where malicious input could occur.

Heterogeneous values can also be specified by directly supplying an array, whose shape needs to be compatible with the boundary, i.e., it needs to have the same shape as the grid but with the dimension of the axis along which the boundary is specified removed.

There also exist special boundary conditions that impose a more complex value of the field (`bc="value_expression"`) or its derivative (`bc="derivative_expression"`). Beyond the spatial coordinates that are already supported for the constant conditions above, the expressions of these boundary conditions can depend on the time variable `t`. Moreover, these boundary conditions also accept python functions with signature *(adjacent_value, dx, *coords, t)*, thus greatly enlarging the flexibility with which boundary conditions can be expressed. Note that PDEs need to supply the current time *t* when setting the boundary conditions, e.g., when applying the differential operators. The pre-defined PDEs and the general class *PDE* already support time-dependent boundary conditions.

To specify the same boundary conditions for many sides, the wildcard specifier can be used: For example, `bc = {"*": {"value": 1}, "y+": {"derivative": 0}}` specifies Dirichlet conditions for all axes, except the upper y-axis, where a Neumann condition is imposed instead. If all axes have the same condition, the outer dictionary can be skipped, so that `bc = {"*": {"value": 1}}` imposes the same conditions as `bc = {"value": 1}`. Moreover, many boundaries have convenient names, so that for instance *x-* can be replaced by *left*, and *y+* can be replaced by *top*.

One important aspect about boundary conditions is that they need to respect the periodicity of the underlying grid. For instance, in a 2d grid with one periodic axis, the following boundary condition can be used:

```
grid = pde.UnitGrid([16, 16], periodic=[True, False])
field = pde.ScalarField(grid)
field.laplace({"x": "periodic", "y": {"derivative": 0}})
```

For convenience, this typical situation can be described with the special boundary condition *auto_periodic_neumann*, e.g., calling the Laplace operator using `field.laplace("auto_periodic_neumann")` is identical to the example above. Similarly, the special condition *auto_periodic_dirichlet* enforces periodic boundary conditions or Dirichlet boundary condition (vanishing value), depending on the periodicity of the underlying grid.

In summary, we have the following options for boundary conditions on a field c

Table 1: Supported boundary conditions

Name	Condition	Example
Dirichlet	$c = 0$	`"dirichlet"` or `"value"`
	$c = \text{const}$	`{"value": 1.5}`
	$c = f(x, t)$	`{"value_expression": "sin(x)"}`
	$c = f(x, t)$	`{"value_expression": func}` with function `func(value, dx, *coords, t)`
Neumann	$\partial_n c = 0$	`"neumann"` or `"derivative"`
	$\partial_n c = \text{const}$	`{"derivative": -2}`
	$\partial_n c = f(x, t)$	`{"derivative_expression": "exp(t)"}`
Robin	$\partial_n c + \text{value} \cdot c = \text{const}$	`{"type": "mixed", "value": 2, "const": 7}`
	$\partial_n c + \text{value} \cdot c = \text{const}$	`{"type": "mixed_expression", "value": "exp(t)", "const": "3 * x"}`
Curvature	$\partial_n^2 c = \text{const}$	`{"curvature": 3}`
Periodic	$c(0) = c(L)$	`"periodic"`
Anti-periodic	$c(0) = -c(L)$	`"anti-periodic"`
Periodic or Dirichlet	$c(0) = c(L)$ or $c = 0$	`"auto_periodic_dirichlet"`
Periodic or Neumann	$c(0) = c(L)$ or $\partial_n c = 0$	`"auto_periodic_neumann"`

Here, ∂_n denotes a derivative in outward normal direction, f denotes an arbitrary function given by an expression (see next section), x denotes coordinates along the boundary, t denotes time.

Finally, we support the advanced technique of setting the virtual points at the boundary manually. This can be achieved by passing a python function that takes as its first argument a `ndarray`, which contains the full field data including the virtual points, and a second, optional argument, which is a dictionary containing additional parameters, like the current time point *t* in case of a simulation; see *`BoundariesSetter`* for more details.

3.3.2 Expressions

Expressions are strings that describe mathematical expressions. They can be used in several places, most prominently in defining PDEs using *`PDE`*, in creating fields using *`from_expression()`*, and in defining boundary conditions; see section above. Expressions are parsed using `sympy`, so the expected syntax is defined by this python package. While we describe some common use cases below, it might be best to test the abilities using the *`evaluate()`* function.

Warning

To interpret arbitrary expressions, the package uses `exec()`. It should therefore not be used in a context where malicious input could occur.

Simple expressions can contain many standard mathematical functions, e.g., `sin(a) + b**2` is a valid expression. *PDE* and *evaluate()* furthermore accept differential operators defined in this package. Note that operators need to be specified with their full name, i.e., *laplace* for a scalar Laplacian and *vector_laplace* for a Laplacian operating on a vector field. Moreover, the dot product between two vector fields can be denoted by using `dot(field1, field2)` in the expression, and `outer(field1, field2)` calculates an outer product. In this case, boundary conditions for the operators can be specified using the *bc* argument, in which case the same boundary conditions are applied to all operators. The additional argument *bc_ops* provides a more fine-grained control, where conditions for each individual operator can be specified.

Field expressions can also directly depend on spatial coordinates. For instance, if a field is defined on a two-dimensional Cartesian grid, the variables `x` and `y` denote the local coordinates. To initialize a step profile in the x-direction, one can use either `(x > 5)` or `heaviside(x - 5, 0.5)`, where the second argument denotes the returned value in case the first argument is *0*. For convenience, Cartesian coordinates are also available when using curvilinear grids. The respective coordinate values at a point can be accessed using `cartesian[i]`, where `i` is an index, e.g., *i=0* for the first axis (normally the x-axis). Finally, expressions for equations in *PDE* can explicitly depend on time, which is denoted by the variable `t`.

Expressions also support user-defined functions via the *user_funcs* argument, which is a dictionary that maps the name of a function to an actual implementation. Finally, constants can be defined using the *consts* argument. Constants can either be individual numbers or spatially extended data, which provide values for each grid point. Note that in the latter case only the actual grid data should be supplied, i.e., the *data* attribute of a potential field class.

3.3.3 Custom PDE classes

To implement a new PDE in a way that all of the machinery of *py-pde* can be used, one needs to subclass *PDEBase* and overwrite at least the *evolution_rate()* method. A simple implementation for the Kuramoto–Sivashinsky equation could read

```
class KuramotoSivashinskyPDE(PDEBase):

    def evolution_rate(self, state, t=0):
        """Evaluate the right hand side of the evolution equation."""
        state_laplacian = state.laplace(bc="auto_periodic_neumann")
        state_gradient = state.gradient(bc="auto_periodic_neumann")
        return (- state_laplacian.laplace(bc="auto_periodic_neumann")
                - state_laplacian
                - 0.5 * state_gradient.to_scalar("squared_sum"))
```

A slightly more advanced example would allow for attributes that for instance define the boundary conditions and the diffusivity:

```
class KuramotoSivashinskyPDE(PDEBase):

    def __init__(self, diffusivity=1, bc="auto_periodic_neumann", bc_laplace="auto_
↪periodic_neumann"):
        """Initialize the class with a diffusivity and boundary conditions."""
        self.diffusivity = diffusivity
        self.bc = bc
        self.bc_laplace = bc_laplace

    def evolution_rate(self, state, t=0):
        """Evaluate the right hand side of the evolution equation."""
        state_laplacian = state.laplace(bc=self.bc)
        state_gradient = state.gradient(bc=self.bc)
        return (- state_laplacian.laplace(bc=self.bc_laplace)
```

(continues on next page)

(continued from previous page)

```
            - state_laplacian
            - 0.5 * self.diffusivity * (state_gradient @ state_gradient))
```

We replaced the call to `to_scalar('squared_sum')` with a dot product with itself (using the @ notation), which is equivalent. Note that the numpy implementation of the right hand side of the PDE is rather slow since it runs mostly in pure python and constructs a lot of intermediate field classes. While such an implementation is helpful for testing initial ideas, actual computations should be performed with compiled PDEs as described below.

Another feature of custom PDE classes is a special function that is called after every time step. This function is defined by `make_post_step_hook()` and allows direct manipulation of the state data and also abortion of the simulation by raising `StopIteration` (this latter function does not work with all backends).

```
class AbortEarlyPDE(PDEBase):

    def make_post_step_hook(self, state, backend):
        """Create a hook function that is called after every time step."""

        def post_step_hook(state_data, t, post_step_data):
            """Limit state to [-1, 1] & abort when standard deviation exceeds 1."""
            np.clip(state_data, -1, 1, out=state_data)  # limit state
            if state_data.std() > 1:
                raise StopIteration  # abort simulation
            post_step_data += 1  # increment number of times hook was called
            return state_data, post_step_data

        return post_step_hook, 0  # hook function and initial value for data

    def evolution_rate(self, state, t=0):
        """Evaluate the right hand side of the evolution equation."""
        return state
```

We here use a simple constant evolution equation. The hook defined by the first method does two things: First, it limits the state to the interval *[-1, 1]* using `numpy.clip()`. Second, it evaluates the standard deviation across the entire data, aborting the simulation when the value exceeds one. Note that the hook always receives the data always as a `ndarray` and not as a full field class. The hook can also keep track of additional data via `post_step_data`, which is a `ndarray` that can be updated in place.

3.3.4 Low-level operators

This section explains how to use the low-level version of the field operators. This is necessary for the compiled implementations described above and it might be necessary to use parts of the *py-pde* package in other packages.

Differential operators

Applying a differential operator to an instance of *ScalarField* is a simple as calling `field.laplace(bc)`, where *bc* denotes the boundary conditions. Calling this method returns another *ScalarField*, which in this case contains the discretized Laplacian of the original field. The equivalent call using the low-level interface is

```
apply_laplace = field.grid.make_operator("laplace", bc)

laplace_data = apply_laplace(field.data)
```

Here, the first line creates a function `apply_laplace` for the given grid `field.grid` and the boundary conditions *bc*. This function can be applied to `numpy.ndarray` instances, e.g. `field.data`. Note that the result of this call is again

a numpy.ndarray.

Similarly, a gradient operator can be defined

```
grid = UnitGrid([6, 8])
apply_gradient = grid.make_operator("gradient", bc="auto_periodic_neumann")

data = np.random.random((6, 8))
gradient_data = apply_gradient(data)
assert gradient_data.shape == (2, 6, 8)
```

Note that this example does not even use the field classes. Instead, it directly defines a *grid* and the respective gradient operator. This operator is then applied to a random field and the resulting numpy.ndarray represents the 2-dimensional vector field.

The make_operator method of the grids generally supports the following differential operators: 'laplacian', 'gradient', 'gradient_squared', 'divergence', 'vector_gradient', 'vector_laplace', and 'tensor_divergence'. Moreover, generic operators that perform a derivative along a single axis are supported: Specifying 'd_dx' for instance performs a single derivative along the *x*-direction, 'd_dy_forward' uses a forward derivative along the *y*-direction, and 'd_d2r' performs a second derivative in *r*-direction. A complete list of operators supported by a certain grid class can be obtained from the class property GridClass.operators. New operators can be added using the class method GridClass.register_operator().

Field integration

The integral of an instance of *ScalarField* is usually determined by accessing the property field.integral. Since the integral of a discretized field is basically a sum weighted by the cell volumes, calculating the integral using only numpy is easy:

```
cell_volumes = field.grid.cell_volumes
integral = (field.data * cell_volumes).sum()
```

Note that cell_volumes is a simple number for Cartesian grids, but is an array for more complicated grids, where the cell volume is not uniform.

Field interpolation

The fields defined in the *py-pde* package also support linear interpolation by calling field.interpolate(point). Similarly to the differential operators discussed above, this call can also be translated to code that does not use the full package:

```
grid = UnitGrid([6, 8])
interpolate = grid.make_integrator(bc="auto_periodic_neumann")

data = np.random.random((6, 8))
value = interpolate(data, np.array([3.5, 7.9]))
```

We first create a function interpolate, which is then used to interpolate the field data at a certain point. Note that the coordinates of the point need to be supplied as a numpy.ndarray and that only the interpolation at single points is supported. However, iteration over multiple points can be fast when the loop is compiled by a backend.

Inner products

For vector and tensor fields, *py-pde* defines inner products that can be accessed conveniently using the @-syntax: `field1 @ field2` determines the scalar product between the two fields. The package also provides an implementation for an dot-operator:

```
grid = UnitGrid([6, 8])
field1 = VectorField.random_normal(grid)
field2 = VectorField.random_normal(grid)

dot_operator = field1.make_dot_operator()

result = dot_operator(field1.data, field2.data)
assert result.shape == (6, 8)
```

Here, `result` is the data of the scalar field resulting from the dot product.

3.3.5 Compiled implementations of PDEs

The compiled operators introduced in the previous section can be used to implement a compiled method for the evolution rate of PDEs. As an example, we now extend the class `KuramotoSivashinskyPDE` introduced above:

```
class KuramotoSivashinskyPDE(PDEBase):

    def __init__(self, diffusivity=1, bc="auto_periodic_neumann", bc_laplace="auto_
↪periodic_neumann"):
        """Initialize class with diffusivity and boundary conditions."""
        self.diffusivity = diffusivity
        self.bc = bc
        self.bc_laplace = bc_laplace


    def evolution_rate(self, state, t=0):
        """Evaluate right hand side of PDE."""
        state_lapacian = state.laplace(bc=self.bc)
        state_gradient = state.gradient(bc="auto_periodic_neumann")
        return (- state_lapacian.laplace(bc=self.bc_laplace)
                - state_lapacian
                - 0.5 * self.diffusivity * (state_gradient @ state_gradient))


    def make_evolution_rate(self, state, backend):
        """Make compilable implementation of the evolution rate."""
        # make attributes locally available
        diffusivity = self.diffusivity

        # create operators with correct attributes
        args = {"backend": backend, "dtype": state.dtype}
        laplace_u = state.grid.make_operator("laplace", bc=self.bc, **args)
        gradient_u = state.grid.make_operator("gradient", bc=self.bc, **args)
        laplace2_u = state.grid.make_operator("laplace", bc=self.bc_laplace, **args)
        dot = VectorField(state.grid).make_dot_operator(backend=backend)

        def pde_rhs(state_data, t=0):
```

(continues on next page)

(continued from previous page)

```
            """Evaluate right hand side of PDE."""
            state_lapacian = laplace_u(state_data)
            state_grad = gradient_u(state_data)
            return (- laplace2_u(state_lapacian)
                    - state_lapacian
                    - diffusivity / 2 * dot(state_grad, state_grad))

        return pde_rhs
```

To activate the compiled implementation of the evolution rate, we simply have to overwrite the *make_evolution_rate()* method. This method expects an example of the state class (e.g., an instance of *ScalarField*) and a backend. The method returns a function that calculates the evolution rate. The *state* argument is necessary to define the grid and the dimensionality of the data that the returned function is supposed to be handling. The implementation of the compiled function is split in several parts, where we first copy the attributes that are required by the implementation. This is necessary, since backends, such as those based on `numba` or `torch`, freeze the values when compiling the function, so that in the example above the diffusivity cannot be altered without recompiling. In the next step, we create all operators that we need subsequently. Here, we use the boundary conditions defined by the attributes, which requires two different laplace operators, since their boundary conditions might differ. In the last step, we define the actual implementation of the evolution rate as a local function, which is returned and can be compiled by the backend.

3.3.6 Stochastic partial differential equation

We also support stochastic differential equations, which can be a lot trickier to deal with then their deterministic counterparts. To support the most common use cases, we offer two different interfaces to define the noise that affects the evolution. In the simplest case, one inherits from *SDEBase*, which already implements additive Gaussian white noise. The following listing is thus sufficient to enable noise

```
class NoisyPDE(SDEBase):

    def evolution_rate(self, state, t=0):
        ...


eq = NoisyPDE(noise=1)
```

In this case, the *noise* argument controls the variance of the additive noise.

In many situations, more complex multiplicative noise is needed. Multiplicative noise implies that the noise variance depends on the value of the field itself. To allow such field-dependent noise, one can overwrite the method *make_noise_variance()*, which implements the additive noise described above. In the simplest case, we thus have

```
class NoisyPDE(PDEBase):

    use_noise_variance = True

    def evolution_rate(self, state, t=0):
        ...

    def make_noise_variance(self, state, backend, ret_diff=False):
        def noise_variance(state_data, t):
            return state_data**2
        return noise_variance
```

(continues on next page)

(continued from previous page)

```
eq = NoisyPDE()
```

Here, the factory method returns a function that can be evaluated to determine the noise variance depending on the field (and time).

Stochastic differential equations with multiplicative noise are much more difficult to analyze and simulate. For instance, it is not sufficient to just give the evolution equation, but it needs to be accompanied with an *interpretation*. By default, we use the standard Itô interpretation, but it is possible to simulate other interpretations by supplying the argument `noise_interpretation` to subclasses of *`SDEBase`*. If another interpretation, such as `stratonovich` or `anti-ito`, is specified, the standard solvers automatically add the corresponding drift terms to convert the equation to the standard Itô form. To make this possible, the *`make_noise_variance()`* now also needs to return the derivative of the noise variance with respect to the field values:

```
class NoisyPDE(SDEBase):

    use_noise_variance = True

    def evolution_rate(self, state, t=0):
        ...

    def make_noise_variance(self, state, backend, ret_diff=False):
        if ret_diff:
            def noise_variance(state_data, t):
                return state_data**2, 2 * state_data
        else:
            def noise_variance(state_data, t):
                return state_data**2
        return noise_variance

eq = NoisyPDE(noise_interpretation="stratonovich")
```

Note that supplying the derivative is also required to use *`MilsteinSolver`*, which has better convergence properties.

Finally, we offer a completely different way of implementing noises, which is a bit more low-level. Enabling the `use_noise_realization` of the PDE, the solvers look for `make_noise_realization()`, which should return a function that can be called to determine a realization of the noise, which will be directly used during time stepping. In this setup, simple additive noise can be implemented as

```
class NoisyPDE(SDEBase):

    use_noise_variance = False
    use_noise_realization = True

    def make_noise_realization(self, state, backend):
        data_shape = state.data.shape
        scale = np.sqrt(1 / state.grid.cell_volumes)

        def noise_realization(state_data, t):
            return scale * np.random.randn(*data_shape)

        return noise_realization
```

This complex approach is more versatile. For example, it allows implementing non-Gaussian or conserved noise, if this

is needed. However, this approach does not benefit from the automatic evaluation of derivatives, which is required to use the *MilsteinSolver* and to change the interpretation of the equation.

3.4 Configuration parameters

Configuration parameters affect how the package behaves. Parameters can generally be set using a dictionary-like interface in either the global configuration `config` or in backend-specific settings at *config*. To provide flexibility, the default backends (whose name is identical to the backend package, e.g. *numba*, *jax*, or *torch*) have a special behavior such that their configuration is linked with the global configuration. Consequently, the following commands all yield identical results:

```
from pde import config, get_backend

get_backend("numba").config["debug"] = True
config["backend"]["numba"]["debug"] = True
config["backend.numba.debug"] = True
```

In contrast, custom backends (example: `get_backend("torch:cuda")`) inherit the global configuration, but modifying their configuration would not influence the global configuration.

Here is a list of all global configuration options, which includes the backend-specific parameters

backend.jax.compile
: Enables compilation in the `jax` backend. **(Default value: True)**

backend.jax.device
: Determines the device that is used for the jax backend. Common options include `cpu` and `gpu`. The special value `auto` chooses `gpu` if available, otherwise `cpu`. **(Default value: 'cpu')**

backend.jax.dtype_downcasting
: Determines whether dtype downcasting is used automatically. A typical example are jax devices that only support float32, so the numpy arrays using float64 need to be converted. If enabled, this happens automatically. **(Default value: True)**

backend.numba.debug
: Determines whether numba uses the debug mode for compilation. If enabled, this emits extra information that might be useful for debugging. **(Default value: False)**

backend.numba.fastmath
: Determines whether the fastmath flag is set during compilation. If enabled, some mathematical operations might be faster, but less precise. This flag does not affect infinity detection and NaN handling. **(Default value: True)**

backend.numba.multithreading
: Determines whether multiple threads are used in numba-compiled code. Enabling this option accelerates a small subset of operators applied to fields defined on large grids. Possible options are 'never' (disable multithreading), 'only_local' (disable on HPC hardware), and 'always' (enable if number of grid points exceeds `multithreading_threshold`) **(Default value: 'only_local')**

backend.numba.multithreading_threshold
: Minimal number of support points of grids before multithreading is enabled in numba compilations. Has no effect when multihreading is disabled via the `multithreading` option. **(Default value: 65536)**

backend.numba.use_spectral
: Indicates whether a spectral implementation should be used where possible. Note that this option is only implemented for few operators. **(Default value: False)**

backend.torch.compile
: Enables compilation in the `torch` backend. **(Default value: True)**

backend.torch.device
Determines the device that is used for the torch backend. Common options include `cpu`, `cuda`, and more specific choices, like `cuda:0`. The special value `auto` chooses `cuda` if available, otherwise `cpu`. **(Default value: 'cpu')**

backend.torch.dtype_downcasting
Determines whether dtype downcasting is used automatically. A typical example are torch devices that only support float32, so the numpy arrays using float64 need to be converted. If enabled, this happens automatically. **(Default value: True)**

boundaries.accept_lists
Indicate whether boundary conditions can be set using the deprecated legacy format, where conditions for individual axes and sides where set using lists. If disabled, only the new format using dicts is supported. **(Default value: True)**

default_backend
Indicate which backend is selected by default. **(Default value: 'numba')**

operators.cartesian.laplacian_2d_corner_weight
Weighting factor for the corner points of the 2d cartesian Laplacian stencil. The standard value is zero, corresponding to the traditional 5-point stencil. Alternative choices are 1/2 (Oono-Puri stencil) and 1/3 (Patra-Karttunen or Mehrstellen stencil); see https://en.wikipedia.org/wiki/Nine-point_stencil. Note that some backends might ignore this option. **(Default value: 0.0)**

operators.conservative_stencil
Indicates whether conservative stencils should be used for differential operators on curvilinear grids. Conservative operators ensure mass conservation at slightly slower computation speed. Note that some backends might ignore this option. **(Default value: True)**

operators.tensor_symmetry_check
Indicates whether tensor fields are checked for having a suitable form for evaluating differential operators in curvilinear coordinates where some axes are assumed to be symmetric. In such cases, some tensor components might need to vanish, so the result of the operator can be expressed. Note that some backends might ignore this option. **(Default value: True)**

Tip

To disable parallel computing via multithreading in the package, the following code could be added to the start of the script:

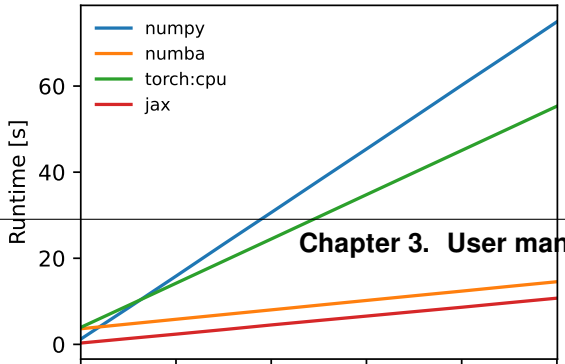
```
from pde import config
config["backend.numba.multithreading"] = "never"

# actual code using py-pde
```

The default value *only_local* already disables multithreading when the package detects it is run in an HPC environment.

3.5 Performance

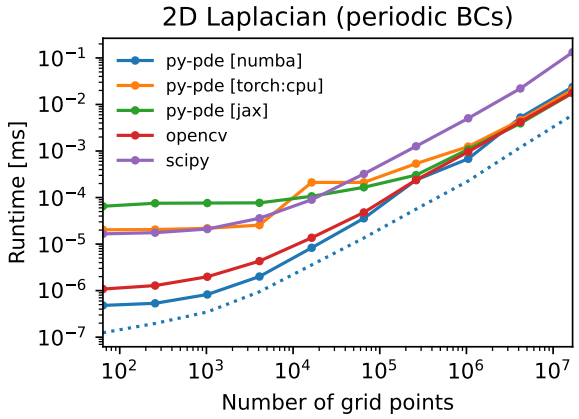
3.5.1 Measuring performance

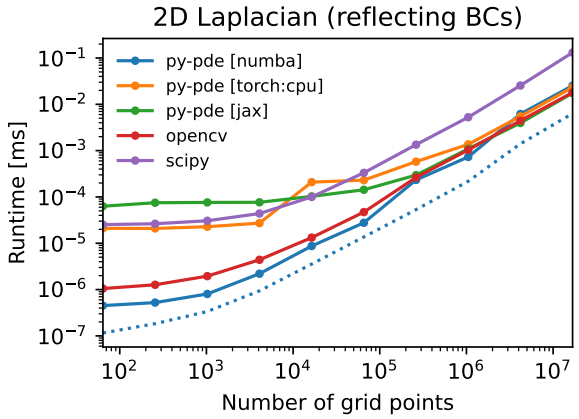
The performance of the *py-pde* package depends on many details and general statements are thus difficult to make. However, since the core operators can be compiled using `numba` or `torch`, many operations of the package proceed at performances close to most compiled languages. For a simple test case, consider the plot on the right, which shows how much real time passed until a certain simulation



time was reached. Here, the slope of the curve quantifies the raw speed of the solver, whereas the intercept with the y-axis indicates the start-up time, which is mainly dominated by the compilation of the code. The trade-off between compilation time and run time varies between backends and will also depend on details of the simulation.

A more microscopic performance measurement concerns the actual operators, which can also be compiled using various backends. Generally, a simple Laplace operator applied to fields defined on a Cartesian grid has performance that is similar to the operators supplied by the OpenCV package. The following figures illustrate this by showing the duration of evaluating the Laplacian on grids of increasing number of support points for two different boundary conditions (lower duration is better):





Note that the call overhead is generally lower in the *py-pde* package, so that the performance on small grids is particularly good using the *numba* backend (blue line). Moreover, further speed-up is possible (blue dotted line) if boundary conditions do not need to be set, i.e., if ghost cells were set by some other function before. However, realistic use cases probably need more complicated operations and it is thus always necessary to profile the respective code. This can be done using the function `estimate_computation_speed()` or the traditional `timeit`, `profile`, or even more sophisticated profilers like pyinstrument.

3.5.2 Improving performance

Besides the underlying implementation of the operators, a major factor for performance is the specific numerical problem being solved and the methods that are used to address it. It is thus paramount to choose the right backend. Here, compiled backends, like *numba* and *torch*, provide the fastest speed, particularly for larger systems, but they require an initial compilation, which can be costly. For small systems, it can thus be advisable to use the *numpy* backend.

As a rule of thumb, simulations run faster when there are fewer degrees of freedom. In the case of partial differential equations, this often means using a coarser grid with fewer support points. However, there often also is an lower bound to the number of support points if structures of a certain length scales need to be resolved. Reducing the number of support points not only reduces the number of variables to be treated, but it can also allow for larger time steps. This is particularly transparent for the simple diffusion equation, where a von Neumann stability analysis reveals that the maximal time step scales as one over the discretization length squared! Choosing the right time step obviously also affects performance of a simulation. The package supports automatic choice of suitable time steps, using adaptive stepping schemes. To enable those, simply specify this in the solver call:

```
eq.solve(t_range=10, adaptive=True)
```

An initial time step can be supplied, but is not necessary. Additional factors influencing the performance of the package include the compiler used for `numpy`, `scipy`, and of course `numba`. Performance of compiled backends, such as *numba*, *jax*, or *torch*, can vary tremendously depending on problem size, the chosen device, and many other details. Moreover, the BLAS and LAPACK libraries might make a difference. The package has some basic support for multithreading, which can be accelerated using the *Threading Building Blocks* library. Finally, it can help to install the intel short vector math library (SVML). However, this is not distributed with **macports** and might thus be more difficult to enable.

Using **macports**, one could for instance install the following variants of typical packages

```
port install py312-numpy +gcc14+openblas
port install py312-scipy +gcc14+openblas
port install py312-numba +tbb
```

Note that you can disable the automatic multithreading via *Configuration parameters*.

Note that many details of the installed packages can affect performance and one should thus test various options if speed is critical. The suggestions above might also already be outdated.

3.5.3 Multiprocessing using MPI

The package also supports parallel simulations of PDEs using the Message Passing Interface (MPI), which allows combining the power of CPU cores that do not share memory. To use this advanced simulation technique, a working implementation of MPI needs to be installed on the computer. Usually, this is done automatically, when the optional package `numba-mpi` is installed via *pip* or *conda*.

To run simulations in parallel, the special solver *`ExplicitMPISolver`* needs to be used and the entire script needs to be started using `mpiexec`. Taken together, a minimal example reads

```
from pde import DiffusionPDE, ScalarField, UnitGrid

grid = UnitGrid([64, 64])
state = ScalarField.random_uniform(grid, 0.2, 0.3)

eq = DiffusionPDE(diffusivity=0.1)
result = eq.solve(state, t_range=10, dt=0.1, solver="explicit_mpi")

if result is not None:  # restrict the output to the main node
    result.plot()
```

Saving this script as *multiprocessing.py*, we can evoke a parallel simulation using

```
mpiexec -n 2 python3 multiprocessing.py
```

Here, the number *2* determines the number of cores that will be used. Note that macOS might require an additional hint on how to connect the processes even when they are run on the same machine (e.g., your workstation). It might help to run `mpiexec -n 2 -host localhost:2 python3 multiprocessing.py` in this case.

In the example above, two python processes will start in parallel and run independently at first. In particular, both processes will load all packages and create the initial *state* field as well as the PDE class *eq*. Once the *explicit_mpi* solver is evoked, the processes will start communicating. *py-pde* will split up the full grid into two sub-grids, in this case of shape 32x64, distribute the associated sub-fields to both processes and ask each process to evolve the PDE for their sub-field. Note that boundary conditions are treated and boundary values are exchanged between neighboring sub-grids automatically. To avoid confusion, trackers will only be used on one process and also the result is only returned in one process to avoid problems where multiple process write data simultaneously. Consequently, the example above checked whether *result is None* (in which case the corresponding process is a child process) and only resumes analysis when the result is actually present.

The automatic treatment tries to use sensible default values, so typical simulations work out of the box. However, in some situations it might be advantageous to adjust these values. For instance, the decomposition of the grid can be affected by an argument *decomposition*, which can be passed to the *solve()* method or the *ExplicitMPISolver*. The argument should be a list with one integer for each axis in the grid, which specifies how often the particular axis is divided.

Warning

The automatic division of the grid into sub-grids can lead to unexpected behavior, particularly in custom PDEs that were not designed for this use case. As a rule of thumb, all local operations are fine (since they can be performed on each subgrid), while global operations might need synchronization between all subgrids. One example is integration, which has been implemented properly in *py-pde*. Consequently, it is safe to use `integral`.

3.6 Contributing code

3.6.1 Structure of the package

The functionality of the *pde* package is split into multiple subpackages. The domain, together with its symmetries, periodicities, and discretizations, is described by classes defined in *grids*. Discretized fields are represented by classes in *fields*, which have methods for differential operators with various boundary conditions collected in *boundaries*. The actual pdes are collected in *pdes* and the respective solvers are defined in *solvers*. The actual numerical computations are done in backends, which are implemented in *backends*.

3.6.2 Extending functionality

All code is build on a modular basis, making it easy to introduce new classes that integrate with the rest of the package. For instance, it is simple to define a new partial differential equation by subclassing *PDEBase*. Alternatively, PDEs can be defined by specifying their evolution rates using mathematical expressions by creating instances of the class *PDE*. Moreover, new grids can be introduced by subclassing *GridBase*.

The actual calculations are done by backends, which offer an interface for doing the calculation details. These backends are defined in *backends* and allow accessing their details independently. For instance, the numba-accelerated operators can be used without any other part of the package by calling the method *make_operator()* on the `numba_backend()` object. Moreover, new operators can be associated with grids by registering them using `numba_backend.register_operator()`. For instance, to create a new operator for the cylindrical grid one needs to define a factory function that creates the operator. This factory function takes an instance of *BoundariesList* as an argument and returns a function that takes as an argument the actual data array for the grid. Note that the grid itself

is an attribute of *BoundariesList*. This operator would be registered with the grid by calling `numba_backend.register_operator(CylindricalSymGrid, "operator", make_operator)`, where the arguments denote the grid class, the name of the operator, and the factory function, respectively. Similar options exist for the *jax* and the *torch* backend, but there can sometimes be subtle difference to comply with specific requirements of the backends.

3.6.3 Data layout

Since the package supports several different backends, data can reside in various places (e.g., CPU or GPU). To avoid confusion, we adhere to the following principles: Data that the user manipulates directly should always be stored in numpy arrays on the CPU. In contrast, inner loops for solving PDEs can move memory to GPU or any other device. Consequently, operators typically assume that data is stored in the native version of the respective backend.

The data layout of field classes (subclasses of *FieldBase*) was chosen to allow for a simple decomposition of different fields and tensor components. Consequently, data is laid out in memory such that spatial indices are last. For instance, the data of a vector field `field` defined on a 2d Cartesian grid will have three dimensions and can be accessed as `field.data[vector_component, x, y]`, where `vector_component` is either 0 or 1. Note that *FieldCollection* linearizes all vector- and tensor-components, to force all data in a unified array format. How this works is shown in the following example:

```
grid = pde.UnitGrid([7, 9])
scalar = pde.ScalarField.random_
↪uniform(grid)
assert scalar.data.shape == (7, 9)
tensor = pde.Tensor2Field.random_
↪uniform(grid)
assert tensor.data.shape == (2, 2, 7, 9)
collection = pde.FieldCollection([scalar,
↪ tensor])
assert collection.data.shape == (5, 7, 9)

assert np.all(collection.data[0] ==␣
↪scalar.data)
assert np.all(collection.data[1] ==␣
↪tensor.data[0, 0])
assert np.all(collection.data[2] ==␣
↪tensor.data[0, 1])
assert np.all(collection.data[3] ==␣
↪tensor.data[1, 0])
assert np.all(collection.data[4] ==␣
↪tensor.data[1, 1])
```

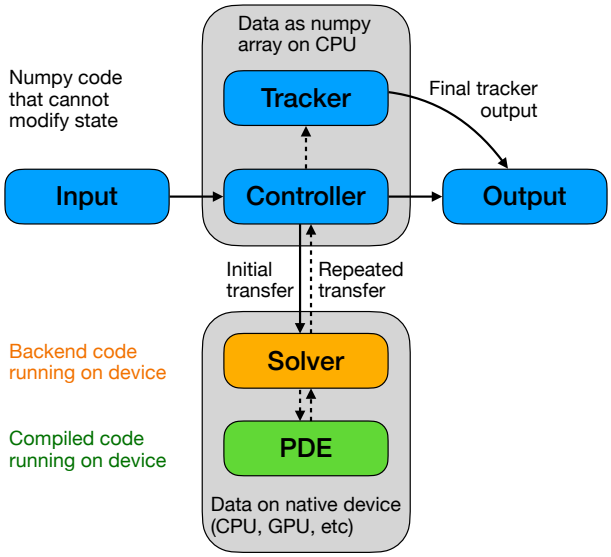


Fig. 2: Schematic of the data flow during a simulation.

3.6.4 Coding style

The coding style is enforced using ruff, based on the styles suggest by isort and black. Moreover, we use Google Style docstrings, which might be best learned by example. The documentation, including the docstrings, are written using reStructuredText, with examples in the following cheatsheet. To ensure the integrity of the code, we also try to provide many test functions, contained in the separate sub-folder `tests`. These tests can be ran using scripts in the `scripts` subfolder in the root folder. This folder also contain a script `tests_types.sh`, which uses `mypy` to check the consistency of the python type annotations. We use these type annotations for additional documentation and they have also already been useful for finding some bugs. Finally, we have pre-commit hooks, which you should install using *pre-commit install*.

We also have some conventions that should make the package more consistent and thus easier to use. For instance, we try to use `properties` instead of getter and setter methods as often as possible. Because we use `numba` or `torch` to speed up computations, we need to pass around (compiled) functions regularly. The names of the methods and functions that make such functions, i.e. that return callables, should start with 'make_*' where the wildcard should describe the purpose of the function being created.

3.6.5 Running unit tests

The *pde* package contains several unit tests, collected in the `tests` folder in the project root. These tests ensure that basic functions work as expected, in particular when code is changed in future versions. To run all tests, there are a few convenience scripts in the root directory `scripts`. The most basic script is `tests_run.sh`, which uses `pytest` to run the tests. Clearly, the python package `pytest` needs to be installed. There are also additional scripts that for instance run tests in parallel (needs the python package `pytest-xdist` installed), measure test coverage (needs package `pytest-cov` installed), and make simple performance measurements. Moreover, there is a script `test_types.sh`, which uses `mypy` to check the consistency of the python type annotations and there is a script `format_code.sh`, which formats the code automatically to adhere to our style.

Before committing a change to the code repository, it is good practice to run the tests, check the type annotations, and the coding style with the scripts described above.

3.7 Citing the package

To cite or reference *py-pde* in other work, please refer to the publication in the Journal of Open Source Software. Here are the respective bibliographic records in Bibtex format:

```
@article{py-pde,
    Author = {David Zwicker},
    Doi = {10.21105/joss.02158},
    Journal = {Journal of Open Source Software},
    Number = {48},
    Pages = {2158},
    Publisher = {The Open Journal},
    Title = {py-pde: A Python package for solving partial differential equations},
    Url = {https://doi.org/10.21105/joss.02158},
    Volume = {5},
    Year = {2020}
}
```

and in RIS format:

```
TY  - JOUR
AU  - Zwicker, David
JO  - Journal of Open Source Software
IS  - 48
```

(continues on next page)

(continued from previous page)

```
SP  - 2158
PB  - The Open Journal
T1  - py-pde: A Python package for solving partial differential equations
UR  - https://doi.org/10.21105/joss.02158
VL  - 5
PY  - 2020
```

3.8 Code of Conduct

3.8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

3.8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

3.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

3.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at david.zwicker@ds.mpg.de. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

3.8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

FOUR

REFERENCE MANUAL

The py-pde package provides tools for solving partial differential equations.

This package provides classes and methods for solving partial differential equations (PDEs) on various grids using different numerical methods. Key components include:

- **Fields**: Data structures representing scalar, vector, and tensor fields on grids
- **Grids**: Spatial discretizations including Cartesian and curvilinear coordinates
- **PDEs**: Pre-defined PDEs and a framework for defining custom PDEs
- **Solvers**: Time-stepping algorithms for evolving PDEs
- **Trackers**: Tools for monitoring and analyzing simulations
- **Storage**: Methods for storing simulation data
- **Visualization**: Functions for visualizing fields and creating movies

See the submodules for detailed documentation.

Subpackages:

4.1 pde.backends package

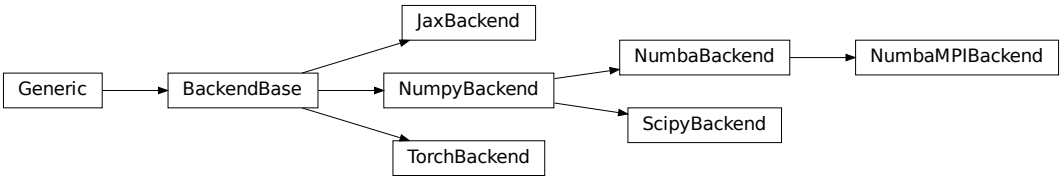
Defines backends, which implement efficient numerical simulations.

Backends are classes that provide logic to carry out numerical calculations. In particular, each of these classes implements various operators. In principle, backend classes can be defined independently, but we use some inheritance to share logic.

Moreover, we provide a *BackendRegistry*, which allows selecting backends by their identifier, so users do not usually need to construct backend classes. Users should access the registry via the function *get_backend()* to load a backend in their code.

BackendRegistry	Class handling all backends and their configurations.
JaxBackend	Defines `jax` backend.
NumbaBackend	Defines `numba` backend.
NumbaMPIBackend	Defines MPI-compatible numba backend.
NumpyBackend	Defines `numpy` backend, from which all other backends inherit.
ScipyBackend	Defines `scipy` backend.
TorchBackend	Defines `torch` backend.

Inheritance structure of the classes:



class BackendBase(*config*, ** (Keyword-only parameters separator (PEP 3102))*, *name=None*)

Bases: `Generic[TNativeArray]`

Basic backend from which all other backends inherit.

The generic type parameter *TNativeArray* determines the type of the native data representation of the backend.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

compile_function(*func*, ***kwargs*)

General method that compiles a user function.

Parameters

- **func** (*callable*) – The function that needs to be compiled for this backend
- ****kwargs** – Additional arguments affecting how the function is compiled

Return type
TFunc

config: *Config*

Configuration options of this backend.

Type
dict

config_inheritance: list[str] = []

Additional backends that are queried for configuration parameters.

Type
list

copy_data: bool = False

Data must be copied from CPU numpy representation to a native device.

Type
bool

classmethod from_args(*config*, *args=''*, ***, *name=None*)

Initialize backend with extra arguments.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **args** (*str*) – Additional arguments that determine how the backend is initialized
- **name** (*str*) – The name of the backend

Return type
Self

get_operator_info(*grid*, *operator*)

Return an operator for a particular grid.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.

Returns
information for the operator

Return type
OperatorInfo

get_registered_operators(*grid_id*)

Returns all operators defined for a grid.

Parameters
grid_id (GridBase or its type) – Grid or grid class for which the operators need to be returned

Return type
set[str]

implementation: str = 'undefined'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

property info: dict[str, Any]

relevant information about the backend

Type
dict

make_data_setter(*grid*, *rank*, *bcs=None*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (GridBase) – Grid for which the data setter is defined
- **rank** (*int*) – Rank of the data represented on the grid
- **bcs** (*BoundariesBase*, optional) – If supplied, the returned function also enforces boundary conditions by setting the ghost cells to the correct values

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The

arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_full_data_setter(*bcs*)

Create a function to set the valid part of a full data array.

Parameters
bcs (*BoundariesBase*, optional) – If supplied, the returned function also enforces boundary conditions by setting the ghost cells to the correct values

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the field from which the grid and other information can be extracted
- **rng** (Generator) – Random number generator (default: `default_rng()`).

Return type
Callable[[], TNativeArray]

make_ghost_cell_setter(*bcs*)

Return function that sets the ghost cells on a full array.

Parameters
bcs (*BoundariesBase*) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns
Callable with signature `(data_full: NumericArray, args=None)`, which sets the ghost cells of the full data, potentially using additional information in *args* (e.g., the time *t* during solving a PDE)

Return type
GhostCellSetter

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
Function that takes two instance of native data arrays, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

Note that this function takes and returns data in the native representation of the backend. If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters
grid (GridBase) – Grid for which the operator is needed

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
Callable[[TNativeArray], TNativeArray]

make_interpolator(*field*, *, *fill=None*, *with_ghost_cells=False*)

Returns a function that can be used to interpolate values.

Parameters

- **field** (*DataFieldBase*) – Field for which the interpolator is defined
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.

Returns
A function which returns interpolated values when called with arbitrary positions within the space of the grid.

Return type
Callable[[FloatingArray, NumericArray], NumberOrArray]

make_mpi_synchronizer(*operator='MAX'*)

Return function that synchronizes values between multiple MPI processes.

> **Warning**
>
> The default implementation does not synchronize anything. This is simply a hook, which can be used by backends that support MPI

Parameters
operator (*str or int*) – Flag determining how the value from multiple nodes is combined. Possible values include “MAX”, “MIN”, and “SUM”.

Returns
Function that can be used to synchronize values across nodes

Return type
Callable[[float], float]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are ‘laplace’, ‘gradient’, or ‘divergence’. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*) – The boundary conditions used before the operator is applied
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **grid** (GridBase)
- **operator** (*str | OperatorInfo*)
- **bcs** (BoundariesBase)
- **dtype** (*DTypeLike | None*)

make_operator_no_bc(*grid*, *operator*, *, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator without boundary conditions.

A function that takes the discretized full data as an input and an array of valid data points to which the result of applying the operator is written.

> **Note**
>
> The resulting function does not check whether the ghost cells of the input array have been supplied with sensible values. It is the responsibility of the user to set the values of the ghost cells beforehand. Use this function only if you absolutely know what you're doing. In all other cases, *make_operator()* is probably the better choice.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns

the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray), so they *out* array need to be supplied explicitly.

Return type

callable

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This supports typically only supports products between two vector fields.

Parameters

field (*DataFieldBase*) – Field for which the outer product is defined

Returns

Function that takes two instance of native data arrays, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type

BinaryOperatorImplType

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (FieldBase) – An example for the state from which information can be extracted

Returns

Function returning deterministic part of the right hand side of the PDE

Return type
Callable[[TNativeArray, float], TNativeArray]

make_stepper(*solver*, *state*)

Create a field-based stepping function for a given solver.

Parameters

- **solver** (*SolverBase*) – The solver instance, which determines how the stepper is constructed
- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*.

Return type
StepperType

make_valid_data_setter(*grid*, *rank*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (GridBase) – Grid for which the data setter is defined
- **rank** (*int*) – Rank of the data represented on the grid

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

native_to_numpy(*value: TNativeArray*) → ndarray[Any, dtype[number]]

native_to_numpy(*value: TValue*) → TValue

Convert native values to numpy representation.

Parameters
value (*Any*) – The value to convert to numpy representation

Return type
Any

numpy_to_native(*value: ndarray[Any, dtype[number]]*) → TNativeArray

numpy_to_native(*value: TValue*) → TValue

Convert values from numpy to native representation.

Parameters
value (*Any*) – The value to convert from numpy representation

Return type
Any

classmethod register_operator(*grid_cls*, *name*, *factory_func=None*, *, *rank_in=0*, *rank_out=0*)

Register an operator for a particular grid.

Example

The method can either be used directly:

```
BackendClass.register_operator(grid_class, "operator", make_operator)
```

or as a decorator for the factory function:

```
@BackendClass.register_operator(grid_class, "operator")
def make_operator(grid: GridBase): ...
```

Parameters

- **grid_cls** (GridBase) – Grid class for which the operator is defined
- **name** (*str*) – The name of the operator to register
- **factory_func** (*callable*) – A function with signature (grid: GridBase, **kwargs), which takes a grid object and optional keyword arguments and returns an implementation of the given operator. This implementation is a function that takes a ndarray of discretized values as arguments and returns the resulting discretized data in a ndarray after applying the operator.
- **rank_in** (*int*) – The rank of the input field for the operator
- **rank_out** (*int*) – The rank of the field that is returned by the operator

supports_mpi: bool = False

This backend supports parallel simulations with MPI.

Type
bool

get_backend(*backend*, *config=None*)

Return backend specified by string of instance.

The returned backend is cached if *config* is not specified. Consequently, the same object will be returned for repeated calls to *get_backend*, which allows sharing configuration parameters. Moreover, if *name* only specifies a backend, i.e., does not contain a colon *:*, the configuration of this backend is linked with the global configuration pde.config, such that changes to the config are reflected globally.

Parameters

- **backend** (str or *BackendBase*) – Backend specified by name given as a string. If the string contains a colon, the first part determines the backend, whereas the second part can be used to convey additional information. For example, torch:cuda may load a torch backend and use a cuda device. As a special case, we also allow full backend objects, which are simply returned. This is a simple way to allow providing full backend objects in places where we otherwise would expect a backend name.
- **config** (*dict*) – Additional configuration options for this specific backend. The full configuration will be taken from the global configuration and merged with the given options here. This option is only permitted if *backend* is a string since there otherwise might be unintended side effects of modifying an existing backend.

Returns
An initialized backend

Return type
BackendBase

registered_backends()

Returns all registered backends.

Return type
list[str]

Subpackages:

4.1.1 pde.backends.jax package

Defines the `jax` backend.

JaxBackend	Defines `jax` backend.

Subpackages:

pde.backends.jax.operators package

Package collecting modules defining discretized operators using jax.

These operators can either be used directly or they are imported by the respective methods defined on fields and grids.

cartesian	This module implements differential operators on Cartesian grids.
cylindrical_sym	This module implements differential operators on cylindrical grids.
polar_sym	This module implements differential operators on polar grids.
spherical_sym	This module implements differential operators on spherical grids.

pde.backends.jax.operators.cartesian module

This module implements differential operators on Cartesian grids.

make_laplace	Make a Laplace operator on a Cartesian grid.
make_gradient	Make a gradient operator on a Cartesian grid.
make_divergence	Make a divergence operator on a Cartesian grid.
make_vector_gradient	Make a vector gradient operator on a Cartesian grid.
make_vector_laplace	Make a vector Laplacian on a Cartesian grid.
make_tensor_divergence	Make a tensor divergence operator on a Cartesian grid.

make_divergence(*grid*, *, *method='central'*)

Make a divergence operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *method='central'*)

Make a gradient operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, ***kwargs*)

Make a Laplace operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *method='central'*)

Make a tensor divergence operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *method='central'*)

Make a vector gradient operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_laplace(*grid*)

Make a vector Laplacian on a Cartesian grid.

Parameters
grid (*CartesianGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.jax.operators.cylindrical_sym module

This module implements differential operators on cylindrical grids.

make_laplace	Make a discretized laplace operator for a cylindrical grid.
make_gradient	Make a discretized gradient operator for a cylindrical grid.
make_gradient_squared	Make a discretized gradient squared operator for a cylindrical grid.
make_divergence	Make a discretized divergence operator for a cylindrical grid.
make_vector_gradient	Make a discretized vector gradient operator for a cylindrical grid.
make_vector_laplace	Make a discretized vector laplace operator for a cylindrical grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a cylindrical grid.

make_divergence(*grid*)

Make a discretized divergence operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*)

Make a discretized gradient operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient_squared(*grid*, *, *central=True*)

Make a discretized gradient squared operator for a cylindrical grid.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*)

Make a discretized laplace operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*)

Make a discretized tensor divergence operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*)

Make a discretized vector gradient operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_laplace(*grid*)

Make a discretized vector laplace operator for a cylindrical grid.

Parameters
grid (*CylindricalSymGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.jax.operators.polar_sym module

This module implements differential operators on polar grids.

make_laplace	Make a discretized laplace operator for a polar grid.
make_gradient	Make a discretized gradient operator for a polar grid.
make_gradient_squared	Make a discretized gradient squared operator for a polar grid.
make_divergence	Make a discretized divergence operator for a polar grid.
make_vector_gradient	Make a discretized vector gradient operator for a polar grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a polar grid.

make_divergence(*grid*)

Make a discretized divergence operator for a polar grid.

Parameters
grid (*PolarSymGrid*) – The polar grid for which this operator will be defined

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *method='central'*)

Make a discretized gradient operator for a polar grid.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient_squared(*grid*, *, *central=True*)

Make a discretized gradient squared operator for a polar grid.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*)

Make a discretized laplace operator for a polar grid.

Parameters
grid (*PolarSymGrid*) – The polar grid for which this operator will be defined

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*)

Make a discretized tensor divergence operator for a polar grid.

Parameters
grid (*PolarSymGrid*) – The polar grid for which this operator will be defined

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*)

Make a discretized vector gradient operator for a polar grid.

Parameters
grid (*PolarSymGrid*) – The polar grid for which this operator will be defined

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.jax.operators.spherical_sym module

This module implements differential operators on spherical grids.

make_laplace	Make a discretized laplace operator for a spherical grid.
make_gradient	Make a discretized gradient operator for a spherical grid.
make_gradient_squared	Make a discretized gradient squared operator for a spherical grid.
make_divergence	Make a discretized divergence operator for a spherical grid.
make_vector_gradient	Make a discretized vector gradient operator for a spherical grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a spherical grid.
make_tensor_double_divergence	Make a discretized tensor double divergence operator for a spherical grid.

make_divergence(*grid*, ***, *conservative=None*, *method='central'*)

Make a discretized divergence operator for a spherical grid.

> **Warning**
>
> This operator ignores the θ-component of the field when calculating the divergence. This is because the resulting scalar field could not be expressed on a `SphericalSymGrid`.

Parameters

- **grid** (*SphericalSymGrid*) – The polar grid for which this operator will be defined
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *method='central'*)

Make a discretized gradient operator for a spherical grid.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient_squared(*grid*, *, *central=True*)

Make a discretized gradient squared operator for a spherical grid.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, *, *conservative=None*)

Make a discretized laplace operator for a spherical grid.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined

- **conservative** (*bool*) – Flag indicating whether the laplace operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *conservative=False*)

Make a discretized tensor divergence operator for a spherical grid.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_double_divergence(*grid*, *, *conservative=None*)

Make a discretized tensor double divergence operator for a spherical grid.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *method='central'*)

Make a discretized vector gradient operator for a spherical grid.

Warning

This operator ignores the two angular components of the field when calculating the gradient. This is because the resulting field could not be expressed on a `SphericalSymGrid`.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.jax.backend module

Defines the `jax` backend class.

class JaxBackend(*config=None*, *, *name=None*, *device='config'*)

Bases: *BackendBase*[Array]

Defines `jax` backend.

Initialize the jax backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend
- **device** (*str*) – The jax device to use. Special values are "config" (read from configuration) and "auto" (use CUDA if available, otherwise CPU)

compile_function(*func*, ***kwargs*)

General method that compiles a user function.

Parameters

- **func** (*callable*) – The function that needs to be compiled for this backend
- ****kwargs** – Additional arguments forwarded to `jax.jit()`

Return type
TFunc

copy_data = True

Data must be copied from CPU numpy representation to a native device.

Type
bool

property device: Device

The currently assigned jax device.

classmethod from_args(*config*, *args=''*, *, *name=None*)

Initialize backend with extra arguments.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **args** (*str*) – Additional arguments that determine how the backend is initialized
- **name** (*str*) – The name of the backend

Return type
Self

get_jax_dtype(*dtype*)

Convert numpy dtype to jax-compatible dtype.

Parameters
dtype (*DTypeLike*) – numpy dtype to convert to corresponding jax dtype

Returns
A proper dtype usable for jax

Return type
np.dtype

implementation = 'jax'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

property info: dict[str, Any]

relevant information about the backend

Type
dict

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_full_data_setter(*bcs*)

Create a function to set the valid part of a full data array.

Parameters
bcs (*BoundariesBase*, optional) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **rng** (Generator) – Random number generator (default: default_rng()) used to initialize the seed.

Return type
Callable[[], jax.Array]

make_ghost_cell_setter(*bcs*)

Return function that sets the ghost cells on a full array.

Parameters
bcs (*BoundariesBase*) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns
Callable with signature `(data_full: NumericArray, args=None)`, which sets the ghost cells of the full data, potentially using additional information in *args* (e.g., the time *t* during solving a PDE)

Return type
JaxGhostCellSetter

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
Callable[[jax.Array, jax.Array, jax.Array | None], jax.Array]

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

Parameters
grid (`GridBase`) – Grid for which the integrator is defined

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
Callable[[jax.Array], jax.Array]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **grid** (`GridBase`) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are ‘laplace’, ‘gradient’, or ‘divergence’. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*, optional) – The boundary conditions used before the operator is applied

- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time. When this backend is used together with JAX' just-in-time compilation (e.g. via `jax.jit()`), the values passed through *args* need to be compatible with JAX's JIT tracing rules.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **grid** (GridBase)
- **operator** (*str | OperatorInfo*)
- **bcs** (BoundariesBase)
- **dtype** (*DTypeLike | None*)

make_operator_no_bc(*grid*, *operator*, *, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator without boundary conditions.

A function that takes the discretized full data as an input and an array of valid data points to which the result of applying the operator is written.

> **Note**
>
> The resulting function does not check whether the ghost cells of the input array have been supplied with sensible values. It is the responsibility of the user to set the values of the ghost cells beforehand. Use this function only if you absolutely know what you're doing. In all other cases, *make_operator()* is probably the better choice.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray), so they *out* array need to be supplied explicitly.

Return type
callable

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This typically only supports products between two vector fields.

Parameters
field (*DataFieldBase*) – Field for which the outer product is defined

Returns
function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
Callable[[jax.Array, jax.Array, jax.Array | None], jax.Array]

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (`FieldBase`) – An example for the state from which information can be extracted

Returns
Function returning deterministic part of the right hand side of the PDE.

Return type
Callable[[jax.Array, float], jax.Array]

make_stepper(*solver*, *state*)

Create a field-based stepping function for a given solver.

Parameters

- **solver** (*SolverBase*) – The solver instance, which determines how the stepper is constructed
- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type
StepperType

make_valid_data_setter(*grid*, *rank*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (`GridBase`) – The grid for which the data setter is created
- **rank** (*int*) – Rank of the data represented on the grid.

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

native_to_numpy(*value*)
Convert native values to numpy representation.

Parameters
value (*Any*)

Return type
Any

numpy_to_native(*value*)
Convert values from numpy to jax representation.

This method also ensures that the value is copied to the selected device.

Parameters
value (*Any*)

Return type
Any

pde.backends.jax.config module

Defines the configuration for the jax backend.

This configuration file will be imported without importing the enclosed module (which will instead be loaded on demand). Consequently, the module should use absolute references to other modules and not import anything from the backend.

pde.backends.jax.typing module

Provides support for mypy type checking of the module.

JaxOperatorType	An operator that acts on an array.
JaxDataSetter	
JaxGhostCellSetter	
JaxVirtualPointEvaluator	

class JaxDataSetter(**args*, ***kwargs*)
Bases: Protocol

class JaxGhostCellSetter(**args*, ***kwargs*)
Bases: Protocol

class JaxInnerStepperType(**args*, ***kwargs*)
Bases: Protocol

General backend-level stepping-function type working with jax arrays.

class JaxOperatorType(**args*, ***kwargs*)
Bases: Protocol

An operator that acts on an array.

class JaxVirtualPointEvaluator(**args*, ***kwargs*)

Bases: Protocol

4.1.2 pde.backends.numba package

Defines the numba backend.

NumbaBackend	Defines numba backend.

Subpackages:

pde.backends.numba.operators package

Package collecting modules defining discretized operators using numba.

These operators can either be used directly or they are imported by the respective methods defined on fields and grids.

cartesian	This module implements differential operators on Cartesian grids.
cylindrical_sym	This module implements differential operators on cylindrical grids.
polar_sym	This module implements differential operators on polar grids.
spherical_sym	This module implements differential operators on spherical grids.

pde.backends.numba.operators.cartesian module

This module implements differential operators on Cartesian grids.

make_laplace	Make a Laplace operator on a Cartesian grid.
make_gradient	Make a gradient operator on a Cartesian grid.
make_divergence	Make a divergence operator on a Cartesian grid.
make_vector_gradient	Make a vector gradient operator on a Cartesian grid.
make_vector_laplace	Make a vector Laplacian on a Cartesian grid.
make_tensor_divergence	Make a tensor divergence operator on a Cartesian grid.

make_divergence(*grid*, ***, *backend=None*, *method='central'*)

Make a divergence operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_gradient(*grid*, *, *backend=None*, *method='central'*)

Make a gradient operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_laplace(*grid*, *, *backend=None*, *spectral=None*, ***kwargs*)

Make a Laplace operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **spectral** (*bool or None*) – Flag deciding whether a spectral implementation is used. If *None*, the value is controlled by the configuration.
- ****kwargs** – Specifies extra arguments influencing how the operator is created. Note that some laplace operators support the *corner_weight* argument, which allows setting weighting factors for corner points of the stencil.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_tensor_divergence(*grid*, *, *backend=None*, *method='central'*)

Make a tensor divergence operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_vector_gradient(*grid*, *, *backend=None*, *method='central'*)

Make a vector gradient operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created

- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_laplace(*grid*, *, *backend=None*)

Make a vector Laplacian on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.numba.operators.common module

Common functions that are used by many operators.

make_derivative(*grid*, *axis=0*, *, *method='central'*, *backend=None*)

Make a derivative operator along a single axis using numba compilation.

Parameters

- **grid** (*GridBase*) – The grid for which the operator is created
- **axis** (*int*) – The axis along which the derivative will be taken
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an full array of values including those at ghost cells. The result will be an array of the same shape containing the actual derivatives at the valid (interior) grid points.

Return type
OperatorImplType

make_derivative2(*grid*, *axis=0*, *, *backend=None*)

Make a second-order derivative operator along a single axis.

Parameters

- **grid** (*GridBase*) – The grid for which the operator is created
- **axis** (*int*) – The axis along which the derivative will be taken
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an full array of values including those at ghost cells. The result will be an array of the same shape containing the actual derivatives at the valid (interior) grid points.

Return type
OperatorImplType

pde.backends.numba.operators.cylindrical_sym module

This module implements differential operators on cylindrical grids.

make_laplace	Make a discretized laplace operator for a cylindrical grid.
make_gradient	Make a discretized gradient operator for a cylindrical grid.
make_divergence	Make a discretized divergence operator for a cylindrical grid.
make_vector_gradient	Make a discretized vector gradient operator for a cylindrical grid.
make_vector_laplace	Make a discretized vector laplace operator for a cylindrical grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a cylindrical grid.

make_divergence(*grid*, *, *backend=None*)

Make a discretized divergence operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate *z*. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *backend=None*)

Make a discretized gradient operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate *z*. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient_squared(*grid*, *, *backend=None*, *central=True*)

Make a discretized gradient squared operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, *, *backend=None*)

Make a discretized laplace operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *backend=None*)

Make a discretized tensor divergence operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *backend=None*)

Make a discretized vector gradient operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_laplace(*grid*, *, *backend=None*)

Make a discretized vector laplace operator for a cylindrical grid.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **grid** (*CylindricalSymGrid*) – The grid for which the operator is created
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.numba.operators.polar_sym module

This module implements differential operators on polar grids.

make_laplace	Make a discretized laplace operator for a polar grid.
make_gradient	Make a discretized gradient operator for a polar grid.
make_gradient_squared	Make a discretized gradient squared operator for a polar grid.
make_divergence	Make a discretized divergence operator for a polar grid.
make_vector_gradient	Make a discretized vector gradient operator for a polar grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a polar grid.

make_divergence(*grid*, *, *backend=None*)

Make a discretized divergence operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *backend=None*, *method='central'*)

Make a discretized gradient operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient_squared(*grid*, *, *backend=None*, *central=True*)

Make a discretized gradient squared operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, *, *backend=None*)

Make a discretized laplace operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *backend=None*)

Make a discretized tensor divergence operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *backend=None*)

Make a discretized vector gradient operator for a polar grid.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*PolarSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.numba.operators.spherical_sym module

This module implements differential operators on spherical grids.

make_laplace	Make a discretized laplace operator for a spherical grid.
make_gradient	Make a discretized gradient operator for a spherical grid.
make_gradient_squared	Make a discretized gradient squared operator for a spherical grid.
make_divergence	Make a discretized divergence operator for a spherical grid.
make_vector_gradient	Make a discretized vector gradient operator for a spherical grid.
make_tensor_divergence	Make a discretized tensor divergence operator for a spherical grid.
make_tensor_double_divergence	Make a discretized tensor double divergence operator for a spherical grid.

make_divergence(*grid*, ***, *backend=None*, *safe=None*, *conservative=None*, *method='central'*)

Make a discretized divergence operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Warning

This operator ignores the θ-component of the field when calculating the divergence. This is because the resulting scalar field could not be expressed on a `SphericalSymGrid`.

Parameters

- **grid** (*SphericalSymGrid*) – The polar grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **safe** (*bool*) – Add extra checks for the validity of the input. If *None*. the value is read from the configuration option *operators.tensor_symmetry_check*.
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_gradient(*grid*, ***, *backend=None*, *method='central'*)

Make a discretized gradient operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns

A function that can be applied to an array of values

Return type

OperatorImplType

make_gradient_squared(*grid*, ***, *backend=None*, *central=True*)

Make a discretized gradient squared operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **central** (*bool*) – Whether a central difference approximation is used for the gradient operator. If this is False, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, *, *backend=None*, *conservative=None*)

Make a discretized laplace operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **conservative** (*bool*) – Flag indicating whether the laplace operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *backend=None*, *safe=None*, *conservative=False*)

Make a discretized tensor divergence operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **safe** (*bool*) – Add extra checks for the validity of the input. If *None*. the value is read from the configuration option *operators.tensor_symmetry_check*.
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_double_divergence(*grid*, *, *backend=None*, *safe=None*, *conservative=None*)

Make a discretized tensor double divergence operator for a spherical grid.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **safe** (*bool*) – Add extra checks for the validity of the input. If *None*. the value is read from the configuration option *operators.tensor_symmetry_check*.
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *backend=None*, *method='central'*, *safe=None*)

Make a discretized vector gradient operator for a spherical grid.

Warning

This operator ignores the two angular components of the field when calculating the gradient. This is because the resulting field could not be expressed on a `SphericalSymGrid`.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The spherical grid for which this operator will be defined
- **backend** (*NumbaBackend*) – References to the backend to read configuration details
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.
- **safe** (*bool*) – Add extra checks for the validity of the input. If *None*. the value is read from the configuration option *operators.tensor_symmetry_check*.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.numba.backend module

Defines the `numba` backend class.

class NumbaBackend(*config*, *, *name=None*)

Bases: *NumpyBackend*

Defines `numba` backend.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

compile_function(*func*, ***kwargs*)

General method that compiles a user function.

Parameters

- **func** (*callable*) – The function that needs to be compiled for this backend
- ****kwargs** – Additional arguments forwarded to *pde.backends.numba.utils.jit()*

Return type
TFunc

get_operator_info(*grid*, *operator*)

Return the operator defined for this backend.

Parameters

- **grid** (`GridBase`) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.

Returns
information for the operator

Return type
`OperatorInfo`

get_registered_operators(*grid_id*)

Returns all operators defined for a backend.

Parameters
grid_id (`GridBase` or its type) – Grid for which the operator need to be returned

Return type
set[str]

implementation = 'numba'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the state from which the grid and other information can be extracted.
- **rng** (Generator) – Random number generator (default: `default_rng()`). Not used in this numba backend.

Return type
Callable[[], NumericArray]

make_ghost_cell_setter(*bcs*)

Return function that sets the ghost cells on a full array.

Parameters
bcs (*BoundariesBase*) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns
Callable with signature `(data_full: NumericArray, args=None)`, which sets the ghost cells of the full data, potentially using additional information in *args* (e.g., the time *t* during solving a PDE)

Return type
GhostCellSetter

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_inserter(*grid*, *, *with_ghost_cells=False*)

Return a compiled function to insert values at interpolated positions.

Parameters

- **grid** (GridBase) – Grid for which the integrator is defined
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the grid points. If this is the case, the boundaries are not checked and the coordinates are used as is.

Returns

A function with signature (data, position, amount), where *data* is the numpy array containing the field data, position is denotes the position in grid coordinates, and *amount* is the that is to be added to the field.

Return type

callable

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters

grid (GridBase) – Grid for which the integrator is defined

Returns

A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type

Callable[[NumericArray], NumberOrArray]

make_interpolator(*field*, *, *fill=None*, *with_ghost_cells=False*)

Returns a function that can be used to interpolate values.

Parameters

- **field** (*DataFieldBase*) – Field for which the interpolator is defined
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.

Returns

A function which returns interpolated values when called with arbitrary positions within the space of the grid.

Return type

Callable[[FloatingArray, NumericArray], NumberOrArray]

make_mpi_synchronizer(*operator='MAX'*)

Return function that synchronizes values between multiple MPI processes.

> **Warning**
>
> The default implementation does not synchronize anything. This is simply a hook, which can be used by backends that support MPI

Parameters
operator (*str or int*) – Flag determining how the value from multiple nodes is combined. Possible values include “MAX”, “MIN”, and “SUM”.

Returns
Function that can be used to synchronize values across nodes

Return type
Callable[[float], float]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are ‘laplace’, ‘gradient’, or ‘divergence’. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*, optional) – The boundary conditions used before the operator is applied
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time. Note that since the returned operator will always be compiled by Numba, the arguments need to be compatible with Numba. The following example shows how to pass the current time *t*:

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **grid** (GridBase)
- **operator** (*str | OperatorInfo*)
- **bcs** (BoundariesBase)

- **dtype** (*DTypeLike | None*)

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This supports typically only supports products between two vector fields.

Parameters

field (*DataFieldBase*) – Field for which the outer product is defined

Returns

function that takes two instance of ndarray, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type

BinaryOperatorImplType

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (FieldBase) – An example for the state from which information can be extracted

Returns

Function returning deterministic part of the right hand side of the PDE

Return type

Callable[[NumericArray, float], NumericArray]

make_stepper(*solver*, *state*)

Create a field-based stepping function for a given solver.

Parameters

- **solver** (*SolverBase*) – The solver instance, which determines how the stepper is constructed
- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted

Returns

Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type

StepperType

use_multithreading()

Determine whether multithreading should be used in numba-compiled code.

This method checks the configuration setting for *numba.multithreading* and determines whether multithreading should be enabled based on the value of this setting. The possible values for *numba.multithreading* are: - ‘always’: Multithreading is always enabled. - ‘never’: Multithreading is never enabled. - ‘only_local’: Multithreading is enabled only if the code is not running in a high-performance computing (HPC) environment.

Returns

True if multithreading should be enabled, False otherwise.

Return type

bool

Raises

- **ValueError** – If the *numba.multithreading* setting is not one of the expected
- **values ('always', 'never', 'only_local').** –

pde.backends.numba.config module

Defines the configuration for the numba backend.

This configuration file will be imported without importing the enclosed module (which will instead be loaded on demand). Consequently, the module should use absolute references to other modules and not import anything from the backend.

pde.backends.numba.grids module

Compiled functions for dealing with grids.

get_grid_numba_type	Return numba type corresponding to a particular grid.
make_cell_volume_getter	Return a compiled function returning the volume of a grid cell.
make_interpolation_axis_data	Factory for obtaining interpolation information.
make_single_interpolator	Return a compiled function for linear interpolation on the grid.

get_grid_numba_type(*grid*, *rank=0*)

Return numba type corresponding to a particular grid.

Parameters

- **grid** (GridBase) – The grid for which we determine the type
- **rank** (*int*) – The rank of the data stored in the grid

Returns

description

Return type

type

make_cell_volume_getter(*grid*, *, *flat_index=False*, *backend=None*)

Return a compiled function returning the volume of a grid cell.

Parameters

- **grid** (GridBase) – Grid for which the function is defined
- **flat_index** (*bool*) – When True, cell_volumes are indexed by a single integer into the flattened array.
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns

returning the volume of the chosen cell

Return type

function

make_interpolation_axis_data(*grid*, *axis*, *, *with_ghost_cells=False*, *cell_coords=False*)

Factory for obtaining interpolation information.

Parameters

- **grid** (GridBase) – Grid for which the interpolator is defined
- **axis** (*int*) – The axis along which interpolation is performed
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.
- **cell_coords** (*bool*) – Flag indicating whether points are given in cell coordinates or actual point coordinates.

Returns
A function that is called with a coordinate value for the axis. The function returns the indices of the neighboring support points as well as the associated weights.

Return type
callable

make_single_interpolator(*grid*, *, *fill=None*, *with_ghost_cells=False*, *cell_coords=False*, *backend=None*)

Return a compiled function for linear interpolation on the grid.

Parameters

- **grid** (GridBase) – Grid for which the interpolator is defined
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.
- **cell_coords** (*bool*) – Flag indicating whether points are given in cell coordinates or actual point coordinates.
- **backend** (*NumbaBackend*) – References to the backend to read configuration details

Returns
A function which returns interpolated values when called with arbitrary positions within the space of the grid. The signature of this function is (data, point), where *data* is the numpy array containing the field data and position denotes the position in grid coordinates.

Return type
callable

pde.backends.numba.overloads module

Defines functions overloads, so numba can use them.

class OnlineStatistics(**args*, ***kwargs*)

Bases: *OnlineStatistics*

Class for using an online algorithm for calculating statistics.

```
class_type = jitclass.OnlineStatistics#72190774ef10<min:float64,max:float64,
mean:float64,_mean2:float64,count:uint64>
```

count: int

recorded number of items

Type
int

mean: float

recorded mean

Type
float

pde.backends.numba.utils module

Defines utilities for the numba backend.

make_get_valid	Create a function to extract the valid part of a full data array.
make_get_arr_1d	Create function that extracts a 1d array at a given position.
numba_environment	Return information about the numba setup used.
jit	Apply nb.jit with predefined arguments.
make_array_constructor	Returns an array within a jitted function using basic information.
numba_dict	Converts a python dictionary to a numba typed dictionary.
get_common_numba_dtype	Returns a numba numerical type in which all arrays can be represented.
random_seed	Sets the seed of the random number generator of numpy and numba.

class Counter(*value=0*)

Bases: object

Helper class for implementing JIT_COUNT.

We cannot use a simple integer for this, since integers are immutable, so if one imports JIT_COUNT from this module it would always stay at the fixed value it had when it was first imported. The workaround would be to import the symbol every time the counter is read, but this is error-prone. Instead, we implement a thin wrapper class around an int, which only supports reading and incrementing the value. Since this object is now mutable it can be used easily. A disadvantage is that the object needs to be converted to int before it can be used in most expressions.

Parameters
value (*int*)

increment()

flat_idx(*arr*, *i*)

Helper function allowing indexing of scalars as if they arrays.

Parameters

- **arr** (ndarray or scalar) – Array or scalar value to index
- **i** (*int*) – Index to access

Return type
Number

get_common_numba_dtype(**args*)

Returns a numba numerical type in which all arrays can be represented.

Parameters
***args** – All items to be tested

Returns: numba.complex128 if any entry is complex, otherwise numba.double

jit(*function*, *signature=None*, *parallel=False*, *, *backend=None*, **kwargs*)

Apply nb.jit with predefined arguments.

Parameters

- **function** (*TFunc*) – The function which will be jitted
- **signature** – Signature of the function to compile
- **parallel** (*bool*) – Allow parallel compilation of the function
- **backend** (NumbaBackend | *None*) – The backend from which compilation options will be read. If *None*, the default numba backend will be used.
- ****kwargs** – Additional arguments to `numba.jit()`

Returns

Function that will be compiled using numba

Return type

TFunc

make_array_constructor(*arr*)

Returns an array within a jitted function using basic information.

Parameters

arr (ndarray) – The array that should be accessible within jit

Return type

Callable[[], NumericArray]

Warning

A reference to the array needs to be retained outside the numba code to prevent garbage collection from removing the array

make_get_arr_1d(*dim*, *axis*)

Create function that extracts a 1d array at a given position.

Parameters

- **dim** (*int*) – The dimension of the space, i.e., the number of axes in the supplied array
- **axis** (*int*) – The axis that is returned as the 1d array

Returns

A numba compiled function that takes the full array *arr* and an index *idx* (a tuple of *dim* integers) specifying the point where the 1d array is extract. The function returns a tuple (arr_1d, i, bc_i), where *arr_1d* is the 1d array, *i* is the index *i* into this array marking the current point and *bc_i* are the remaining components of *idx*, which locate the point in the orthogonal directions. Consequently, *i = idx[axis]* and *arr[..., idx] == arr_1d[..., i]*.

Return type

function

make_get_valid(*grid*)

Create a function to extract the valid part of a full data array.

Parameters
grid (*GridBase*) – The grid for which the function is created

Returns

Mapping a numpy array containing the full data of the grid to a
numpy array of only the valid data

Return type
callable

numba_dict(*data=None*, */ (Positional-only parameter separator (PEP 570))*, ***kwargs*)

Converts a python dictionary to a numba typed dictionary.

Parameters

- **data** (*dict, optional*) – Data to be converted to a dictionary. If None, an empty dictionary is created.
- ****kwargs** – Additional items added to the dictionary

Returns
A dictionary of numba type

Return type
Dict

numba_environment(*backend=None*)

Return information about the numba setup used.

Parameters
backend (*NumbaBackend | None*) – The backend from which compilation options will be read. If *None*, the default numba backend will be used.

Returns
(dict) information about the numba setup

Return type
dict[str, *Any*]

ol_flat_idx(*arr*, *i*)

Helper function allowing indexing of scalars as if they arrays.

random_seed(*seed=0*)

Sets the seed of the random number generator of numpy and numba.

Parameters
seed (*int*) – Sets random seed

Return type
None

4.1.3 pde.backends.numba_mpi package

Defines the `numba_mpi` backend, which supports MPI parallelism

NumbaMPIBackend	Defines MPI-compatible numba backend.

pde.backends.numba_mpi.backend module

Defines a numba backend class that support MPI parallelism.

class NumbaMPIBackend(*config*, *, *name=None*)

Bases: *NumbaBackend*

Defines MPI-compatible numba backend.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

config_inheritance = ['numba']

Additional backends that are queried for configuration parameters.

Type
list

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters
grid (GridBase) – Grid for which the integrator is defined

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
Callable[[NumericArray], NumberOrArray]

make_mpi_synchronizer(*operator='MAX'*)

Return function that synchronizes values between multiple MPI processes.

Warning

The default implementation does not synchronize anything. This is simply a hook, which can be used by backends that support MPI

Parameters
operator (*str or int*) – Flag determining how the value from multiple nodes is combined. Possible values include “MAX”, “MIN”, and “SUM”.

Returns
Function that can be used to synchronize values across nodes

Return type
Callable[[float], float]

supports_mpi = True

This backend supports parallel simulations with MPI.

Type
bool

pde.backends.numba_mpi.overloads module

Defines MPI functions overloads, so numba can use them.

ol_mpi_allreduce(*data*, *operator*)
Overload the *mpi_allreduce* function.

ol_mpi_recv(*data*, *source*, *tag*)
Overload the *mpi_recv* function.

Parameters

- **source** (*int*)
- **tag** (*int*)

ol_mpi_send(*data*, *dest*, *tag*)
Overload the *mpi_send* function.

Parameters

- **dest** (*int*)
- **tag** (*int*)

4.1.4 pde.backends.numpy package

Defines the numpy backend.

NumpyBackend	Defines numpy backend, from which all other backends inherit.

pde.backends.numpy.backend module

Defines base class of backends that implement computations.

class NumpyBackend(*config*, *, *name=None*)
Bases: *BackendBase*[ndarray[Any, dtype[number]]]

Defines numpy backend, from which all other backends inherit.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

compile_function(*func*, **kwargs*)
General method that compiles a user function.

Parameters
func (*callable*) – The function that needs to be compiled for this backend

Return type
TFunc

implementation = 'numpy'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_full_data_setter(*bcs*)

Create a function to set the valid part of a full data array.

Parameters
bcs (*BoundariesBase*, optional) – If supplied, the returned function also enforces boundary conditions by setting the ghost cells to the correct values

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **rng** (Generator) – Random number generator (default: default_rng()).

Return type
Callable[[], NumericArray]

make_ghost_cell_setter(*bcs*)

Return function that sets the ghost cells on a full array.

Parameters
bcs (*BoundariesBase*) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns
Callable with signature (data_full: NumericArray, args=None), which sets the ghost

cells of the full data, potentially using additional information in *args* (e.g., the time *t* during solving a PDE)

Return type
GhostCellSetter

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters
grid (`GridBase`) – Grid for which the operator is needed

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
Callable[[NumericArray], NumberOrArray]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **grid** (`GridBase`) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*, optional) – The boundary conditions used before the operator is applied
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports

an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **grid** (*GridBase*)
- **operator** (*str | OperatorInfo*)
- **bcs** (*BoundariesBase*)
- **dtype** (*DTypeLike | None*)

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This supports typically only supports products between two vector fields.

Parameters
field (*DataFieldBase*) – Field for which the outer product is defined

Returns
function that takes two instance of ndarray, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (*FieldBase*) – An example for the state from which information can be extracted

Returns
Function returning deterministic part of the right hand side of the PDE

Return type
Callable[[NumericArray, float], NumericArray]

make_valid_data_setter(*grid*, *rank*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (*GridBase*) – The grid for which the data setter is created
- **rank** (*int*) – Rank of the data represented on the grid.

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The

arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

4.1.5 pde.backends.scipy package

Defines the `scipy` backend.

ScipyBackend	Defines `scipy` backend.

Subpackages:

pde.backends.scipy.operators package

Package collecting modules defining discretized operators for different grids.

These operators can either be used directly or they are imported by the respective methods defined on fields and grids.

cartesian	This module implements differential operators on Cartesian grids using scipy.
cylindrical_sym	This module implements differential operators on cylindrical grids.
polar_sym	This module implements differential operators on polar grids.
spherical_sym	This module implements differential operators on spherical grids.

pde.backends.scipy.operators.cartesian module

This module implements differential operators on Cartesian grids using scipy.

make_laplace	Make a Laplace operator using the scipy module.
make_gradient	Make a gradient operator using the scipy module.
make_divergence	Make a divergence operator using the scipy module.
make_vector_gradient	Make a vector gradient operator on a Cartesian grid.
make_vector_laplace	Make a vector Laplacian on a Cartesian grid.
make_tensor_divergence	Make a tensor divergence operator on a Cartesian grid.
make_poisson_solver	Make a operator that solves Poisson's equation.

make_divergence(*grid*, ***, *method='central'*)

Make a divergence operator using the scipy module.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_gradient(*grid*, *, *method='central'*)

Make a gradient operator using the scipy module.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_laplace(*grid*, **kwargs*)

Make a Laplace operator using the scipy module.

This only supports uniform discretizations.

Parameters
grid (*CartesianGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_poisson_solver(*bcs*, *, *method='auto'*)

Make a operator that solves Poisson's equation.

Parameters

- **bcs** (*BoundariesList*) – {ARG_BOUNDARIES_INSTANCE}
- **method** (*str*) – Method used for calculating the tensor divergence operator. If method='auto', a suitable method is chosen automatically.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_tensor_divergence(*grid*, *, *method='central'*)

Make a tensor divergence operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_gradient(*grid*, *, *method='central'*)

Make a vector gradient operator on a Cartesian grid.

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

make_vector_laplace(*grid*)

Make a vector Laplacian on a Cartesian grid.

Parameters
grid (*CartesianGrid*) – The grid for which the operator is created

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.scipy.operators.common module

Common functions that are used by many operators.

make_general_poisson_solver(*matrix*, *vector*, *method='auto'*)

Make an operator that solves Poisson's problem.

Parameters

- **matrix** – The (sparse) matrix representing the laplace operator on the given grid.
- **vector** – The constant part representing the boundary conditions of the Laplace operator.
- **method** (*str*) – The chosen method for implementing the operator

Returns
A function that can be applied to an array of values to obtain the solution to Poisson's equation where the array is used as the right hand side

Return type
OperatorImplType

make_laplace_from_matrix(*matrix*, *vector*)

Make a Laplace operator using matrix vector products.

Parameters

- **matrix** – (Sparse) matrix representing the laplace operator on the given grid
- **vector** – Constant part representing the boundary conditions of the Laplace operator

Returns
A function that can be applied to an array of values to obtain the result of applying the linear operator *matrix* and the offset given by *vector*.

Return type
Callable[[NumericArray, NumericArray | None], NumericArray]

uniform_discretization(*grid*)

Returns the uniform discretization or raises RuntimeError.

Parameters
grid (*GridBase*) – The grid whose discretization is tested

Raises
RuntimeError –

Returns
the common discretization of all axes

Return type
float

pde.backends.scipy.operators.cylindrical_sym module

This module implements differential operators on cylindrical grids.

make_poisson_solver	Make a operator that solves Poisson's equation.

make_poisson_solver(*bcs*, *, *method='auto'*)

Make a operator that solves Poisson's equation.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate r and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Parameters

- **bcs** (*BoundariesList*) – Specifies the boundary conditions applied to the field. This must be an instance of *BoundariesList*, which can be created from various data formats using the class method *from_data()*.
- **method** (*str*) – The chosen method for implementing the operator

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.scipy.operators.polar_sym module

This module implements differential operators on polar grids.

make_poisson_solver	Make a operator that solves Poisson's equation.

make_poisson_solver(*bcs*, *, *method='auto'*)

Make a operator that solves Poisson's equation.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **bcs** (*BoundariesList*) – Specifies the boundary conditions applied to the field. This must be an instance of *BoundariesList*, which can be created from various data formats using the class method *from_data()*.
- **method** (*str*) – The chosen method for implementing the operator

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.scipy.operators.spherical_sym module

This module implements differential operators on spherical grids.

make_poisson_solver	Make a operator that solves Poisson's equation.

make_poisson_solver(*bcs*, *, *method='auto'*)

Make a operator that solves Poisson's equation.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Parameters

- **bcs** (*BoundariesList*) – Specifies the boundary conditions applied to the field. This must be an instance of *BoundariesList*, which can be created from various data formats using the class method *from_data()*.
- **method** (*str*) – The chosen method for implementing the operator

Returns
A function that can be applied to an array of values

Return type
OperatorImplType

pde.backends.scipy.backend module

Defines the scipy backend class.

class ScipyBackend(*config*, *, *name=None*)

Bases: *NumpyBackend*

Defines `scipy` backend.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

4.1.6 pde.backends.torch package

Defines the `torch` backend.

TorchBackend	Defines `torch` backend.

Subpackages:

pde.backends.torch.operators package

Package collecting modules defining discretized operators using torch.

These operators can either be used directly or they are imported by the respective methods defined on fields and grids.

cartesian	This module implements differential operators on Cartesian grids.
cylindrical_sym	This module implements differential operators on spherical grids.
polar_sym	This module implements differential operators on polar grids.
spherical_sym	This module implements differential operators on spherical grids.

pde.backends.torch.operators.cartesian module

This module implements differential operators on Cartesian grids.

CartesianLaplacian	Cartesian Laplace using torch.
CartesianGradient	Cartesian gradient operator using torch.
CartesianGradientSquared	Cartesian gradient squared operator using torch.
CartesianDivergence	Cartesian divergence operator using torch.
CartesianVectorGradient	Cartesian vector gradient operator using torch.
CartesianVectorLaplacian	Cartesian vector Laplacian operator using torch.
CartesianTensorDivergence	Cartesian tensor divergence operator using torch.

class CartesianDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian divergence operator using torch.

Initialize the Cartesian divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters

arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class CartesianGradient(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian gradient operator using torch.

Initialize the Cartesian gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CartesianGradientSquared(*grid*, *bcs*, *, *central=True*, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian gradient squared operator using torch.

Initialize the Cartesian gradient squared operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **central** (*bool*) – Whether to use central differences. If *False*, forward and backward differences are used.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CartesianLaplacian(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian Laplace using torch.

Initialize the Cartesian Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CartesianTensorDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian tensor divergence operator using torch.

Initialize the Cartesian tensor divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 2

The rank of the input tensor

Type
int

class CartesianVectorGradient(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian vector gradient operator using torch.

Initialize the Cartesian vector gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class CartesianVectorLaplacian(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cartesian vector Laplacian operator using torch.

Initialize the Cartesian vector Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

pde.backends.torch.operators.common module

This module implements infrastructure for differential operators using torch.

TorchDifferentialOperator	Base class for differential operators implemented in torch.

class TorchDifferentialOperator(*grid*, *bcs*, *, *dtype*)

Bases: *TorchOperatorBase*

Base class for differential operators implemented in torch.

Initialize the torch operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced and it is assumed that the operator is applied to the full field.
- **dtype** (*DTypeLike*) – The data type of the field using the numpy convention

data_full: Tensor

get_full_data(*arr*, *args=None*)

Get full data array including ghost cells.

Parameters

- **arr** (torch.Tensor) – The input data. If boundary conditions are applied, this should contain only the valid grid points. Otherwise, it should already include ghost cells.
- **args** – Additional arguments passed to ghost cell setters, e.g., the time *t*.

Returns
The full data array including ghost cells with boundary conditions applied if necessary.

Return type
torch.Tensor

rank_in: int = 0

The rank of the input tensor

Type
int

set_valid(*arr*)

Set valid data in the internal full array.

Parameters
arr (torch.Tensor) – The data of the valid grid points

Return type
None

class TorchIntegralOperator(*grid*, *, *dtype*)

Bases: *TorchOperatorBase*

Operator integrating a field implemented in torch.

Initialize the torch operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **dtype** (*DTypeLike*) – The data type of the field

forward(*arr*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

pde.backends.torch.operators.cylindrical_sym module

This module implements differential operators on spherical grids.

CylindricalLaplacian	Cylindrical Laplace using torch.
CylindricalGradient	Cylindrical gradient operator using torch.
CylindricalGradientSquared	Cylindrical gradient squared operator using torch.
CylindricalDivergence	Cylindrical divergence operator using torch.
CylindricalVectorGradient	Cylindrical vector gradient operator using torch.
CylindricalVectorLaplacian	Cylindrical vector Laplacian operator using torch.
CylindricalTensorDivergence	Cylindrical tensor divergence operator using torch.

class CylindricalDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical divergence operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate r and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class CylindricalGradient(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical gradient operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CylindricalGradientSquared(*grid*, *bcs*, *, *central=True*, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical gradient squared operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical gradient squared operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **central** (*bool*) – Whether to use central differences. If *False*, forward and backward differences are used.

- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CylindricalLaplacian(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical Laplace using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class CylindricalTensorDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical tensor divergence operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical tensor divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 2

The rank of the input tensor

Type
int

class CylindricalVectorGradient(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical vector gradient operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate z. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical vector gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class CylindricalVectorLaplacian(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Cylindrical vector Laplacian operator using torch.

The cylindrical grid assumes polar symmetry, so that fields only depend on the radial coordinate *r* and the axial coordinate *z*. Here, the first axis is along the radius, while the second axis is along the axis of the cylinder. The radial discretization is defined as $r_i = (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$.

Initialize the Cylindrical vector Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

pde.backends.torch.operators.polar_sym module

This module implements differential operators on polar grids.

PolarLaplacian	Polar Laplace using torch.
PolarGradient	Polar gradient operator using torch.
PolarGradientSquared	Polar gradient squared operator using torch.
PolarDivergence	Polar divergence operator using torch.
PolarVectorGradient	Polar vector gradient operator using torch.
PolarTensorDivergence	Polar tensor divergence operator using torch.

class PolarDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Polar divergence operator using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\text{min}} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where r_{min} is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\text{max}} = r_{\text{min}} + N_r \Delta r$.

Initialize the Polar divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class PolarGradient(*grid*, *bcs*, *, *dtype*, *method='central'*)

Bases: *TorchDifferentialOperator*

Polar gradient operator using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Polar gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class PolarGradientSquared(*grid*, *bcs*, *, *central=True*, *dtype*)

Bases: *TorchDifferentialOperator*

Polar gradient squared operator using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Polar gradient squared operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **central** (*bool*) – Whether to use central differences. If *False*, forward and backward differences are used.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class PolarLaplacian(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Polar Laplace using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Polar Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class PolarTensorDivergence(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Polar tensor divergence operator using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Polar tensor divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 2

The rank of the input tensor

Type
int

class PolarVectorGradient(*grid*, *bcs*, *, *dtype*)

Bases: *TorchDifferentialOperator*

Polar vector gradient operator using torch.

The polar grid assumes polar symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Polar vector gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

pde.backends.torch.operators.spherical_sym module

This module implements differential operators on spherical grids.

SphericalLaplacian	Spherical Laplace using torch.
SphericalGradient	Spherical gradient operator using torch.
SphericalGradientSquared	Spherical gradient squared operator using torch.
SphericalDivergence	Spherical divergence operator using torch.
SphericalVectorGradient	Spherical vector gradient operator using torch.
SphericalTensorDivergence	Spherical tensor divergence operator using torch.

class SphericalDivergence(*grid*, *bcs*, *, *dtype*, *conservative=None*, *method='central'*)

Bases: *TorchDifferentialOperator*

Spherical divergence operator using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

class SphericalGradient(*grid*, *bcs*, *, *dtype*, *method='central'*)

Bases: *TorchDifferentialOperator*

Spherical gradient operator using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class SphericalGradientSquared(*grid*, *bcs*, *, *central=True*, *dtype*)

Bases: *TorchDifferentialOperator*

Spherical gradient squared operator using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate *r*. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical gradient squared operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **central** (*bool*) – Whether to use central differences. If *False*, forward and backward differences are used.
- **dtype** (*np.dtype*) – The data type of the field

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class SphericalLaplacian(*grid*, *bcs*, *, *dtype*, *conservative=None*)

Bases: *TorchDifferentialOperator*

Spherical Laplace using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical Laplacian operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **conservative** (*bool*) – Flag indicating whether the Laplace operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 0

The rank of the input tensor

Type
int

class SphericalTensorDivergence(*grid*, *bcs*, *, *dtype*, *conservative=False*)

Bases: *TorchDifferentialOperator*

Spherical tensor divergence operator using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical tensor divergence operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **conservative** (*bool*) – Flag indicating whether the operator should be conservative (which results in slightly slower computations). Conservative operators ensure mass conservation. If *None*, the value is read from the configuration option *operators.conservative_stencil*.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 2

The rank of the input tensor

Type
int

class SphericalVectorGradient(*grid*, *bcs*, *, *dtype*, *method='central'*)

Bases: *TorchDifferentialOperator*

Spherical vector gradient operator using torch.

The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate r. The radial discretization is defined as $r_i = r_{\min} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\min}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\max} = r_{\min} + N_r \Delta r$.

Initialize the Spherical vector gradient operator.

Parameters

- **grid** (*GridBase*) – The grid on which the operator acts
- **bcs** (*BoundariesList* or None) – The boundary conditions applied to the field. If *None*, no boundary conditions are enforced.
- **dtype** (*np.dtype*) – The data type of the field
- **method** (*str*) – The method for calculating the derivative. Possible values are 'central', 'forward', and 'backward'.

forward(*arr*, *args=None*)

Fill internal data array, apply operator, and return valid data.

Parameters
arr (*Tensor*)

Return type
Tensor

rank_in = 1

The rank of the input tensor

Type
int

pde.backends.torch.backend module

Defines base class of backends that implement computations.

class TorchBackend(*config=None*, *, *name=None*, *device='config'*)

Bases: *BackendBase*[Tensor]

Defines `torch` backend.

Initialize the torch backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend
- **device** (*str*) – The torch device to use. Special values are "config" (read from configuration) and "auto" (use CUDA if available, otherwise CPU)

compile_function(*func*, *, *to_device=False*, ***kwargs*)

General method that compiles a user function.

Parameters

- **func** (*callable*) – The function that needs to be compiled for this backend
- **to_device** (*bool*) – Moves (compiled) function to device
- ****kwargs** – Additional keyword arguments forwarded to `torch.compile()`

Return type
TFunc

compile_options = {'backend': 'inductor', 'dynamic': False, 'fullgraph': True, 'options': {'epilogue_fusion': True, 'max_autotune': True}}

defines options that affect compilation by torch

Type
dict

copy_data = True

Data must be copied from CPU numpy representation to a native device.

Type
bool

property device: device

The currently assigned torch device.

classmethod from_args(*config*, *args=''*, *, *name=None*)

Initialize backend with extra arguments.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **args** (*str*) – Additional arguments that determine how the backend is initialized
- **name** (*str*) – The name of the backend

Return type
Self

get_numpy_dtype(*dtype*)

Determine numpy dtype suitable for the torch backend.

Parameters
dtype (*DTypeLike*) – numpy dtype to convert to supported dtype

Returns
A numpy dtype that is compatible with the torch backend

Return type
torch.dtype

get_torch_dtype(*dtype*)

Convert dtype to torch dtype.

The torch dtype might be narrower than the corresponding numpy dtype if the configuration parameter *dtype_downcasting* is enabled.

Parameters
dtype (*DTypeLike*) – numpy dtype to convert to corresponding torch dtype

Returns
A proper dtype for torch

Return type
torch.dtype

implementation = 'torch'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

property info: dict[str, Any]

relevant information about the backend

Type
dict

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **rng** (Generator) – Random number generator (default: `default_rng()`) used to initialize the seed.

Return type
Callable[[], torch.Tensor]

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
Callable[[torch.Tensor, torch.Tensor, torch.Tensor | None], torch.Tensor]

make_integrator(*grid*, *, *dtype=<class 'numpy.float64'>*)

Return function that integrates discretized data over a grid.

Parameters

- **grid** (GridBase) – Grid for which the integrator is defined
- **dtype** (*DTypeLike*) – The data type of the field that is being integrated

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
Callable[[torch.Tensor], torch.Tensor]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a torch function applying an operator with boundary conditions.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*, optional) – The boundary conditions used before the operator is applied
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
TorchDifferentialOperator

Warning

The same operator should not be assigned to different variables that are used in the same code, because `torch` has problems compiling the resulting code. This particularly precludes caching the operators, since they then might be reused, e.g., if boundary conditions agree between different operators.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **grid** (GridBase)
- **operator** (*str | OperatorInfo*)
- **bcs** (BoundariesBase)
- **dtype** (*DTypeLike | None*)

make_operator_no_bc(*grid*, *operator*, *, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator without boundary conditions.

A function that takes the discretized full data as an input and an array of valid data points to which the result of applying the operator is written.

Note

The resulting function does not check whether the ghost cells of the input array have been supplied with sensible values. It is the responsibility of the user to set the values of the ghost cells beforehand. Use this function only if you absolutely know what you're doing. In all other cases, *make_operator()* is probably the better choice.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray), so they *out* array need to be supplied explicitly.

Return type
callable

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This supports typically only supports products between two vector fields.

Parameters
field (*DataFieldBase*) – Field for which the outer product is defined

Returns
function that takes two instance of ndarray, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
Callable[[torch.Tensor, torch.Tensor, torch.Tensor | None], torch.Tensor]

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (FieldBase) – An example for the state from which information can be extracted

Returns
Function returning deterministic part of the right hand side of the PDE.

Return type
TorchRHSType

make_stepper(*solver*, *state*)

Create a field-based stepping function for a given solver.

Parameters

- **solver** (*SolverBase*) – The solver instance, which determines how the stepper is constructed
- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type
StepperType

native_to_numpy(*value*)

Convert native values to numpy representation.

Parameters
value (*Any*)

Return type
Any

numpy_to_native(*value*)

Convert values from numpy to torch representation.

This method also ensures that the value is copied to the selected device.

Parameters
value (*Any*)

Return type
Tensor

pde.backends.torch.config module

Defines the configuration for the torch backend.

This configuration file will be imported without importing the enclosed module (which will instead be loaded on demand). Consequently, the module should use absolute references to other modules and not import anything from the backend.

pde.backends.torch.typing module

Defines types specific to the torch backend.

class TorchInnerStepperType(**args*, ***kwargs*)

Bases: `Protocol`

General backend-level stepping-function type working with torch tensors.

class TorchRHSType(**args*, ***kwargs*)

Bases: `Protocol`

General right-hand-side function type working with torch tensors.

pde.backends.torch.utils module

Defines utilities for the torch backend.

class TorchGaussianNoise(*data_shape*, ***, *dtype*, *generator=None*)

Bases: *TorchOperatorBase*

Operator that returns uncorrelated Gaussian random field.

Parameters

- **data_shape** (*tuple of ints*) – Shape of the output array
- **dtype** – Torch dtype of the returned data
- **generator** (`torch.Generator` or None) – Torch random number generator, which also allows setting the device on which the data is stored.

forward()

Define the computation performed at every call.

Should be overridden by all subclasses.

> **Note**
>
> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class TorchOperatorBase(***, *dtype*)

Bases: `Module`

Base class for operators implemented in torch.

Initialize the torch operator.

Parameters

dtype (*type[Any] | dtype[Any] | _HasDType[dtype[Any]] | _HasNumPyDType[dtype[Any]] | tuple[Any, Any] | list[Any] | _DTypeDict | str*) – The data type of the field using the numpy convention

register_array(*name*, *arr*)

Register an array as a buffer in the torch module.

Parameters

- **name** (*str*) – The name under which the buffer is registered
- **arr** (numpy.ndarray or torch.Tensor) – The array to register. If a numpy array is provided, it will be converted to a torch tensor with the appropriate dtype.

Return type

None

torch_heaviside(*x1*, *x2=None*)

Return the Heaviside step function using torch.

This does not use `torch.heaviside()` since this is not implemented for the MPS device.

Parameters

- **x1** (torch.Tensor) – Input values at which the Heaviside function is evaluated.
- **x2** (torch.Tensor, optional) – Value used where *x1 == 0*. If omitted, *0.5* is used.

Returns

Tensor containing the Heaviside values of *x1*.

Return type

torch.Tensor

torch_hypot(*x1*, *x2*)

Return the Euclidean norm `sqrt(x1**2 + x2**2)` using torch.

This wraps `torch.hypot()` and ensures that both inputs are converted to tensors before evaluation.

Parameters

- **x1** (torch.Tensor) – First input values.
- **x2** (torch.Tensor) – Second input values.

Returns

Tensor containing the element-wise hypotenuse of *x1* and *x2*.

Return type

torch.Tensor

4.1.7 pde.backends.base module

Defines base class of backends that implement computations.

class BackendBase(*config*, *, *name=None*)

Bases: Generic[TNativeArray]

Basic backend from which all other backends inherit.

The generic type parameter *TNativeArray* determines the type of the native data representation of the backend.

Initialize the backend.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **name** (*str*) – The name of the backend

compile_function(*func*, ***kwargs*)

General method that compiles a user function.

Parameters

- **func** (*callable*) – The function that needs to be compiled for this backend
- ****kwargs** – Additional arguments affecting how the function is compiled

Return type
TFunc

config: *Config*

Configuration options of this backend.

Type
dict

config_inheritance: list[str] = []

Additional backends that are queried for configuration parameters.

Type
list

copy_data: bool = False

Data must be copied from CPU numpy representation to a native device.

Type
bool

classmethod from_args(*config*, *args=''*, ***, *name=None*)

Initialize backend with extra arguments.

Parameters

- **config** (*Config*) – Configuration data for the backend
- **args** (*str*) – Additional arguments that determine how the backend is initialized
- **name** (*str*) – The name of the backend

Return type
Self

get_operator_info(*grid*, *operator*)

Return an operator for a particular grid.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.

Returns
information for the operator

Return type
OperatorInfo

get_registered_operators(*grid_id*)

Returns all operators defined for a grid.

Parameters
grid_id (GridBase or its type) – Grid or grid class for which the operators need to be returned

Return type
set[str]

implementation: str = 'undefined'

The name of the python module that is used to implement this backend. This information can be used to distinguish the general implementation of backends.

Type
str

property info: dict[str, Any]

relevant information about the backend

Type
dict

make_data_setter(*grid*, *rank*, *bcs=None*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (GridBase) – Grid for which the data setter is defined
- **rank** (*int*) – Rank of the data represented on the grid
- **bcs** (*BoundariesBase*, optional) – If supplied, the returned function also enforces boundary conditions by setting the ghost cells to the correct values

Returns
Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type
callable

make_expression_function(*expression*, *, *single_arg=False*, *user_funcs=None*)

Return a function evaluating an expression.

Parameters

- **expression** (ExpressionBase) – The expression that is converted to a function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

make_full_data_setter(*bcs*)

Create a function to set the valid part of a full data array.

Parameters

bcs (*BoundariesBase*, optional) – If supplied, the returned function also enforces boundary conditions by setting the ghost cells to the correct values

Returns

Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type

callable

make_gaussian_noise(*field*, *, *rng*)

Create a function generating Gaussian white noise.

Parameters

- **field** (*FieldBase*) – An example for the field from which the grid and other information can be extracted
- **rng** (`Generator`) – Random number generator (default: `default_rng()`).

Return type

Callable[[], TNativeArray]

make_ghost_cell_setter(*bcs*)

Return function that sets the ghost cells on a full array.

Parameters

bcs (*BoundariesBase*) – Defines the boundary conditions for a particular grid, for which the setter should be defined.

Returns

Callable with signature `(data_full: NumericArray, args=None)`, which sets the ghost cells of the full data, potentially using additional information in *args* (e.g., the time *t* during solving a PDE)

Return type

GhostCellSetter

make_inner_prod_operator(*field*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **field** (*DataFieldBase*) – Field for which the inner product is defined
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns

Function that takes two instance of native data arrays, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type

BinaryOperatorImplType

make_integrator(*grid*)

Return function that integrates discretized data over a grid.

Note that this function takes and returns data in the native representation of the backend. If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters

grid (GridBase) – Grid for which the operator is needed

Returns

A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type

Callable[[TNativeArray], TNativeArray]

make_interpolator(*field*, *, *fill=None*, *with_ghost_cells=False*)

Returns a function that can be used to interpolate values.

Parameters

- **field** (*DataFieldBase*) – Field for which the interpolator is defined
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.

Returns

A function which returns interpolated values when called with arbitrary positions within the space of the grid.

Return type

Callable[[FloatingArray, NumericArray], NumberOrArray]

make_mpi_synchronizer(*operator='MAX'*)

Return function that synchronizes values between multiple MPI processes.

Warning

The default implementation does not synchronize anything. This is simply a hook, which can be used by backends that support MPI

Parameters

operator (*str or int*) – Flag determining how the value from multiple nodes is combined. Possible values include “MAX”, “MIN”, and “SUM”.

Returns

Function that can be used to synchronize values across nodes

Return type

Callable[[float], float]

make_operator(*grid*, *operator*, *, *bcs*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed
- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **bcs** (*BoundariesBase*) – The boundary conditions used before the operator is applied
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type

OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time.

Returns

the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type

callable

Parameters

- **grid** (GridBase)
- **operator** (*str | OperatorInfo*)
- **bcs** (BoundariesBase)
- **dtype** (*DTypeLike | None*)

make_operator_no_bc(*grid*, *operator*, *, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator without boundary conditions.

A function that takes the discretized full data as an input and an array of valid data points to which the result of applying the operator is written.

Note

The resulting function does not check whether the ghost cells of the input array have been supplied with sensible values. It is the responsibility of the user to set the values of the ghost cells beforehand. Use this function only if you absolutely know what you're doing. In all other cases, *make_operator()* is probably the better choice.

Parameters

- **grid** (GridBase) – Grid for which the operator is needed

- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray), so they *out* array need to be supplied explicitly.

Return type
callable

make_outer_prod_operator(*field*)

Return operator calculating the outer product between two fields.

This supports typically only supports products between two vector fields.

Parameters
field (*DataFieldBase*) – Field for which the outer product is defined

Returns
Function that takes two instance of native data arrays, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_pde_rhs(*eq*, *state*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **eq** (*PDEBase*) – The object describing the differential equation
- **state** (FieldBase) – An example for the state from which information can be extracted

Returns
Function returning deterministic part of the right hand side of the PDE

Return type
Callable[[TNativeArray, float], TNativeArray]

make_stepper(*solver*, *state*)

Create a field-based stepping function for a given solver.

Parameters

- **solver** (*SolverBase*) – The solver instance, which determines how the stepper is constructed
- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*.

Return type
StepperType

make_valid_data_setter(*grid*, *rank*)

Create a function to set the valid part of a full data array.

Parameters

- **grid** (GridBase) – Grid for which the data setter is defined
- **rank** (*int*) – Rank of the data represented on the grid

Returns

Takes two numpy arrays, setting the valid data in the first one, using the second array. The arrays need to be allocated already and they need to have the correct dimensions, which are not checked. If *bcs* are given, a third argument is allowed, which sets arguments for the BCs.

Return type

callable

native_to_numpy(*value: TNativeArray*) → ndarray[Any, dtype[number]]

native_to_numpy(*value: TValue*) → TValue

Convert native values to numpy representation.

Parameters

value (*Any*) – The value to convert to numpy representation

Return type

Any

numpy_to_native(*value: ndarray[Any, dtype[number]]*) → TNativeArray

numpy_to_native(*value: TValue*) → TValue

Convert values from numpy to native representation.

Parameters

value (*Any*) – The value to convert from numpy representation

Return type

Any

classmethod register_operator(*grid_cls*, *name*, *factory_func=None*, ***, *rank_in=0*, *rank_out=0*)

Register an operator for a particular grid.

Example

The method can either be used directly:

```
BackendClass.register_operator(grid_class, "operator", make_operator)
```

or as a decorator for the factory function:

```
@BackendClass.register_operator(grid_class, "operator")
def make_operator(grid: GridBase): ...
```

Parameters

- **grid_cls** (GridBase) – Grid class for which the operator is defined
- **name** (*str*) – The name of the operator to register
- **factory_func** (*callable*) – A function with signature `(grid: GridBase, **kwargs)`, which takes a grid object and optional keyword arguments and returns an implementation of the given operator. This implementation is a function that takes a `ndarray`

of discretized values as arguments and returns the resulting discretized data in a ndarray after applying the operator.

- **rank_in** (*int*) – The rank of the input field for the operator
- **rank_out** (*int*) – The rank of the field that is returned by the operator

supports_mpi: bool = False

This backend supports parallel simulations with MPI.

Type
bool

4.1.8 pde.backends.registry module

Defines the registry for managing backends.

BackendRegistry	Class handling all backends and their configurations.
load_default_config	Load a default configuration from a module.
get_backend	Return backend specified by string of instance.
registered_backends	Returns all registered backends.

class BackendRegistry

Bases: object

Class handling all backends and their configurations.

Backends can exist in three different states in registry: * Registered meta-information on how to load a backend package * Loaded backend module, so the class is available * Fully instantiated *BackendBase* classes

get_backend(*name*, *, *config=None*, ***kwargs*)

Return backend object, potentially loading the respective package.

The returned backend is cached if *config* is not specified. Consequently, the same object will be returned for repeated calls to *get_backend*, which allows sharing configuration parameters. Moreover, if *name* only specifies a backend, i.e., does not contain a colon :, the configuration of this backend is linked with the global configuration `pde.config`, such that changes to the config are reflected globally.

Parameters

- **name** (*str*) – Name of the backend to be loaded.
- **config** (*dict*) – Additional configuration options for this specific backend. The full configuration will be taken from the global configuration and merged with the given options here.
- ****kwargs** – Additional options of the backend

Returns
An instance of the backend with the particular configuration

Return type
BackendBase

register_backend(*backend*)

Register a loaded backend object.

Parameters
backend (*BackendBase*) – Implementation of the backend

Return type
None

register_class(*name*, *cls*)

Register a backend class.

Parameters

- **name** (*str*) – Name of the backend
- **cls** (subclass of `BackendBase`) – The class for creating a backend

register_package(*name*, *package_path*, *, *config=None*)

Register a backend python package (without loading it yet)

Parameters

- **name** (*str*) – Name of the backend
- **package_path** (*str*) – Import path for the package
- **config** (*list*) – Configuration options for the package

Return type
None

values()

Iterate over all backends that can be imported.

Return type
Iterator[*BackendBase*]

get_backend(*backend*, *config=None*)

Return backend specified by string of instance.

The returned backend is cached if *config* is not specified. Consequently, the same object will be returned for repeated calls to *get_backend*, which allows sharing configuration parameters. Moreover, if *name* only specifies a backend, i.e., does not contain a colon *:*, the configuration of this backend is linked with the global configuration `pde.config`, such that changes to the config are reflected globally.

Parameters

- **backend** (str or *BackendBase*) – Backend specified by name given as a string. If the string contains a colon, the first part determines the backend, whereas the second part can be used to convey additional information. For example, `torch:cuda` may load a torch backend and use a cuda device. As a special case, we also allow full backend objects, which are simply returned. This is a simple way to allow providing full backend objects in places where we otherwise would expect a backend name.
- **config** (*dict*) – Additional configuration options for this specific backend. The full configuration will be taken from the global configuration and merged with the given options here. This option is only permitted if *backend* is a string since there otherwise might be unintended side effects of modifying an existing backend.

Returns
An initialized backend

Return type
BackendBase

load_default_config(*module_path*)

Load a default configuration from a module.

Parameters
module_path (*str*) – String to the module to be loaded

Return type
list[*Parameter*] | None

registered_backends()
Returns all registered backends.

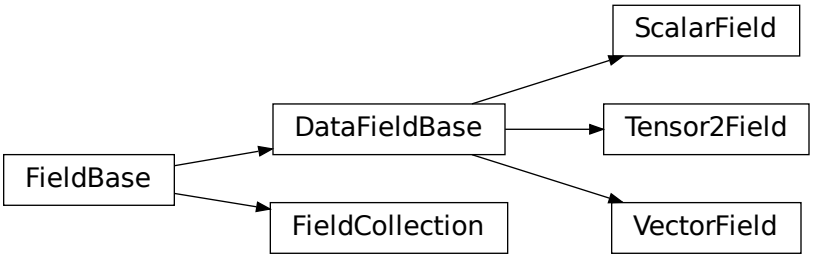
Return type
list[str]

4.2 pde.fields package

Package defining fields, which contain the actual data stored on discrete grids.

ScalarField	Scalar field discretized on a grid.
VectorField	Vector field discretized on a grid.
Tensor2Field	Tensor field of rank 2 discretized on a grid.
FieldCollection	Collection of fields defined on the same grid.

Inheritance structure of the classes:



The details of the classes are explained below:

class FieldCollection(*fields*, *, *copy_fields=False*, *label=None*, *labels=None*, *dtype=None*)
Bases: *FieldBase*

Collection of fields defined on the same grid.

Note

All fields in a collection must have the same data type. This might lead to up-casting, where for instance a combination of a real-valued and a complex-valued field will be both stored as complex fields.

Parameters

- **fields** (sequence or mapping of DataFieldBase) – Sequence or mapping of the individual fields. If a mapping is used, the keys set the names of the individual fields.

- **copy_fields** (*bool*) – Flag determining whether the individual fields given in *fields* are copied. Note that fields are always copied if some of the supplied fields are identical. If fields are copied the original fields will be left untouched. Conversely, if *copy_fields* == *False*, the original fields are modified so their data points to the collection. It is thus basically impossible to have fields that are linked to multiple collections at the same time.
- **label** (*str*) – Label of the field collection
- **labels** (*list of str*) – Labels of the individual fields. If omitted, the labels from the *fields* argument are used.
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

append(**fields*, *label=None*)

Create new collection with appended field(s)

Parameters

- ***fields** (*FieldCollection* or *DataFieldBase*) – A sequence of single fields or collection of fields that will be appended to the fields in the current collection. The data of all fields will be copied.
- **label** (*str*) – Label of the new field collection. If omitted, the current label is used

Returns

A new field collection, which combines the current one with fields given by *fields*.

Return type

FieldCollection

assert_field_compatible(*other*, *accept_scalar=False*)

Checks whether *other* is compatible with the current field.

Parameters

- **other** (FieldBase) – Other field this is compared to
- **accept_scalar** (*bool, optional*) – Determines whether it is acceptable that *other* is an instance of *ScalarField*.

property attributes: dict[str, Any]

describes the state of the instance (without the data)

Type

dict

property attributes_serialized: dict[str, str]

serialized version of the attributes

Type

dict

property averages: list

Averages of all fields.

copy(*, *label=None*, *dtype=None*)

Return a copy of the data, but not of the grid.

Parameters

- **label** (*str, optional*) – Name of the returned field

- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **self** (FieldCollection)

Return type
FieldCollection

property fields: list[*DataFieldBase*]

the fields of this collection

Type
list

classmethod from_data(*field_classes*, *grid*, *data*, *, *with_ghost_cells=True*, *label=None*, *labels=None*, *dtype=None*)

Create a field collection from classes and data.

Parameters

- **field_classes** (*list*) – List of the classes that define the individual fields
- **data** (ndarray, optional) – Data values of all fields at support points of the grid
- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data
- **label** (*str*) – Label of the field collection
- **labels** (*list of str*) – Labels of the individual fields. If omitted, the labels from the *fields* argument are used.
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

Return type
FieldCollection

classmethod from_scalar_expressions(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *labels=None*, *dtype=None*)

Create a field collection on a grid from given expressions.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expressions** (*list of str*) – A list of mathematical expression, one for each field in the collection. The expressions determine the values as a function of the position on the grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or ndarray.

- **label** (*str, optional*) – Name of the whole collection
- **labels** (*list of str, optional*) – Names of the individual fields
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

Return type
FieldCollection

classmethod from_state(*attributes*, *data=None*)

Create a field collection from given state.

Parameters

- **attributes** (*dict*) – The attributes that describe the current instance
- **data** (ndarray, optional) – Data values at support points of the grid defining all fields

Return type
FieldCollection

get_image_data(*index=0*, ***kwargs*)

Return data for plotting an image of the field.

Parameters

- **index** (*int*) – Index of the field whose data is returned
- ****kwargs** – Arguments forwarded to the *get_image_data* method

Returns
Information useful for plotting an image of the field

Return type
dict

get_line_data(*index=0*, *scalar='auto'*, *extract='auto'*)

Return data for a line plot of the field.

Parameters

- **index** (*int*) – Index of the field whose data is returned
- **scalar** (*str or int*) – The method for extracting scalars as described in `DataFieldBase.to_scalar()`.
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.

Returns
Information useful for performing a line plot of the field

Return type
dict

property integrals: list

Integrals of all fields.

interpolate_to_grid(*grid*, ***, *fill=None*, *label=None*)

Interpolate the data of this field collection to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.

- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field collection

Returns
Interpolated data

Return type
FieldCollection

property labels: _FieldLabels

the labels of all fields.

> **Note**
>
> The attribute returns a special class `_FieldLabels` to allow specific manipulations of the field labels. The returned object behaves much like a list, but assigning values will modify the labels of the fields in the collection.

Type
`_FieldLabels`

property magnitudes: NumericArray

scalar magnitudes of all fields.

Type
ndarray

plot(*kind='auto'*, *figsize='auto'*, *arrangement='horizontal'*, *subplot_args=None*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the fields in the collection.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image*, *line*, *vector*, *interactive*, or *merged*. Alternatively, *auto* determines the best visualization based on each field itself. Instead of a single value for all fields, a list with individual values can be given, unless *merged* is chosen.
- **figsize** (*str or tuple of numbers*) – Determines the figure size. The figure size is unchanged if the string *default* is passed. Conversely, the size is adjusted automatically when *auto* is passed. Finally, a specific figure size can be specified using two values, using `matplotlib.figure.Figure.set_size_inches()`.
- **arrangement** (*str or tuple of int*) – Determines how the sub panels will be arranged. The default value *horizontal* places all subplots next to each other, whereas *vertical* puts them below each other. Alternatively, an exact number of rows and columns can be specified by the tuple `(nrows, ncols)`. Negative values will be replaced by suitable values that ensure enough panels.
- **subplot_args** (*list*) – Additional arguments for the specific subplots. Should be a list with a dictionary of arguments for each subplot. Supplying an empty dict allows to keep the default setting of specific subplots.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.

- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **fig** (`matplotlib.figures.Figure`) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns
Instances that contain information to update all the plots with new data later.

Return type
List of `PlotReference`

project(*axes*, *, *label=None*, ***kwargs*)

Project fields along given axes.

This is currently only implemented for scalar fields. If any field in the collection has higher rank, the entire process fails.

Parameters

- **axes** (*list of str*) – The names of the axes that are removed by the projection operation. The valid names for a given grid are the ones in the `GridBase.axes` attribute.
- **label** (*str, optional*) – Name of the returned collection. If omitted, the current label is used.
- ****kwargs** – Additional arguments forwarded to the projection method. In particular, this allows selecting a projection method.

Returns
The projected data of all fields on a subgrid of the original grid.

Return type
FieldCollection

classmethod scalar_random_uniform(*num_fields*, *grid*, *vmin=0*, *vmax=1*, *, *label=None*, *labels=None*, *rng=None*)

Create scalar fields with random values between *vmin* and *vmax*

Parameters

- **num_fields** (*int*) – The number of fields to create
- **grid** (*GridBase*) – Grid defining the space on which the fields are defined
- **vmin** (*float*) – Lower bound. Can be complex to create complex fields

- **vmax** (*float*) – Upper bound. Can be complex to create complex fields
- **label** (*str, optional*) – Name of the field collection
- **labels** (*list of str, optional*) – Names of the individual fields
- **rng** (Generator) – Random number generator (default: default_rng())

Return type
FieldCollection

slice(*position*, *, *label=None*, ***kwargs*)

Slice all fields at a given position.

This is currently only implemented for scalar fields. If any field in the collection has higher rank, the entire process fails.

Parameters

- **position** (*dict*) – Determines the location of the slice using a dictionary supplying coordinate values for a subset of axes. Axes not mentioned in the dictionary are retained and form the slice. For instance, in a 2d Cartesian grid, *position = {'x': 1}* slices along the y-direction at x=1. Additionally, the special positions 'low', 'mid', and 'high' are supported to reference relative positions along the axis.
- **label** (*str, optional*) – Name of the returned collection. If omitted, the current label is used.
- ****kwargs** – Additional arguments forwarded to the slicing method (e.g., method).

Returns
The projected data of all fields on a subgrid of the original grid.

Return type
FieldCollection

smooth(*sigma=1*, *, *out=None*, *label=None*)

Applies Gaussian smoothing with the given standard deviation.

This function respects periodic boundary conditions of the underlying grid, using reflection when no periodicity is specified.

Parameters

- **sigma** (*float*) – Gives the standard deviation of the smoothing in real length units (default: 1)
- **out** (FieldCollection, *optional*) – Optional field into which the smoothed data is stored
- **label** (*str, optional*) – Name of the returned field

Returns
Smoothed data, stored at *out* if given.

Return type
FieldCollection

classmethod unserialize_attributes(*attributes*)

Unserializes the given attributes.

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

class ScalarField(*grid*, *data='zeros'*, ***, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Scalar field discretized on a grid.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

classmethod from_expression(*grid*, *expression*, ***, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a scalar field on a grid from a given expression.

Warning

This implementation uses exec() and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expression** (*str*) – Mathematical expression for the scalar value as a function of the position on the grid. The expression may contain standard mathematical functions and it may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or ndarray.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type
ScalarField

classmethod from_image(*path*, *bounds=None*, *periodic=False*, *, *label=None*)

Create a scalar field from an image.

Parameters

- **path** (Path or str) – The path to the image file
- **bounds** (*tuple, optional*) – Gives the coordinate range for each axis. This should be two tuples of two numbers each, which mark the lower and upper bound for each axis.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. This is either a list of booleans defining periodicity for each individual axis or a single boolean value specifying the same periodicity for all axes.
- **label** (*str, optional*) – Name of the field

Return type

ScalarField

get_boundary_field(*index*, *bc=None*, *, *label=None*)

Get the field on the specified boundary.

Parameters

- **index** (*str or tuple*) – Index specifying the boundary. Can be either a string given in *boundary_names*, like "left", or a tuple of the axis index perpendicular to the boundary and a boolean specifying whether the boundary is at the upper side of the axis or not, e.g., (1, True).
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **label** (*str*) – Label of the returned field

Returns

The field on the boundary

Return type

ScalarField

gradient(*bc*, *out=None*, ***kwargs*)

Apply gradient operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions

enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **out** (pde.fields.vectorial.VectorField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional keyword arguments (e.g., label)

Returns
result of applying the operator

Return type
VectorField

gradient_squared(*bc*, *out=None*, ***kwargs*)

Apply squared gradient operator and return result as a field.

This evaluates $|\nabla\phi|^2$ for the scalar field ϕ

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.scalar.ScalarField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Extra arguments are forwarded to *apply_operator()*

Returns
the squared gradient of the field

Return type
ScalarField

interpolate_to_grid(*grid*, *, *bc=None*, *fill=None*, *label=None*)

Interpolate the data of this scalar field to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like

x- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field
- **self** (ScalarField)

Returns
Field of the same rank as the current one.

Return type
ScalarField

laplace(*bc*, *out=None*, ***kwargs*)

Apply Laplace operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.scalar.ScalarField, *optional*) – Optional scalar field to which the result is written.
- ****kwargs** – Additional keyword arguments (e.g., label)

Returns
the Laplacian of the field

Return type
ScalarField

project(*axes*, *, *method='integral'*, *label=None*)

Project scalar field along given axes.

Parameters

- **axes** (*list of str*) – The names of the axes that are removed by the projection operation. The valid names for a given grid are the ones in the `GridBase.axes` attribute.
- **method** (*str*) – The projection method. Supported values are:
  - `"integral"`: Integrate over the removed axes.

- `"average"` or `"mean"`: Return the average over the removed axes.
- `"maximum"` or `"max"`: Return the maximum over the removed axes.
- `"minimum"` or `"min"`: Return the minimum over the removed axes.

- **label** (*str, optional*) – The label of the returned field

Returns
The projected data is a scalar field defined on a subgrid of the original grid.

Return type
ScalarField

rank = 0

slice(*position*, *, *method='nearest'*, *label=None*)
Slice data at a given position.

> **Note**
> This method should not be used to evaluate fields right at the boundary since it does not respect boundary conditions. Use *get_boundary_field()* to obtain the values directly on the boundary.

Parameters

- **position** (*dict*) – Determines the location of the slice using a dictionary supplying coordinate values for a subset of axes. Axes not mentioned in the dictionary are retained and form the slice. For instance, in a 2d Cartesian grid, *position = {'x': 1}* slices along the y-direction at x=1. Additionally, the special positions 'low', 'mid', and 'high' are supported to reference relative positions along the axis.
- **method** (*str*) – The method used for slicing. Currently, we only support *nearest*, which takes data from cells defined on the grid.
- **label** (*str, optional*) – The label of the returned field

Returns
The sliced data is a scalar field defined on a subgrid of the original grid.

Return type
ScalarField

to_scalar(*scalar='auto'*, *, *label=None*)
Return a modified scalar field by applying method *scalar*

Parameters

- **scalar** (*str or callable*) – Determines the method used for obtaining the scalar. If this is a callable, it is simply applied to self.data and a new scalar field with this data is returned. Alternatively, pre-defined methods can be selected using strings. Here, *abs* and *norm* denote the norm of each entry of the field, while *norm_squared* returns the squared norm. The default *auto* is to return a (unchanged) copy of a real field and the norm of a complex field.
- **label** (*str, optional*) – Name of the returned field

Returns
Scalar field after applying the operation

Return type
ScalarField

class Tensor2Field(*grid*, *data='zeros'*, ***, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Tensor field of rank 2 discretized on a grid.

> **Warning**
>
> Components of the tensor field are given in the local basis. While the local basis is identical to the global basis in Cartesian coordinates, the local basis depends on position in curvilinear coordinate systems. Moreover, the field always contains all components, even if the underlying grid assumes symmetries.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

convert(*form*, *inplace=False*, ***, *label=None*)

Convert tensor to a specific form in each point in space.

Parameters

- **form** (*str*) – Determines the form (*symmetric*, *anti-symmetric*, *transposed*, or *traceless*) that the converted tensors should have.
- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns

converted tensor field

Return type

Tensor2Field

divergence(*bc*, *out=None*, ***kwargs*)

Apply tensor divergence and return result as a field.

The tensor divergence is a vector field v_α resulting from a contracting of the derivative of the tensor field $t_{\alpha\beta}$:

$$v_\alpha = \sum_\beta \frac{\partial t_{\alpha\beta}}{\partial x_\beta}$$

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for

the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **out** (pde.fields.vectorial.VectorField, *optional*) – Optional scalar field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
result of applying the operator

Return type
VectorField

dot(*other:* VectorField, *out:* VectorField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *VectorField*

dot(*other:* Tensor2Field, *out:* Tensor2Field | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *Tensor2Field*

Calculate the dot product involving a tensor field.

This supports the dot product between two tensor fields as well as the product between a tensor and a vector. The resulting fields will be a tensor or vector, respectively.

Parameters

- **other** (pde.fields.vectorial.VectorField *or* pde.fields.tensorial.Tensor2Field) – the second field
- **out** (pde.fields.vectorial.VectorField *or* pde.fields.tensorial.Tensor2Field, *optional*) – Optional field to which the result is written.
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand
- **label** (*str*, *optional*) – Name of the returned field

Returns
VectorField or *Tensor2Field*: result of applying dot operator

Return type
VectorField | Tensor2Field

classmethod from_expression(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a tensor field on a grid from given expressions.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined

- **expressions** (*list of str*) – A 2d list of mathematical expression, one for each component of the tensor field. The expressions determine the values as a function of the position on the grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type
Tensor2Field

is_symmetric(*rtol=1e-05*, *atol=1e-08*)

Returns whether the tensor is symmetric.

Parameters

- **rtol** (*float*) – The relative tolerance parameter (see `allclose()`).
- **atol** (*float*) – The absolute tolerance parameter (see `allclose()`).

Return type
bool

plot_components(*kind='auto'*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the components of this tensor field.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image* or *line*. Alternatively, *auto* determines the best visualization based on the grid.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.

- **fig** (matplotlib.figures.Figure) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns
Instances that contain information to update all the plots with new data later.

Return type
2d list of PlotReference

rank = 2

symmetrize(*make_traceless=False*, *inplace=False*, *, *label=None*)

Symmetrize the tensor field.

Parameters

- **make_traceless** (*bool*) – Determines whether the result is also traceless
- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns
result of the operation

Return type
Tensor2Field

to_scalar(*scalar='auto'*, *, *label='scalar `{scalar}`'*)

Return scalar variant of the field.

The invariants of the tensor field $\boldsymbol{A}$ are

$$I_1 = \mathrm{tr}(\boldsymbol{A})$$
$$I_2 = \frac{1}{2}\left[(\mathrm{tr}(\boldsymbol{A})^2 - \mathrm{tr}(\boldsymbol{A}^2)\right]$$
$$I_3 = \det(A)$$

where *tr* denotes the trace and *det* denotes the determinant. Note that the three invariants can only be distinct and non-zero in three dimensions. In two dimensional spaces, we have the identity $2I_2 = I_3$ and in one-dimensional spaces, we have $I_1 = I_3$ as well as $I_2 = 0$.

Parameters

- **scalar** (*str*) – The method to calculate the scalar. Possible choices include *norm* (the default chosen when the value is *auto*), *min*, *max*, *squared_sum*, *norm_squared*, *trace* (or *invariant1*), *invariant2*, and *determinant* (or *invariant3*)
- **label** (*str, optional*) – Name of the returned field

Returns
the scalar field after applying the operation

Return type
ScalarField

trace(*, *label='trace'*)

Return the trace of the tensor field as a scalar field.

Parameters
label (*str, optional*) – Name of the returned field

Returns
scalar field of traces

Return type
ScalarField

transpose(*inplace=False*, *, *label='transpose'*)

Return the transpose of the tensor field.

Parameters

- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns
transpose of the tensor field

Return type
Tensor2Field

class VectorField(*grid*, *data='zeros'*, *, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Vector field discretized on a grid.

Warning

Components of the vector field are given in the local basis. While the local basis is identical to the global basis in Cartesian coordinates, the local basis depends on position in curvilinear coordinate systems. Moreover, the field always contains all components, even if the underlying grid assumes symmetries.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

divergence(*bc*, *out=None*, ***kwargs*)

Apply divergence operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions

enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **out** (`pde.fields.scalar.ScalarField, optional`) – Optional scalar field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
Divergence of the field

Return type
ScalarField

dot(*other:* VectorField, *out:* ScalarField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *ScalarField*

dot(*other:* Tensor2Field, *out:* VectorField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *VectorField*

Calculate the dot product involving a vector field.

This supports the dot product between two vectors fields as well as the product between a vector and a tensor. The resulting fields will be a scalar or vector, respectively.

Parameters

- **other** (`VectorField` or `Tensor2Field`) – the second field
- **out** (`pde.fields.scalar.ScalarField` *or* `pde.fields.vectorial.VectorField, optional`) – Optional field to which the result is written.
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand
- **label** (*str, optional*) – Name of the returned field

Returns
ScalarField or *VectorField*: result of applying the operator

Return type
ScalarField | *VectorField*

classmethod from_expression(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a vector field on a grid from given expressions.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expressions** (*list of str*) – A list of mathematical expression, one for each component of the vector field. The expressions determine the values as a function of the position on the

grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.

- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or ndarray.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type
VectorField

classmethod from_scalars(*fields*, *, *label=None*, *dtype=None*)

Create a vector field from a list of ScalarFields.

Note that the data of the scalar fields is copied in the process

Parameters

- **fields** (*list*) – The list of (compatible) scalar fields
- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Returns
the resulting vector field

Return type
pde.fields.vectorial.VectorField

get_vector_data(*transpose=False*, *max_points=None*, ***kwargs*)

Return data for a vector plot of the field.

Parameters

- **transpose** (*bool*) – Determines whether the transpose of the data should be plotted.
- **max_points** (*int*) – The maximal number of points that is used along each axis. This option can be used to sub-sample the data.
- ****kwargs** – Additional parameters forwarded to *grid.get_image_data*

Returns
Information useful for plotting an vector field

Return type
dict

gradient(*bc*, *out=None*, ***kwargs*)

Apply vector gradient operator and return result as a field.

The vector gradient field is a tensor field $t_{\alpha\beta}$ that specifies the derivatives of the vector field v_α with respect to all coordinates x_β.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions need to determine all components of the vector field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (*pde.fields.vectorial.VectorField, optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
Gradient of the field

Return type
Tensor2Field

interpolate_to_grid(*grid*, *, *bc=None*, *fill=None*, *label=None*)

Interpolate the data of this vector field to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field
- **self** (*VectorField*)

Returns
Field of the same rank as the current one.

Return type
VectorField

laplace(*bc*, *out=None*, ***kwargs*)

Apply vector Laplace operator and return result as a field.

The vector Laplacian is a vector field L_α containing the second derivatives of the vector field v_α with respect to the coordinates x_β:

$$L_\alpha = \sum_\beta \frac{\partial^2 v_\alpha}{\partial x_\beta \partial x_\beta}$$

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.vectorial.VectorField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns

Laplacian of the field

Return type

VectorField

make_outer_prod_operator(*backend='numba'*)

Return operator calculating the outer product of two vector fields.

> **Warning**
>
> This function does not check types or dimensions.

Parameters

backend (*str*) – The backend (e.g., 'numba' or 'numba_mpi') used for this operator.

Returns

function that takes two instance of ndarray, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type

BinaryOperatorImplType

outer_product(*other*, *out=None*, ***, *label=None*)

Calculate the outer product of this vector field with another.

Parameters

- **other** (*VectorField*) – The second vector field

- **out** (*Tensor2Field*, optional) – Optional tensorial field to which the result is written.
- **label** (*str, optional*) – Name of the returned field

Returns
result of the operation

Return type
Tensor2Field

plot_components(*kind='auto'*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the components of this vector field.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image* or *line*. Alternatively, *auto* determines the best visualization based on the grid.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **fig** (`matplotlib.figures.Figure`) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns
Instances that contain information to update all the plots with new data later.

Return type
list of `PlotReference`

rank = 1

to_scalar(*scalar='auto'*, ***, *label='scalar `{scalar}`'*)

Return scalar variant of the field.

Parameters

- **scalar** (*str*) – Choose the method to use. Possible choices are *norm*, *max*, *min*, *squared_sum*, *norm_squared*, or an integer specifying which component is returned (indexing starts at *0*). The default value *auto* picks the method automatically: The first (and only)

component is returned for real fields on one-dimensional spaces, while the norm of the vector is returned otherwise.

- **label** (*str, optional*) – Name of the returned field

Returns
The scalar field after applying the operation

Return type
pde.fields.scalar.ScalarField

4.2.1 pde.fields.base module

Defines base class of fields or collections, which are discretized on grids.

FieldBase	Abstract base class for describing (discretized) fields.
RankError	Error indicating that the field has the wrong rank.

class FieldBase(*grid*, *data*, *, *label=None*)

Bases: object

Abstract base class for describing (discretized) fields.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **data** (ndarray, optional) – Field values at the support points of the grid and the ghost cells
- **label** (*str, optional*) – Name of the field

apply(*func*, *out=None*, *, *label=None*, *evaluate_args=None*)

Applies a function/expression to the data and returns it as a field.

Parameters

- **func** (*callable or str*) – (Vectorized) function being applied to the data or an expression that can be parsed using sympy (*evaluate()* is used in this case). The local field values can be accessed using the field labels for a field collection and via the variable *c* otherwise.
- **out** (*FieldBase, optional*) – Optional field into which the data is written
- **label** (*str, optional*) – Name of the returned field
- **evaluate_args** (*dict*) – Additional arguments passed to *evaluate()*. Only used when *func* is a string.

Returns
Field with new data. Identical to *out* if given.

Return type
FieldBase

assert_field_compatible(*other*, *accept_scalar=False*)

Checks whether *other* is compatible with the current field.

Parameters

- **other** (*FieldBase*) – The other field this one is compared to
- **accept_scalar** (*bool, optional*) – Determines whether it is acceptable that *other* is an instance of *ScalarField*.

Return type
None

property attributes: dict[str, Any]

describes the state of the instance (without the data)

Type
dict

property attributes_serialized: dict[str, str]

serialized version of the attributes

Type
dict

conjugate()

Returns complex conjugate of the field.

Returns
the complex conjugated field

Return type
FieldBase

abstractmethod copy(*, *label=None*, *dtype=None*)

Return a new field with the data (but not the grid) copied.

Parameters

- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically or the dtype of the current field is used.
- **self** (*TField*)

Returns
A copy of the current field

Return type
DataFieldBase

property data: NumericArray

discretized data at the support points.

Type
ndarray

data_shape: tuple[int, ...]

the shape of the data at each grid point

Type
tuple

property dtype: DTypeLike

the numpy dtype of the underlying data.

Type
DTypeLike

classmethod from_file(*filename*)

Create field from data stored in a file.

Field can be written to a file using *FieldBase.to_file()*.

Example

Write a field to a file and then read it back:

```
field = pde.ScalarField(...)
field.write_to("test.hdf5")

field_copy = pde.FieldBase.from_file("test.hdf5")
```

Parameters

filename (*str*) – Path to the file being read

Returns

The field with the appropriate sub-class

Return type

FieldBase

classmethod from_state(*attributes*, *data=None*)

Create a field from given state.

Parameters

- **attributes** (*dict*) – The attributes that describe the current instance
- **data** (ndarray, optional) – Data values at the support points of the grid defining the field

Returns

The field created from the state

Return type

FieldBase

abstractmethod get_image_data()

Return data for plotting an image of the field.

Returns

Information useful for plotting an image of the field

Return type

dict

abstractmethod get_line_data(*scalar='auto'*, *extract='auto'*)

Return data for a line plot of the field.

Parameters

- **scalar** (*str or int*) – The method for extracting scalars as described in DataFieldBase.to_scalar().
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.

Returns

Information useful for performing a line plot of the field

Return type
dict

property grid: *GridBase*

The grid on which the field is defined.

Type
base,GridBase

property imag: Self

Imaginary part of the field.

Type
FieldBase

property is_complex: bool

whether the field contains real or complex data

Type
bool

property label: str | None

the name of the field

Type
str

abstractmethod plot(**args*, ***kwargs*)

Visualize the field.

plot_interactive(*viewer_args=None*, ***kwargs*)

Create an interactive plot of the field using napari

For a detailed description of the launched program, see the napari webpage.

Parameters

- **viewer_args** (*dict*) – Arguments passed to napari.viewer.Viewer to affect the viewer.
- ****kwargs** – Extra arguments passed to the plotting function

Return type
None

property real: Self

Real part of the field.

Type
FieldBase

split_mpi(*decomposition='auto'*)

Splits the field onto subgrids in an MPI run.

In a normal serial simulation, the method simply returns the field itself. In contrast, in an MPI simulation, the field provided on the main node is split onto all nodes using the given decomposition. The field data provided on all other nodes is not used.

Parameters
decomposition (*list of ints*) – Number of subdivision in each direction. Should be a list of length *grid.num_axes* specifying the number of nodes for this axis. If one value is *-1*, its

value will be determined from the number of available nodes. The default value *auto* tries to determine an optimal decomposition by minimizing communication between nodes.

Returns
The part of the field that corresponds to the subgrid associated with the current MPI node.

Return type
FieldBase

to_file(*filename*, ***kwargs*)

Store field in a file.

The extension of the filename determines what format is being used. If it ends in *.h5* or *.hdf*, the Hierarchical Data Format is used. The other supported format are images, where only the most typical formats are supported.

To load the field back from the file, you may use *FieldBase.from_file()*.

Example

Write a field to a file and then read it back:

```
field = pde.ScalarField(...)
field.write_to("test.hdf5")

field_copy = pde.FieldBase.from_file("test.hdf5")
```

Parameters

- **filename** (*str*) – Path where the data is stored
- ****kwargs** – Additional parameters may be supported for some formats

Return type
None

classmethod unserialize_attributes(*attributes*)

Unserializes the given attributes.

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

property writeable: bool

whether the field data can be changed or not

Type
bool

exception RankError

Bases: TypeError

Error indicating that the field has the wrong rank.

4.2.2 pde.fields.collection module

Defines a collection of fields to represent multiple fields defined on a common grid.

class FieldCollection(*fields*, *, *copy_fields=False*, *label=None*, *labels=None*, *dtype=None*)

Bases: *FieldBase*

Collection of fields defined on the same grid.

> **Note**
>
> All fields in a collection must have the same data type. This might lead to up-casting, where for instance a combination of a real-valued and a complex-valued field will be both stored as complex fields.

Parameters

- **fields** (sequence or mapping of DataFieldBase) – Sequence or mapping of the individual fields. If a mapping is used, the keys set the names of the individual fields.
- **copy_fields** (*bool*) – Flag determining whether the individual fields given in *fields* are copied. Note that fields are always copied if some of the supplied fields are identical. If fields are copied the original fields will be left untouched. Conversely, if *copy_fields* == *False*, the original fields are modified so their data points to the collection. It is thus basically impossible to have fields that are linked to multiple collections at the same time.
- **label** (*str*) – Label of the field collection
- **labels** (*list of str*) – Labels of the individual fields. If omitted, the labels from the *fields* argument are used.
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

append(**fields*, *label=None*)

Create new collection with appended field(s)

Parameters

- ***fields** (*FieldCollection* or *DataFieldBase*) – A sequence of single fields or collection of fields that will be appended to the fields in the current collection. The data of all fields will be copied.
- **label** (*str*) – Label of the new field collection. If omitted, the current label is used

Returns

A new field collection, which combines the current one with fields given by *fields*.

Return type

FieldCollection

assert_field_compatible(*other*, *accept_scalar=False*)

Checks whether *other* is compatible with the current field.

Parameters

- **other** (FieldBase) – Other field this is compared to
- **accept_scalar** (*bool, optional*) – Determines whether it is acceptable that *other* is an instance of *ScalarField*.

property attributes: dict[str, Any]

describes the state of the instance (without the data)

Type
dict

property attributes_serialized: dict[str, str]

serialized version of the attributes

Type
dict

property averages: list

Averages of all fields.

copy(*, *label=None*, *dtype=None*)

Return a copy of the data, but not of the grid.

Parameters

- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **self** (FieldCollection)

Return type
FieldCollection

property fields: list[DataFieldBase]

the fields of this collection

Type
list

classmethod from_data(*field_classes*, *grid*, *data*, *, *with_ghost_cells=True*, *label=None*, *labels=None*, *dtype=None*)

Create a field collection from classes and data.

Parameters

- **field_classes** (*list*) – List of the classes that define the individual fields
- **data** (ndarray, optional) – Data values of all fields at support points of the grid
- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data
- **label** (*str*) – Label of the field collection
- **labels** (*list of str*) – Labels of the individual fields. If omitted, the labels from the *fields* argument are used.
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

Return type
FieldCollection

classmethod from_scalar_expressions(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *labels=None*, *dtype=None*)

Create a field collection on a grid from given expressions.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expressions** (*list of str*) – A list of mathematical expression, one for each field in the collection. The expressions determine the values as a function of the position on the grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **label** (*str, optional*) – Name of the whole collection
- **labels** (*list of str, optional*) – Names of the individual fields
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

Return type
FieldCollection

classmethod from_state(*attributes*, *data=None*)

Create a field collection from given state.

Parameters

- **attributes** (*dict*) – The attributes that describe the current instance
- **data** (ndarray, optional) – Data values at support points of the grid defining all fields

Return type
FieldCollection

get_image_data(*index=0*, ***kwargs*)

Return data for plotting an image of the field.

Parameters

- **index** (*int*) – Index of the field whose data is returned
- ****kwargs** – Arguments forwarded to the *get_image_data* method

Returns
Information useful for plotting an image of the field

Return type
dict

get_line_data(*index=0*, *scalar='auto'*, *extract='auto'*)

Return data for a line plot of the field.

Parameters

- **index** (*int*) – Index of the field whose data is returned

- **scalar** (*str or int*) – The method for extracting scalars as described in `DataFieldBase.to_scalar()`.
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.

Returns
Information useful for performing a line plot of the field

Return type
dict

property integrals: list

Integrals of all fields.

interpolate_to_grid(*grid*, *, *fill=None*, *label=None*)

Interpolate the data of this field collection to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field collection

Returns
Interpolated data

Return type
FieldCollection

property labels: _FieldLabels

the labels of all fields.

> **Note**
>
> The attribute returns a special class `_FieldLabels` to allow specific manipulations of the field labels. The returned object behaves much like a list, but assigning values will modify the labels of the fields in the collection.

Type
`_FieldLabels`

property magnitudes: NumericArray

scalar magnitudes of all fields.

Type
ndarray

plot(*kind='auto'*, *figsize='auto'*, *arrangement='horizontal'*, *subplot_args=None*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the fields in the collection.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image*, *line*, *vector*, *interactive*, or *merged*. Alternatively, *auto* determines the best visualization based on each field itself. Instead of a single value for all fields, a list with individual values can be given, unless *merged* is chosen.
- **figsize** (*str or tuple of numbers*) – Determines the figure size. The figure size is unchanged if the string *default* is passed. Conversely, the size is adjusted automatically when *auto* is passed. Finally, a specific figure size can be specified using two values, using `matplotlib.figure.Figure.set_size_inches()`.
- **arrangement** (*str or tuple of int*) – Determines how the sub panels will be arranged. The default value *horizontal* places all subplots next to each other, whereas *vertical* puts them below each other. Alternatively, an exact number of rows and columns can be specified by the tuple `(nrows, ncols)`. Negative values will be replaced by suitable values that ensure enough panels.
- **subplot_args** (*list*) – Additional arguments for the specific subplots. Should be a list with a dictionary of arguments for each subplot. Supplying an empty dict allows to keep the default setting of specific subplots.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **fig** (`matplotlib.figures.Figure`) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns

Instances that contain information to update all the plots with new data later.

Return type

List of `PlotReference`

project(*axes*, *, *label=None*, ***kwargs*)

Project fields along given axes.

This is currently only implemented for scalar fields. If any field in the collection has higher rank, the entire process fails.

Parameters

- **axes** (*list of str*) – The names of the axes that are removed by the projection operation. The valid names for a given grid are the ones in the `GridBase.axes` attribute.

- **label** (*str, optional*) – Name of the returned collection. If omitted, the current label is used.
- ****kwargs** – Additional arguments forwarded to the projection method. In particular, this allows selecting a projection method.

Returns
The projected data of all fields on a subgrid of the original grid.

Return type
FieldCollection

classmethod scalar_random_uniform(*num_fields*, *grid*, *vmin=0*, *vmax=1*, *, *label=None*, *labels=None*, *rng=None*)

Create scalar fields with random values between *vmin* and *vmax*

Parameters

- **num_fields** (*int*) – The number of fields to create
- **grid** (*GridBase*) – Grid defining the space on which the fields are defined
- **vmin** (*float*) – Lower bound. Can be complex to create complex fields
- **vmax** (*float*) – Upper bound. Can be complex to create complex fields
- **label** (*str, optional*) – Name of the field collection
- **labels** (*list of str, optional*) – Names of the individual fields
- **rng** (*Generator*) – Random number generator (default: `default_rng()`)

Return type
FieldCollection

slice(*position*, *, *label=None*, ***kwargs*)

Slice all fields at a given position.

This is currently only implemented for scalar fields. If any field in the collection has higher rank, the entire process fails.

Parameters

- **position** (*dict*) – Determines the location of the slice using a dictionary supplying coordinate values for a subset of axes. Axes not mentioned in the dictionary are retained and form the slice. For instance, in a 2d Cartesian grid, *position = {'x': 1}* slices along the y-direction at x=1. Additionally, the special positions 'low', 'mid', and 'high' are supported to reference relative positions along the axis.
- **label** (*str, optional*) – Name of the returned collection. If omitted, the current label is used.
- ****kwargs** – Additional arguments forwarded to the slicing method (e.g., method).

Returns
The projected data of all fields on a subgrid of the original grid.

Return type
FieldCollection

smooth(*sigma=1*, *, *out=None*, *label=None*)

Applies Gaussian smoothing with the given standard deviation.

This function respects periodic boundary conditions of the underlying grid, using reflection when no periodicity is specified.

Parameters

- **sigma** (*float*) – Gives the standard deviation of the smoothing in real length units (default: 1)
- **out** (*FieldCollection, optional*) – Optional field into which the smoothed data is stored
- **label** (*str, optional*) – Name of the returned field

Returns
Smoothed data, stored at *out* if given.

Return type
FieldCollection

classmethod unserialize_attributes(*attributes*)

Unserializes the given attributes.

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

4.2.3 pde.fields.datafield_base module

Defines base class of single fields with arbitrary rank.

class DataFieldBase(*grid*, *data='zeros'*, ***, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *FieldBase*

Abstract base class for describing fields of single entities.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

apply_operator(*operator*, *bc*, *out=None*, ***, *label=None*, *backend='default'*, *args=None*, ***kwargs*)

Apply a (differential) operator and return result as a field.

Parameters

- **operator** (*str*) – An identifier determining the operator. Note that not all grids support the same operators.

- **bc** (*BoundariesData | None*) – Boundary conditions applied to the field before applying the operator. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (*DataFieldBase*, optional) – Optional field to which the result is written.
- **label** (*str, optional*) – Name of the returned field
- **backend** (*str*) – Backend (e.g., *numba* or *numpy*) for constructing the operator
- **args** (*dict*) – Additional arguments for the boundary conditions
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
Field data after applying the operator. This field is identical to *out* if this argument was specified.

Return type
DataFieldBase

property average: NumberOrArray

the average of data.

This is calculated by integrating each component of the field over space and dividing by the grid volume

Type
Float or ndarray

copy(*, *label=None*, *dtype=None*)

Return a copy of the field.

This method creates a new instance of the field with the same grid and data. Optionally, a new label and/or data type can be specified for the copy.

Parameters

- **label** (*str, optional*) – Name of the copied field. If omitted, the label of the original field is used.
- **dtype** (*numpy dtype, optional*) – Data type of the copied field. If omitted, the data type of the original field is used.

Returns
A copy of the current field instance.

Return type
DataFieldBase

property data_shape: tuple[int, ...]

the shape of the data at each grid point

Type
tuple

property fluctuations: NumberOrArray

quantification of the average fluctuations.

The fluctuations are defined as the standard deviation of the data scaled by the cell volume. This definition makes the fluctuations independent of the discretization. It corresponds to the physical scaling available in the *random_normal()*.

Returns

A tensor with the same rank of the field, specifying the fluctuations of each component of the tensor field individually. Consequently, a simple scalar is returned for a *ScalarField*.

Return type

ndarray

Type

Float or ndarray

classmethod from_state(*attributes*, *data=None*)

Create a field from given state.

Parameters

- **attributes** (*dict*) – The attributes that describe the current instance
- **data** (ndarray, optional) – Data values at the support points of the grid defining the field

Returns

The instance created from the stored state

Return type

DataFieldBase

get_boundary_values(*axis*, *upper*, *bc*)

Get the field values directly on the specified boundary.

Parameters

- **axis** (*int*) – The axis perpendicular to the boundary
- **upper** (*bool*) – Whether the boundary is at the upper side of the axis
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. If this is *None*, it is assumed that boundary conditions have been set beforehand, e.g., using *set_ghost_cells()*. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

Returns

The discretized values on the boundary

Return type

ndarray

classmethod get_class_by_rank(*rank*)

Return a *DataFieldBase* subclass describing a field with a given rank.

Parameters
rank (*int*) – The rank of the tensor field

Returns
The DataField class that corresponds to the rank

Return type
type[DataFieldBase]

get_image_data(*scalar='auto'*, *transpose=False*, ***kwargs*)

Return data for plotting an image of the field.

Returns
Information useful for plotting an image of the field

Return type
dict

Parameters

- **scalar** (*str*)
- **transpose** (*bool*)

get_line_data(*scalar='auto'*, *extract='auto'*)

Return data for a line plot of the field.

Parameters

- **scalar** (*str or int*) – The method for extracting scalars as described in *DataFieldBase.to_scalar()*.
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.

Returns
Information useful for performing a line plot of the field

Return type
dict

get_vector_data(*transpose=False*, ***kwargs*)

Return data for a vector plot of the field.

Parameters

- **transpose** (*bool*) – Determines whether the transpose of the data should be plotted.
- ****kwargs** – Additional parameters are forwarded to *grid.get_image_data*

Returns
Information useful for plotting an vector field

Return type
dict

insert(*point*, *amount*)

Adds an (integrated) value to the field at an interpolated position.

Parameters

- **point** (ndarray) – The point inside the grid where the value is added. This is given in grid coordinates.
- **amount** (Number or ndarray) – The amount that will be added to the field. The value describes an integrated quantity (given by the field value times the discretization volume). This is important for consistency with different discretizations and in particular grids with non-uniform discretizations.

Return type
None

property integral: NumericArray

integral of each component over space.

Type
ndarray

interpolate(*point*, *, *bc=None*, *fill=None*)

Interpolate the field to points between support points.

Parameters

- **point** (ndarray) – The points at which the values should be obtained. This is given in grid coordinates.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.

Returns
the values of the field

Return type
ndarray

interpolate_to_grid(*grid*, *, *bc=None*, *fill=None*, *label=None*)

Interpolate the data of this field to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one

condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field

Returns
Field of the same rank as the current one.

Return type
Self

property magnitude: float

determine the (scalar) magnitude of the field

This is calculated by getting a scalar field using the default arguments of the *to_scalar()* method, averaging the result over the whole grid, and taking the absolute value.

Type
float

make_dot_operator(*backend='default'*, *, *conjugate=True*)

Return operator calculating the dot product between two fields.

This supports both products between two vectors as well as products between a vector and a tensor.

Parameters

- **backend** (*str*) – Backend (e.g., *numba* or *numpy*) for constructing the operator
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand

Returns
function that takes two instance of ndarray, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type
BinaryOperatorImplType

make_interpolator(*, *fill=None*, *with_ghost_cells=False*, *backend='default'*)

Returns a function that can be used to interpolate values.

Parameters

- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **with_ghost_cells** (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the ghost points. If this is the case, the boundaries are not checked and the coordinates are used as is.

- **backend** (*str*) – The name of the backend to use to implement this operator.

Returns
A function which returns interpolated values when called with arbitrary positions within the space of the grid.

Return type
Callable[[FloatingArray, NumericArray], NumberOrArray]

plot(*kind='auto'*, **args*, *title=None*, *filename=None*, *action='auto'*, *ax_style=None*, *fig_style=None*, *ax=None*, ***kwargs*)

Visualize the field.

Parameters

- **kind** (*str*) – Determines the visualizations. Supported values are *image*, *line*, *vector*, or *interactive*. Alternatively, *auto* determines the best visualization based on the field itself.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically.
- **filename** (*str, optional*) – If given, the plot is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *auto* or *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given.
- **ax_style** (*dict*) – Dictionary with properties that will be changed on the axis after the plot has been drawn by calling `matplotlib.pyplot.setp()`. A special item i this dictionary is *use_offset*, which is flag that can be used to control whether offset are shown along the axes of the plot.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **ax** (`matplotlib.axes.Axes`) – Figure axes to be used for plotting. The special value "create" creates a new figure, while "reuse" attempts to reuse an existing figure, which is the default.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function determined by *kind*.

Returns
Instance that contains information to update the plot with new data later.

Return type
PlotReference

Tip

Typical additional arguments for the various plot kinds include

- `kind == "line"`:
  - *scalar*: Sets method for extracting scalars as described in *DataFieldBase.to_scalar()*.
  - *extract*: Method used for extracting the line data.
  - *ylabel*: Label of the y-axis.
  - *ylim*: Data limits of the y-axis.

  - Additional arguments are passed to `matplotlib.pyplot.plot()`

- `kind == "image"`:
  - *colorbar*: Determines whether a colorbar is shown
  - ***scalar*: Sets method for extracting scalars as described in** *DataFieldBase.to_scalar()*.
  - *transpose* Determines whether the transpose of the data is plotted
  - Most remaining arguments are passed to `matplotlib.pyplot.imshow()`
- `kind == `"vector"`:
  - *method* Can be either *quiver* or *streamplot*
  - *transpose* Determines whether the transpose of the data is plotted
  - *max_points* Sets max. number of points along each axis in quiver plots
  - Additional arguments are passed to `matplotlib.pyplot.quiver()` or `matplotlib.pyplot.streamplot()`.

classmethod random_colored(*grid*, *exponent=0*, *scale=1*, ***, *label=None*, *dtype=None*, *rng=None*)

Create a field of random values with colored noise.

The spatially correlated values obey

$$\langle c_i(\boldsymbol{k}) c_j(\boldsymbol{k'}) \rangle = \Gamma^2 |\boldsymbol{k}|^\nu \delta_{ij} \delta(\boldsymbol{k} - \boldsymbol{k'})$$

in spectral space, where $\boldsymbol{k}$ is the wave vector. The special case $\nu = 0$ corresponds to white noise. Note that the spatial correlations always assume periodic boundary conditions (even if the underlying grid does not) and that the components of tensor fields are uncorrelated.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **exponent** (*float*) – Exponent ν of the power spectrum
- **scale** (*float*) – Scaling factor Γ determining noise strength
- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double*.
- **rng** (*Generator*) – Random number generator (default: `default_rng()`)

Return type

Self

classmethod random_harmonic(*grid*, *modes=3*, *harmonic=<ufunc 'cos'>*, *axis_combination=<ufunc 'multiply'>*, ***, *label=None*, *dtype=None*, *rng=None*)

Create a random field build from harmonics.

The resulting fields will be highly correlated in space and can thus serve for testing differential operators.

With the default settings, the resulting field $c_i(\mathbf{x})$ is given by

$$c_i(\mathbf{x}) = \prod_{\alpha=1}^{N} \sum_{j=1}^{M} a_{ij\alpha} \cos\left(\frac{2\pi x_\alpha}{j L_\alpha}\right) ,$$

where N is the number of spatial dimensions, each with length L_α, M is the number of modes given by *modes*, and $a_{ij\alpha}$ are random amplitudes, chosen from a uniform distribution over the interval [0, 1].

Note that the product could be replaced by a sum when *axis_combination* = *numpy.add* and the cos() could be any other function given by the parameter *harmonic*.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **modes** (*int*) – Number M of harmonic modes
- **harmonic** (*callable*) – Determines which harmonic function is used. Typical values are `numpy.sin()` and `numpy.cos()`, which basically relate to different boundary conditions applied at the grid boundaries.
- **axis_combination** (*callable*) – Determines how values from different axis are combined. Typical choices are `numpy.multiply()` and `numpy.add()` resulting in products and sums of the values along axes, respectively.
- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double*.
- **rng** (*Generator*) – Random number generator (default: `default_rng()`)

Return type
Self

classmethod random_normal(*grid*, *mean=0*, *std=1*, ***, *scaling='none'*, *correlation='none'*, *label=None*, *dtype=None*, *rng=None*, ***kwargs*)

Creates Gaussian random field with normal distributed random values.

A complex field is returned when either *mean* or *std* is a complex number. In this case, the real and imaginary parts of these arguments are used to determine the distribution of the real and imaginary parts of the resulting field. Consequently, `pde.fields.scalar.ScalarField.random_normal(grid, 0, 1 + 1j)` creates a complex field where the real and imaginary parts are chosen from a standard normal distribution.

Real and imaginary parts of fields, as well as all components of vector and tensor fields, are always uncorrelated. Correlations in spatial positions are supported through the *correlation* argument. If set, the returned field f obeys the selected correlation function $C(k)$ (see table below for details). In Fourier space, we thus have

$$\langle f(\boldsymbol{k}) f(\boldsymbol{k}') \rangle = C(|\boldsymbol{k}|)\delta(\boldsymbol{k} - \boldsymbol{k}')$$

For simplicity, the correlations respect periodic boundary conditions.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **mean** (*float*) – Mean of the Gaussian distribution
- **std** (*float*) – Standard deviation of the Gaussian distribution
- **scaling** (*str*) – Determines how the values are scaled. Possible choices are 'none' (values are drawn from a normal distribution with given mean and standard deviation) or 'physical' (the variance of the random number is scaled by the inverse volume of the grid cell; this is for instance useful for concentration fields, which vary less in larger cells).
- **correlation** (*str*) – Selects the autocorrelation function to create spatially correlated noise. Many of the options (described below) support additional parameters that can be supplied as keyword arguments.

- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *float* if both *mean* and *std* are real, otherwise it is *complex*. Note that if a complex dtype is supplied, the mean and std are also assumed to be complex, so that specifying them as real will lead to a vanishing imaginary part of the returned field.
- **rng** (Generator) – Random number generator (default: `default_rng()`)
- ****kwargs** – Additional parameters can affect details of the correlation function.

Return type
Self

Table 35: Supported correlation functions

Identifier	Correlation function
`none`	No correlation, $C(k) = \delta(k)$
`gaussian`	$C(k) = \exp(\frac{1}{2}k^2\lambda^2)$ with the length scale λ set by argument `length_scale`
`power law`	$C(k) = k^{\nu/2}$ with exponent ν set by argument `exponent`.
`cosine`	$C(k) = \exp\left(-s^2(\lambda k - 1)^2\right)$ with the length scale λ set by argument `length_scale`, whereas the sharpness parameter s is set by `sharpness` and defaults to 10.

Note

The returned field only has the correct standard deviation (set by *std*) for correlation functions that decrease monotonously. In other cases (i.e., for `cosine` correlation), the variance depends on details, like the resolution of the grid.

classmethod random_uniform(*grid*, *vmin=0*, *vmax=1*, ***, *label=None*, *dtype=None*, *rng=None*)

Create field with uncorrelated uniform distributed random values.

A complex field is returned when *vmin* or *vmax* is a complex number. In this case, the real and imaginary parts of these arguments are used to determine the distribution of the real and imaginary parts of the resulting field. Consequently, `pde.fields.scalar.ScalarField.random_uniform(grid, 0, 1 + 1j)` creates a complex field where the real and imaginary parts are chosen from a standard uniform distribution.

Real and imaginary parts of fields, all components of vector and tensor fields, as well as all spatial positions are always uncorrelated.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **vmin** (*float*) – Lower bound for the random values
- **vmax** (*float*) – Upper bound for the random values
- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double* if both *vmin* and *vmax* are real, otherwise it is *complex*.
- **rng** (Generator) – Random number generator (default: `default_rng()`)

Return type
Self

rank: int

set_ghost_cells(*bc*, *, *args=None*, ***kwargs*)

Set the boundary values on virtual points for all boundaries.

Parameters

- **bc** (*str or list or tuple or dict*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **args** – Additional arguments that might be supported by special boundary conditions.
- ****kwargs** – Some boundary conditions might provide additional control via keyword arguments, e.g., *set_corners* to control whether corner cells are set using interpolation.

Return type
None

smooth(*sigma=1*, *, *out=None*, *label=None*)

Applies Gaussian smoothing with the given standard deviation.

This function respects periodic boundary conditions of the underlying grid, using reflection when no periodicity is specified.

Parameters

- **sigma** (*float*) – Gives the standard deviation of the smoothing in real length units (default: 1)
- **out** (*FieldBase, optional*) – Optional field into which the smoothed data is stored. Setting this to the input field enables in-place smoothing.
- **label** (*str, optional*) – Name of the returned field

Returns
Field with smoothed data. This is stored at *out* if given.

Return type
Self

abstractmethod to_scalar(*scalar='auto'*, *, *label=None*)

Return scalar variant of the field.

Parameters

- **scalar** (*str*)
- **label** (*str | None*)

Return type
ScalarField

classmethod unserialize_attributes(*attributes*)

Unserializes the given attributes.

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

4.2.4 pde.fields.scalar module

Defines a scalar field over a grid.

class ScalarField(*grid*, *data='zeros'*, ***, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Scalar field discretized on a grid.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

classmethod from_expression(*grid*, *expression*, ***, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a scalar field on a grid from a given expression.

> **Warning**
>
> This implementation uses exec() and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expression** (*str*) – Mathematical expression for the scalar value as a function of the position on the grid. The expression may contain standard mathematical functions and it may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or ndarray.
- **label** (*str, optional*) – Name of the field

- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type
ScalarField

classmethod from_image(*path*, *bounds=None*, *periodic=False*, *, *label=None*)

Create a scalar field from an image.

Parameters

- **path** (Path or str) – The path to the image file
- **bounds** (*tuple, optional*) – Gives the coordinate range for each axis. This should be two tuples of two numbers each, which mark the lower and upper bound for each axis.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. This is either a list of booleans defining periodicity for each individual axis or a single boolean value specifying the same periodicity for all axes.
- **label** (*str, optional*) – Name of the field

Return type
ScalarField

get_boundary_field(*index*, *bc=None*, *, *label=None*)

Get the field on the specified boundary.

Parameters

- **index** (*str or tuple*) – Index specifying the boundary. Can be either a string given in *boundary_names*, like `"left"`, or a tuple of the axis index perpendicular to the boundary and a boolean specifying whether the boundary is at the upper side of the axis or not, e.g., `(1, True)`.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **label** (*str*) – Label of the returned field

Returns
The field on the boundary

Return type
ScalarField

gradient(*bc*, *out=None*, ***kwargs*)

Apply gradient operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.vectorial.VectorField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional keyword arguments (e.g., label)

Returns
result of applying the operator

Return type
VectorField

gradient_squared(*bc*, *out=None*, ***kwargs*)

Apply squared gradient operator and return result as a field.

This evaluates $|\nabla\phi|^2$ for the scalar field ϕ

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.scalar.ScalarField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Extra arguments are forwarded to *apply_operator()*

Returns
the squared gradient of the field

Return type
ScalarField

interpolate_to_grid(*grid*, *, *bc=None*, *fill=None*, *label=None*)

Interpolate the data of this scalar field to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field
- **self** (ScalarField)

Returns
Field of the same rank as the current one.

Return type
ScalarField

laplace(*bc*, *out=None*, ***kwargs*)

Apply Laplace operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (*pde.fields.scalar.ScalarField, optional*) – Optional scalar field to which the result is written.
- ****kwargs** – Additional keyword arguments (e.g., label)

Returns
the Laplacian of the field

Return type
ScalarField

project(*axes*, *, *method='integral'*, *label=None*)

Project scalar field along given axes.

Parameters

- **axes** (*list of str*) – The names of the axes that are removed by the projection operation. The valid names for a given grid are the ones in the `GridBase.axes` attribute.
- **method** (*str*) – The projection method. Supported values are:
  - `"integral"`: Integrate over the removed axes.
  - `"average"` or `"mean"`: Return the average over the removed axes.
  - `"maximum"` or `"max"`: Return the maximum over the removed axes.
  - `"minimum"` or `"min"`: Return the minimum over the removed axes.
- **label** (*str, optional*) – The label of the returned field

Returns

The projected data is a scalar field defined on a subgrid of the original grid.

Return type

ScalarField

rank = 0

slice(*position*, *, *method='nearest'*, *label=None*)

Slice data at a given position.

> **Note**
>
> This method should not be used to evaluate fields right at the boundary since it does not respect boundary conditions. Use *get_boundary_field()* to obtain the values directly on the boundary.

Parameters

- **position** (*dict*) – Determines the location of the slice using a dictionary supplying coordinate values for a subset of axes. Axes not mentioned in the dictionary are retained and form the slice. For instance, in a 2d Cartesian grid, *position = {'x': 1}* slices along the y-direction at x=1. Additionally, the special positions 'low', 'mid', and 'high' are supported to reference relative positions along the axis.
- **method** (*str*) – The method used for slicing. Currently, we only support *nearest*, which takes data from cells defined on the grid.
- **label** (*str, optional*) – The label of the returned field

Returns

The sliced data is a scalar field defined on a subgrid of the original grid.

Return type

ScalarField

to_scalar(*scalar='auto'*, *, *label=None*)

Return a modified scalar field by applying method *scalar*

Parameters

- **scalar** (*str or callable*) – Determines the method used for obtaining the scalar. If this is a callable, it is simply applied to self.data and a new scalar field with this data is returned. Alternatively, pre-defined methods can be selected using strings. Here, *abs* and *norm* denote

the norm of each entry of the field, while *norm_squared* returns the squared norm. The default *auto* is to return a (unchanged) copy of a real field and the norm of a complex field.

- **label** (*str, optional*) – Name of the returned field

Returns
Scalar field after applying the operation

Return type
ScalarField

4.2.5 pde.fields.tensorial module

Defines a tensorial field of rank 2 over a grid.

class Tensor2Field(*grid*, *data='zeros'*, ***, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Tensor field of rank 2 discretized on a grid.

Warning

Components of the tensor field are given in the local basis. While the local basis is identical to the global basis in Cartesian coordinates, the local basis depends on position in curvilinear coordinate systems. Moreover, the field always contains all components, even if the underlying grid assumes symmetries.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or *ndarray*, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values. Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

convert(*form*, *inplace=False*, ***, *label=None*)

Convert tensor to a specific form in each point in space.

Parameters

- **form** (*str*) – Determines the form (*symmetric*, *anti-symmetric*, *transposed*, or *traceless*) that the converted tensors should have.
- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns
converted tensor field

Return type
Tensor2Field

divergence(*bc*, *out=None*, ***kwargs*)

Apply tensor divergence and return result as a field.

The tensor divergence is a vector field v_α resulting from a contracting of the derivative of the tensor field $t_{\alpha\beta}$:

$$v_\alpha = \sum_\beta \frac{\partial t_{\alpha\beta}}{\partial x_\beta}$$

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (*pde.fields.vectorial.VectorField, optional*) – Optional scalar field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns

result of applying the operator

Return type

VectorField

dot(*other:* VectorField, *out:* VectorField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *VectorField*

dot(*other:* Tensor2Field, *out:* Tensor2Field | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *Tensor2Field*

Calculate the dot product involving a tensor field.

This supports the dot product between two tensor fields as well as the product between a tensor and a vector. The resulting fields will be a tensor or vector, respectively.

Parameters

- **other** (*pde.fields.vectorial.VectorField or pde.fields.tensorial.Tensor2Field*) – the second field
- **out** (*pde.fields.vectorial.VectorField or pde.fields.tensorial.Tensor2Field, optional*) – Optional field to which the result is written.
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand
- **label** (*str, optional*) – Name of the returned field

Returns

VectorField or *Tensor2Field*: result of applying dot operator

Return type

VectorField | Tensor2Field

classmethod from_expression(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a tensor field on a grid from given expressions.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expressions** (*list of str*) – A 2d list of mathematical expression, one for each component of the tensor field. The expressions determine the values as a function of the position on the grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type

Tensor2Field

is_symmetric(*rtol=1e-05*, *atol=1e-08*)

Returns whether the tensor is symmetric.

Parameters

- **rtol** (*float*) – The relative tolerance parameter (see `allclose()`).
- **atol** (*float*) – The absolute tolerance parameter (see `allclose()`).

Return type

bool

plot_components(*kind='auto'*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the components of this tensor field.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image* or *line*. Alternatively, *auto* determines the best visualization based on the grid.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements.

Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.

- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **fig** (`matplotlib.figures.Figure`) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns
Instances that contain information to update all the plots with new data later.

Return type
2d list of `PlotReference`

rank = 2

symmetrize(*make_traceless=False*, *inplace=False*, ***, *label=None*)

Symmetrize the tensor field.

Parameters

- **make_traceless** (*bool*) – Determines whether the result is also traceless
- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns
result of the operation

Return type
Tensor2Field

to_scalar(*scalar='auto'*, ***, *label='scalar `{scalar}`'*)

Return scalar variant of the field.

The invariants of the tensor field $\boldsymbol{A}$ are

$$
\begin{aligned}
I_1 &= \mathrm{tr}(\boldsymbol{A}) \\
I_2 &= \frac{1}{2}\left[(\mathrm{tr}(\boldsymbol{A})^2 - \mathrm{tr}(\boldsymbol{A}^2)\right] \\
I_3 &= \det(A)
\end{aligned}
$$

where *tr* denotes the trace and *det* denotes the determinant. Note that the three invariants can only be distinct and non-zero in three dimensions. In two dimensional spaces, we have the identity $2I_2 = I_3$ and in one-dimensional spaces, we have $I_1 = I_3$ as well as $I_2 = 0$.

Parameters

- **scalar** (*str*) – The method to calculate the scalar. Possible choices include *norm* (the default chosen when the value is *auto*), *min*, *max*, *squared_sum*, *norm_squared*, *trace* (or *invariant1*), *invariant2*, and *determinant* (or *invariant3*)
- **label** (*str, optional*) – Name of the returned field

Returns
the scalar field after applying the operation

Return type
ScalarField

trace(*, *label='trace'*)

Return the trace of the tensor field as a scalar field.

Parameters
label (*str, optional*) – Name of the returned field

Returns
scalar field of traces

Return type
ScalarField

transpose(*inplace=False*, *, *label='transpose'*)

Return the transpose of the tensor field.

Parameters

- **inplace** (*bool*) – Overwrites current field if *True*
- **label** (*str, optional*) – Name of the returned field

Returns
transpose of the tensor field

Return type
Tensor2Field

4.2.6 pde.fields.vectorial module

Defines a vectorial field over a grid.

class VectorField(*grid*, *data='zeros'*, *, *label=None*, *dtype=None*, *with_ghost_cells=False*)

Bases: *DataFieldBase*

Vector field discretized on a grid.

Warning

Components of the vector field are given in the local basis. While the local basis is identical to the global basis in Cartesian coordinates, the local basis depends on position in curvilinear coordinate systems. Moreover, the field always contains all components, even if the underlying grid assumes symmetries.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined.
- **data** (Number or ndarray, optional) – Field values at the support points of the grid. The flag *with_ghost_cells* determines whether this data array contains values for the ghost cells, too. The resulting field will contain real data unless the *data* argument contains complex values.

Special values are "zeros" or None, initializing the field with zeros, and "empty", just allocating memory with unspecified values.

- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.
- **with_ghost_cells** (*bool*) – Indicates whether the ghost cells are included in data

divergence(*bc*, *out=None*, ***kwargs*)

Apply divergence operator and return result as a field.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.scalar.ScalarField, *optional*) – Optional scalar field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns

Divergence of the field

Return type

ScalarField

dot(*other:* VectorField, *out:* ScalarField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *ScalarField*

dot(*other:* Tensor2Field, *out:* VectorField | *None* = *None*, *, *conjugate: bool* = *True*, *label: str* = *'dot product'*) → *VectorField*

Calculate the dot product involving a vector field.

This supports the dot product between two vectors fields as well as the product between a vector and a tensor. The resulting fields will be a scalar or vector, respectively.

Parameters

- **other** (VectorField or Tensor2Field) – the second field
- **out** (pde.fields.scalar.ScalarField *or* pde.fields.vectorial.VectorField, *optional*) – Optional field to which the result is written.
- **conjugate** (*bool*) – Whether to use the complex conjugate for the second operand
- **label** (*str, optional*) – Name of the returned field

Returns

ScalarField or *VectorField*: result of applying the operator

Return type
ScalarField | *VectorField*

classmethod from_expression(*grid*, *expressions*, *, *user_funcs=None*, *consts=None*, *label=None*, *dtype=None*)

Create a vector field on a grid from given expressions.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **expressions** (*list of str*) – A list of mathematical expression, one for each component of the vector field. The expressions determine the values as a function of the position on the grid. The expressions may contain standard mathematical functions and they may depend on the axes labels of the grid. More information can be found in the *expression documentation*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Return type
VectorField

classmethod from_scalars(*fields*, *, *label=None*, *dtype=None*)

Create a vector field from a list of ScalarFields.

Note that the data of the scalar fields is copied in the process

Parameters

- **fields** (*list*) – The list of (compatible) scalar fields
- **label** (*str, optional*) – Name of the returned field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it will be determined from *data* automatically.

Returns
the resulting vector field

Return type
pde.fields.vectorial.VectorField

get_vector_data(*transpose=False*, *max_points=None*, ***kwargs*)

Return data for a vector plot of the field.

Parameters

- **transpose** (*bool*) – Determines whether the transpose of the data should be plotted.

- **max_points** (*int*) – The maximal number of points that is used along each axis. This option can be used to sub-sample the data.
- ****kwargs** – Additional parameters forwarded to *grid.get_image_data*

Returns
Information useful for plotting an vector field

Return type
dict

gradient(*bc*, *out=None*, ***kwargs*)

Apply vector gradient operator and return result as a field.

The vector gradient field is a tensor field $t_{\alpha\beta}$ that specifies the derivatives of the vector field v_α with respect to all coordinates x_β.

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions need to determine all components of the vector field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (*pde.fields.vectorial.VectorField, optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
Gradient of the field

Return type
Tensor2Field

interpolate_to_grid(*grid*, *, *bc=None*, *fill=None*, *label=None*)

Interpolate the data of this vector field to another grid.

Parameters

- **grid** (*GridBase*) – The grid of the new field onto which the current field is interpolated.
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field, which affects values close to the boundary. If omitted, the argument *fill* is used to determine values outside the domain. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.

- **fill** (*Number, optional*) – Determines how values out of bounds are handled. If *None*, a *ValueError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.
- **label** (*str, optional*) – Name of the returned field
- **self** (VectorField)

Returns
Field of the same rank as the current one.

Return type
VectorField

laplace(*bc*, *out=None*, ***kwargs*)

Apply vector Laplace operator and return result as a field.

The vector Laplacian is a vector field L_α containing the second derivatives of the vector field v_α with respect to the coordinates x_β:

$$L_\alpha = \sum_\beta \frac{\partial^2 v_\alpha}{\partial x_\beta \partial x_\beta}$$

Parameters

- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*. If the special value *None* is given, no boundary conditions are enforced. The user then needs to ensure that the ghost cells are set accordingly.
- **out** (pde.fields.vectorial.VectorField, *optional*) – Optional vector field to which the result is written.
- ****kwargs** – Additional arguments affecting how the operator behaves.

Returns
Laplacian of the field

Return type
VectorField

make_outer_prod_operator(*backend='numba'*)

Return operator calculating the outer product of two vector fields.

> **Warning**
>
> This function does not check types or dimensions.

Parameters

backend (*str*) – The backend (e.g., 'numba' or 'numba_mpi') used for this operator.

Returns

function that takes two instance of `ndarray`, which contain the discretized data of the two operands. An optional third argument can specify the output array to which the result is written.

Return type

BinaryOperatorImplType

outer_product(*other*, *out=None*, ***, *label=None*)

Calculate the outer product of this vector field with another.

Parameters

- **other** (*VectorField*) – The second vector field
- **out** (*Tensor2Field*, optional) – Optional tensorial field to which the result is written.
- **label** (*str, optional*) – Name of the returned field

Returns

result of the operation

Return type

Tensor2Field

plot_components(*kind='auto'*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Visualize all the components of this vector field.

Parameters

- **kind** (*str or list of str*) – Determines the kind of the visualizations. Supported values are *image* or *line*. Alternatively, *auto* determines the best visualization based on the grid.
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.

- **fig** (matplotlib.figures.Figure) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – All additional keyword arguments are forwarded to the actual plotting function of all subplots.

Returns
Instances that contain information to update all the plots with new data later.

Return type
list of PlotReference

rank = 1

to_scalar(*scalar='auto'*, *, *label='scalar `{scalar}`'*)

Return scalar variant of the field.

Parameters

- **scalar** (*str*) – Choose the method to use. Possible choices are *norm*, *max*, *min*, *squared_sum*, *norm_squared*, or an integer specifying which component is returned (indexing starts at *0*). The default value *auto* picks the method automatically: The first (and only) component is returned for real fields on one-dimensional spaces, while the norm of the vector is returned otherwise.
- **label** (*str, optional*) – Name of the returned field

Returns
The scalar field after applying the operation

Return type
pde.fields.scalar.ScalarField

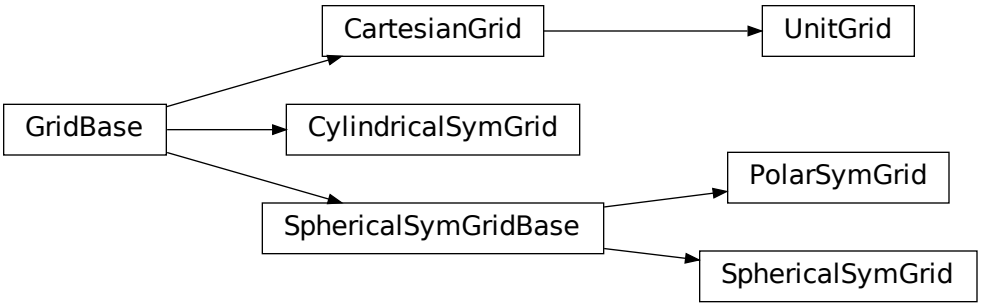
4.3 pde.grids package

Grids define the domains on which PDEs will be solved. In particular, symmetries, periodicities, and the discretizations are defined by the underlying grid.

We only consider regular, orthogonal grids, which are constructed from orthogonal coordinate systems with equidistant discretizations along each axis. The dimension of the space that the grid describes is given by the attribute dim. Cartesian coordinates can be mapped to grid coordinates and the corresponding discretization cells using the method transform().

UnitGrid	D-dimensional Cartesian grid with unit discretization in all directions.
CartesianGrid	D-dimensional Cartesian grid with uniform discretization for each axis.
PolarSymGrid	2-dimensional polar grid assuming angular symmetry.
SphericalSymGrid	3-dimensional spherical grid assuming spherical symmetry.
CylindricalSymGrid	3-dimensional cylindrical grid assuming polar symmetry.

Inheritance structure of the classes:



Subpackages:

4.3.1 pde.grids.boundaries package

This package contains classes for handling the boundary conditions of fields.

Boundary conditions

The mathematical details of boundary conditions for partial differential equations are treated in more detail in the `documentation document`. Since the *pde* package only supports orthogonal grids, boundary conditions generally need to be applied at both ends of each axis. Consequently, methods expecting boundary conditions typically receive a dictionary of conditions for each axes:

```
field = pde.fields.scalar.ScalarField(UnitGrid([16, 16], periodic=[True, False]))
field.laplace(bc={"x": bc_x, "y-": bc_y_lower, "y+": bc_y_upper})
```

If both sides of an axis have the same boundary condition, they can be specified together, e.g., if *bc_y_lower* == *bc_y_upper*, one could have used `{"x": bc_x, "y": bc_y_lower}` instead of the example above. Moreover, it is possible to specify boundary conditions for all sides that have not a specific condition specified using `{"*": default_bc}`. Similarly, boundary conditions for entire axes can be overwritten by conditions specified on one side. Finally, the boundary sides often have aliases defined by the grid, so one can use *left* instead of *x-* and so on.

If an axis is periodic (like the first one in the example above), the only valid boundary conditions are 'periodic' and its cousin 'anti-periodic', which imposes opposite signs on both sides. For non-periodic axes (e.g., the second axis), different boundary conditions can be specified for the lower and upper end of the axis, as in the example above. Typical choices for individual conditions are Dirichlet conditions that enforce a value NUM (specified by *{'value': NUM}*) and Neumann conditions that enforce the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*). The specific choices for the example above could be

```
bc_x = "periodic"
bc_y_lower = {"value": 2}
bc_y_upper = {"derivative": -1}
```

which enforces a value of *2* at the lower side of the y-axis and a derivative (in outward normal direction) of *-1* on the upper side. Instead of plain numbers, which enforce the same condition along the whole boundary, expressions can be used to support inhomogeneous boundary conditions. These mathematical expressions are given as a string that can be parsed by *sympy*. They can depend on all coordinates of the grid. An alternative boundary condition to the example above could thus read

```
bc_y_lower = {"value": "y**2"}
bc_y_upper = {"derivative": "-sin(x)"}
```

Warning

To interpret arbitrary expressions, the package uses `exec()`. It should therefore not be used in a context where malicious input could occur.

Inhomogeneous values can also be specified by directly supplying an array, whose shape needs to be compatible with the boundary, i.e., it needs to have the same shape as the grid but with the dimension of the axis along which the boundary is specified removed.

The package also supports mixed boundary conditions (depending on both the value and the derivative of the field) and imposing a second derivative. An example is

```
bc_y_lower = {"type": "mixed", "value": 2, "const": 7}
bc_y_upper = {"curvature": 2}
```

which enforces the condition $\partial_n c + 2c = 7$ and $\partial_n^2 c = 2$ onto the field c on the lower and upper side of the axis, respectively.

Beside the full specification of boundary conditions, various short-hand notations are supported. If both sides of an axis have the same boundary condition, only one needs to be specified. For instance, `{"x": {"value": 2}}` is equivalent to `{"x-": {"value": 2}, "x+": {"value": 2}}` and imposes a value of *2* on both sides of the x-axis. In the special case where all sides have the same boundary conditions, only this condition can be specified instead of the full dictionary, e.g.

```
field = pde.fields.scalar.ScalarField(UnitGrid([16, 16], periodic=False))
field.laplace(bc={"value": 2})
```

imposes a value of *2* on all sides of the grid. Finally, the special values `"auto_periodic_neumann"` and `"auto_periodic_dirichlet"` impose periodic boundary conditions for periodic axis and a vanishing derivative or value otherwise. For example,

```
field = pde.fields.scalar.ScalarField(UnitGrid([16, 16], periodic=[True, False]))
field.laplace(bc="auto_periodic_neumann")
```

enforces periodic boundary conditions on the first axis, while the second one has standard Neumann conditions.

Note

Derivatives are given relative to the outward normal vector, such that positive derivatives correspond to a function that increases across the boundary.

If more complex boundary conditions are required, a custom function that directly sets the boundary conditions can also be supplied. This special approach comes with few checks, so only use it in exceptional circumstances. The following example shows a setter function, which sets specific boundary conditions in the x-direction and periodic conditions in the y-direction of a grid with two axes.

```
def setter(data, args=None):
    data[0, :] = data[1, :]  # Vanishing derivative at left side
    data[-1, :] = 2 - data[-2, :]  # Fixed value `1` at right side
```

(continues on next page)

(continued from previous page)

```
    data[:, 0] = data[:, -2]  # Periodic BC at top
    data[:, -1] = data[:, 1]  # Periodic BC at bottom


field = pde.fields.scalar.ScalarField(UnitGrid([16, 16], periodic=[False, True]))
field.laplace(bc=setter)
```

Boundaries overview

The *boundaries* package defines the following classes:

Local boundary conditions:

- *DirichletBC*: Imposing a constant value of the field at the boundary
- *ExpressionValueBC*: Imposing the value of the field at the boundary given by an expression or a python function
- *NeumannBC*: Imposing a constant derivative of the field in the outward normal direction at the boundary
- *ExpressionDerivativeBC*: Imposing the derivative of the field in the outward normal direction at the boundary given by an expression or a python function
- *MixedBC*: Imposing the derivative of the field in the outward normal direction proportional to its value at the boundary
- *ExpressionMixedBC*: Imposing the derivative of the field in the outward normal direction proportional to its value at the boundary with coefficients given by expressions or python functions
- *CurvatureBC*: Imposing a constant second derivative (curvature) of the field at the boundary

There are corresponding classes that only affect the normal component of a field, which can be useful when dealing with vector and tensor fields: *NormalDirichletBC*, *NormalNeumannBC*, *NormalMixedBC*, and *NormalCurvatureBC*.

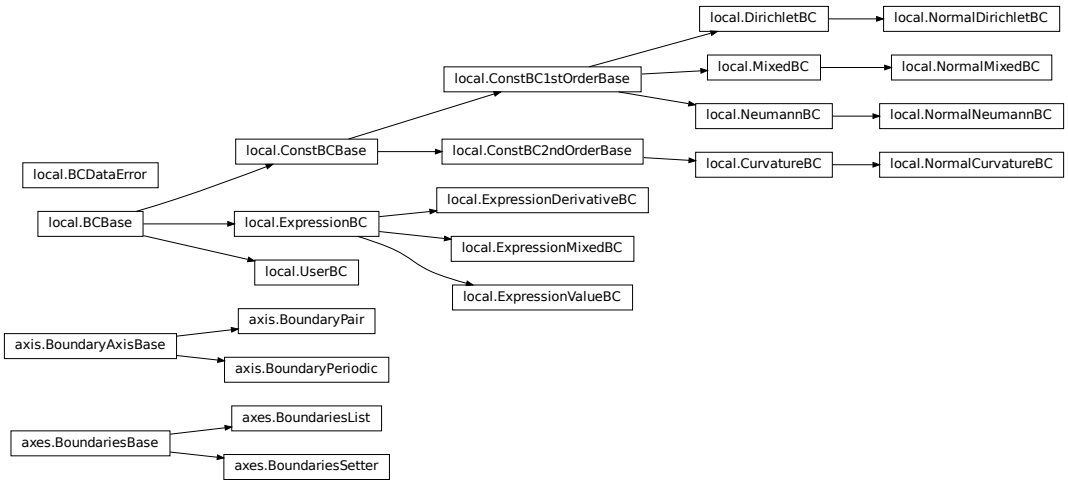
Boundaries for an axis:

- *BoundaryPair*: Uses the local boundary conditions to specify the two boundaries along an axis
- *BoundaryPeriodic*: Indicates that an axis has periodic boundary conditions

BoundariesList for all axes of a grid:

- *BoundariesList*: Collection of boundaries to describe conditions for all axes
- *BoundariesSetter*: Describes custom function setting virtual points to impose boundary conditions

Inheritance structure of the classes:



The details of the classes are explained below:

pde.grids.boundaries.axes module

This module handles the boundaries of all axes of a grid.

BoundariesBase	Base class keeping information about how to set conditions on all boundaries.
BoundariesList	Defines boundary conditions for all axes individually.
BoundariesSetter	Represents a function that sets ghost cells to determine boundary conditions.
set_default_bc	Set a default boundary condition.

class BoundariesBase

Bases: object

Base class keeping information about how to set conditions on all boundaries.

classmethod from_data(*data*, *grid*, *rank=0*, ***kwargs*)

Creates all boundaries from given data.

Parameters

- **data** (*str or dict or callable*) – Data that describes the boundaries. If this is a callable, we create *BoundariesSetter*. In all other cases *BoundariesList* is created and *data* can either be string denoting a specific boundary condition applied to all sides or a dictionary with detailed information.
- **grid** (*GridBase*) – The grid with which the boundary condition is associated
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- ****kwargs** – In some cases additional data can be specified or is even required.

Return type
BoundariesBase

classmethod get_help()
Return information on how boundary conditions can be set.

Return type
str

grid: *GridBase*
Grid on which boundaries are defined.

Type
GridBase

rank: int
The tensorial rank of the field for this boundary condition.

Type
int

set_ghost_cells(*data_full*, *, *args=None*)
Set the ghost cells for all boundaries.

Parameters
- **data_full** (ndarray) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

Return type
None

class BoundariesList(*boundaries*)
Bases: *BoundariesBase*

Defines boundary conditions for all axes individually.

Initialize with a list of boundaries.

Parameters
boundaries (*list*) – List of boundary axis conditions

property boundaries: Iterator[*BCBase*]
Iterator over all non-periodic boundaries.

check_value_rank(*rank*)
Check whether the values at the boundaries have the correct rank.

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Return type
None

Throws:
RuntimeError: if any value does not have rank *rank*

copy()
Create a copy of the current boundaries.

Return type
BoundariesList

classmethod from_data(*data*, *, *grid*, *rank=0*, ***kwargs*)

Creates all boundaries from given data.

Parameters

- **data** (*str or dict*) – Data that describes the boundaries. This should either be a string naming a boundary condition or a dictionary with detailed information.
- **grid** (*GridBase*) – The grid with which the boundary condition is associated
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- ****kwargs** – Additional keyword arguments (unused)

Return type
BoundariesList

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
str

property periodic: list[bool]

a boolean array indicating which dimensions are periodic according to the boundary conditions.

Type
ndarray

set_ghost_cells(*data_full*, *, *set_corners=False*, *args=None*)

Set the ghost cells for all boundaries.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **set_corners** (*bool*) – Determines whether the corner cells are set using interpolation
- **args** – Additional arguments that might be supported by special boundary conditions.

Return type
None

class BoundariesSetter(*setter*, *grid*, *rank=0*)

Bases: *BoundariesBase*

Represents a function that sets ghost cells to determine boundary conditions.

The function must have accept a ndarray, which contains the full field data including the ghost points, and a second, optional argument, which is a dictionary containing additional parameters, like the current time point *t* in case of a simulation.

Example

Here is an example for a simple boundary setter, which sets specific boundary conditions in the x-direction and periodic conditions in the y-direction of a grid with two axes. Note that this boundary condition will not work for grids with other number of axes and no additional checks are performed.

```
def setter(data, args=None):
    data[0, :] = data[1, :]  # Vanishing derivative at left side
    data[-1, :] = 2 - data[-2, :]  # Fixed value `1` at right side
    data[:, 0] = data[:, -2]  # Periodic BC at top
    data[:, -1] = data[:, 1]  # Periodic BC at bottom
```

Parameters

- **setter** (GhostCellSetter)
- **grid** (GridBase)
- **rank** (*int*)

classmethod from_data(*data*, *grid*, *rank=0*, ***kwargs*)

Creates all boundaries from given data.

Parameters

- **data** (*callable*) – Function that sets the ghost cell
- **grid** (*GridBase*) – The grid with which the boundary condition is associated
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- ****kwargs** – Additional keyword arguments (unused)

Return type

BoundariesSetter

set_ghost_cells(*data_full*, ***, *args=None*)

Set the ghost cells for all boundaries.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

Return type

None

set_default_bc(*bc_data*, *default*)

Set a default boundary condition.

Parameters

- **bc_data** (*str or list or tuple or dict or callable*) – User-supplied data specifying boundary conditions
- **default** (*dict | str |* BCBase) – Default condition that should be imposed where user conditions are not given

Returns

Modified *bc_data* with added defaults

Return type

dict[str, dict | str | BCBase] | dict | str | BCBase | tuple[dict | str | BCBase, dict | str | BCBase] | BoundaryAxisBase | *Sequence*[dict[str, dict | str | BCBase] | dict | str | BCBase | tuple[dict | str | BCBase, dict | str | BCBase] | BoundaryAxisBase] | *Callable* | BoundariesBase

pde.grids.boundaries.axis module

This module handles the boundaries of a single axis of a grid. There are generally only two options, depending on whether the axis of the underlying grid is defined as periodic or not. If it is periodic, the class *BoundaryPeriodic* should be used, while non-periodic axes have more option, which are represented by *BoundaryPair*.

BoundaryAxisBase	Base class for defining boundaries of a single axis in a grid.
BoundaryPair	Represents the two boundaries of an axis along a single dimension.
BoundaryPeriodic	Represent a periodic axis.

class BoundaryAxisBase(*low*, *high*)

Bases: object

Base class for defining boundaries of a single axis in a grid.

Parameters

- **low** (*BCBase*) – Instance describing the lower boundary
- **high** (*BCBase*) – Instance describing the upper boundary

property axis: int

The axis along which the boundaries are defined

Type
int

check_value_rank(*rank*)

Check whether the values at the boundaries have the correct rank.

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Return type
None

copy()

Return a copy of itself, but with a reference to the same grid.

Return type
BoundaryAxisBase

classmethod get_help()

Return information on how boundary conditions can be set.

Return type
str

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
tuple[str, str]

get_sparse_matrix_data(*idx*)

Sets the elements of the sparse representation of this condition.

Parameters

idx (*tuple*) – The index of the point that must lie on the boundary condition

Returns

A constant value and a dictionary with indices and factors that can be used to calculate this virtual point

Return type

float, dict

property grid: *GridBase*

Underlying grid.

Type

GridBase

high: *BCBase*

Boundary condition at upper end.

Type

BCBase

low: *BCBase*

Boundary condition at lower end.

Type

BCBase

property periodic: bool

whether the axis is periodic

Type

bool

property rank: int

rank of the associated boundary condition

Type

int

set_ghost_cells(*data_full*, ***, *args=None*)

Set the ghost cell values for all boundaries.

Parameters

- **data_full** (*ndarray*) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

Return type

None

class BoundaryPair(*low*, *high*)

Bases: *BoundaryAxisBase*

Represents the two boundaries of an axis along a single dimension.

Parameters

- **low** (*BCBase*) – Instance describing the lower boundary

- **high** (*BCBase*) – Instance describing the upper boundary

check_value_rank(*rank*)

Check whether the values at the boundaries have the correct rank.

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Return type
None

Throws:
RuntimeError: if the value does not have rank *rank*

classmethod from_data(*grid*, *axis*, *data*, *rank=0*)

Create boundary pair from some data.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **data** (*str or dict*) – Data that describes the boundary pair
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

Returns
the instance created from the data

Return type
BoundaryPair

Throws:
ValueError if *data* cannot be interpreted as a boundary pair

class BoundaryPeriodic(*grid*, *axis*, *rank=0*, *flip_sign=False*)

Bases: *BoundaryAxisBase*

Represent a periodic axis.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **flip_sign** (*bool*) – Impose different signs on the two sides of the boundary

check_value_rank(*rank*)

Check whether the values at the boundaries have the correct rank.

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Return type
None

copy()

Return a copy of itself, but with a reference to the same grid.

Return type
BoundaryPeriodic

property flip_sign

Whether different signs are imposed on the two sides of the boundary

Type
bool

get_boundary_axis(*grid*, *axis*, *data*, *rank=0*)

Return object representing the boundary condition for a single axis.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **data** (*str or tuple or dict*) – Data describing the boundary conditions for this axis
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

Returns
Appropriate boundary condition for the axis

Return type
BoundaryAxisBase

pde.grids.boundaries.local module

This module contains classes for handling a single boundary of a non-periodic axis. Since an axis has two boundary, we simply distinguish them by a boolean flag *upper*, which is *True* for the side of the axis with the larger coordinate.

The module currently supports the following standard boundary conditions:

- *DirichletBC*: Imposing the value of a field at the boundary
- *NeumannBC*: Imposing the derivative of a field in the outward normal direction at the boundary
- *MixedBC*: Imposing the derivative of a field in the outward normal direction proportional to its value at the boundary
- *CurvatureBC*: Imposing the second derivative (curvature) of a field at the boundary

There are also variants of these boundary conditions that only affect the normal components of a vector or tensor field: *NormalDirichletBC*, *NormalNeumannBC*, *NormalMixedBC*, and *NormalCurvatureBC*.

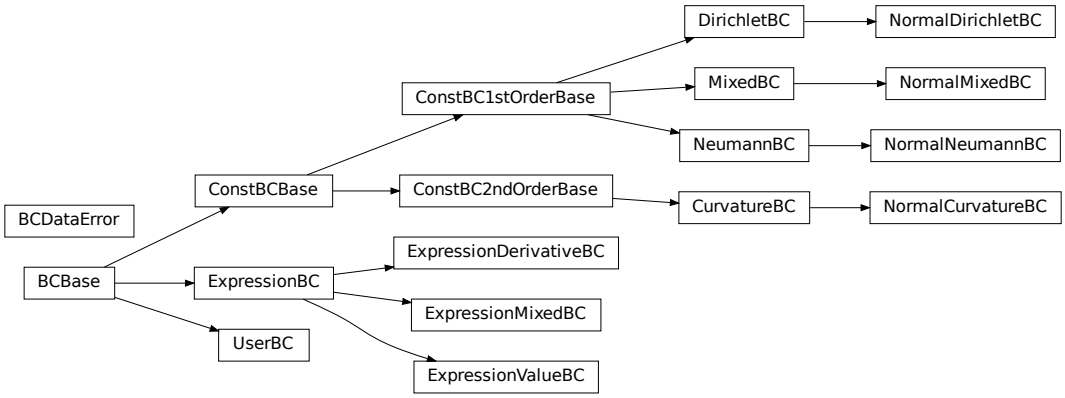
Finally, there are more specialized classes, which offer greater flexibility, but might also require a slightly deeper understanding for proper use:

- *ExpressionValueBC*: Imposing the value of a field at the boundary based on a mathematical expression or a python function.
- *ExpressionDerivativeBC*: Imposing the derivative of a field in the outward normal direction at the boundary based on a mathematical expression or a python function.
- *ExpressionMixedBC*: Imposing a mixed (Robin) boundary condition using mathematical expressions or python functions.
- *UserBC*: Allows full control for setting virtual points, values, or derivatives. The boundary conditions are never enforced automatically. It is thus the user's responsibility to ensure virtual points are set correctly before operators are applied. To set boundary conditions a dictionary `{TARGET: value}` must be supplied as argument *args* to

`set_ghost_cells()` or the numba equivalent. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Note that derivatives are generally given in the direction of the outward normal vector, such that positive derivatives correspond to a function that increases across the boundary.

Inheritance structure of the classes:



class BCBase(*grid*, *axis*, *upper*, *, *rank=0*)

Bases: `object`

Represents a single boundary in an BoundaryPair instance.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

property axis_coord: float

value of the coordinate that defines this boundary condition

Type
float

check_value_rank(*rank*)

Check whether the values at the boundaries have the correct rank.

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Return type
None

Throws:
RuntimeError: if the value does not have rank *rank*

copy(*upper=None*, *rank=None*)

Parameters

- **upper** (*bool | None*)
- **rank** (*int | None*)

Return type
Self

classmethod from_data(*grid*, *axis*, *upper*, *data*, *, *rank=0*)

Create boundary from some data.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Indicates whether this boundary condition is associated with the upper or lower side of the axis.
- **data** (*str or dict*) – Data that describes the boundary
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

Returns
the instance created from the data

Return type
BCBase

Throws:
ValueError if *data* cannot be interpreted as a boundary condition

classmethod from_dict(*grid*, *axis*, *upper*, *data*, *, *rank=0*)

Create boundary from data given in dictionary.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Indicates whether this boundary condition is associated with the upper or lower side of the axis.
- **data** (*dict*) – The dictionary defining the boundary condition
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

Return type
BCBase

classmethod from_str(*grid*, *axis*, *upper*, *condition*, *, *rank=0*, ***kwargs*)

Creates boundary from a given string identifier.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined

- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Indicates whether this boundary condition is associated with the upper or lower side of the axis.
- **condition** (*str*) – Identifies the boundary condition
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- ****kwargs** – Additional arguments passed to the constructor

Return type
BCBase

classmethod get_help()

Return information on how boundary conditions can be set.

Return type
str

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
str

get_sparse_matrix_data(*idx*)

Parameters
idx (*tuple[int, ...]*)

Return type
tuple[float, dict[int, float]]

get_virtual_point(*arr*, *idx=None*)

Parameters
idx (*tuple[int, ...] | None*)

Return type
float

homogeneous: bool

determines whether the boundary condition depends on space

Type
bool

names: list[str]

identifiers used to specify the given boundary class

Type
list

normal: bool = False

determines whether the boundary condition only affects normal components.

If this flag is *False*, boundary conditions must specify values for all components of the field. If *True*, only the normal components at the boundary are specified.

Type
bool

property periodic: bool

whether the boundary condition is periodic

Type
bool

abstractmethod set_ghost_cells(*data_full*, *, *args=None*)

Set the ghost cell values for this boundary.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** (ndarray) – Determines what boundary conditions are set. *args* should be set to `{TARGET: value}`. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Return type
None

to_subgrid(*subgrid*)

Converts this boundary condition to one valid for a given subgrid.

Parameters
subgrid (GridBase) – Grid of the new boundary conditions

Returns
Boundary conditions valid on the subgrid

Return type
BCBase

exception BCDataError

Bases: ValueError

Exception that signals that incompatible data was supplied for the BC.

class ConstBC1stOrderBase(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *ConstBCBase*

Represents a single boundary in an BoundaryPair instance.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.

- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_sparse_matrix_data(*idx*)

Sets the elements of the sparse representation of this condition.

Parameters

idx (*tuple*) – The index of the point that must lie on the boundary condition

Returns

A constant value and a dictionary with indices and factors that can be used to calculate this virtual point

Return type

float, dict

get_virtual_point(*arr*, *idx=None*)

Calculate the value of the virtual point outside the boundary.

Parameters

- **arr** (*array*) – The data values associated with the grid
- **idx** (*tuple*) – The index of the point to evaluate. This is a tuple of length *grid.num_axes* with the either -1 or *dim* as the entry for the axis associated with this boundary condition. Here, *dim* is the dimension of the axis. The index is optional if dim == 1.

Returns

Value at the virtual support point

Return type

float

abstractmethod get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns

the data structure associated with this virtual point

Return type

tuple

set_ghost_cells(*data_full*, ***, *args=None*)

Set the ghost cell values for this boundary.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** (ndarray) – Determines what boundary conditions are set. *args* should be set to `{TARGET: value}`. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Return type

None

class ConstBC2ndOrderBase(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *ConstBCBase*

Abstract base class for boundary conditions of 2nd order.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or `ndarray`) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_sparse_matrix_data(*idx*)

Sets the elements of the sparse representation of this condition.

Parameters

idx (*tuple*) – The index of the point that must lie on the boundary condition

Returns

A constant value and a dictionary with indices and factors that can be used to calculate this virtual point

Return type

float, dict

get_virtual_point(*arr*, *idx=None*)

Calculate the value of the virtual point outside the boundary.

Parameters

- **arr** (*array*) – The data values associated with the grid
- **idx** (*tuple*) – The index of the point to evaluate. This is a tuple of length *grid.num_axes* with the either -1 or *dim* as the entry for the axis associated with this boundary condition. Here, *dim* is the dimension of the axis. The index is optional if dim == 1.

Returns

Value at the virtual support point

Return type

float

abstractmethod get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns
the data structure associated with this virtual point

Return type
tuple

set_ghost_cells(*data_full*, *, *args=None*)

Set the ghost cell values for this boundary.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** (ndarray) – Determines what boundary conditions are set. *args* should be set to `{TARGET: value}`. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Return type
None

class ConstBCBase(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *BCBase*

Base class representing a boundary whose virtual point is set from constants.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

copy(*upper=None*, *rank=None*, *value=None*)

Return a copy of itself, but with a reference to the same grid.

Parameters

- **upper** (*bool*) – The upper flag of the returned object

- **rank** (*int*) – The rank of the returned object
- **value** (float or ndarray or str) – The value of the returned object
- **self** (ConstBCBase)

Return type
ConstBCBase

link_value(*value*)

Link value of this boundary condition to external array.

Parameters
value (ndarray) – The array to link to

to_subgrid(*subgrid*)

Converts this boundary condition to one valid for a given subgrid.

Parameters

- **subgrid** (GridBase) – Grid of the new boundary conditions
- **self** (ConstBCBase)

Returns
Boundary conditions valid on the subgrid

Return type
ConstBCBase

property value: NumericArray

value_is_linked: bool

flag that indicates whether the value associated with this boundary condition is linked to ndarray managed by external code.

Type
bool

class CurvatureBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *ConstBC2ndOrderBase*

Represents a boundary condition imposing the 2nd normal derivative at the boundary.

Warning

This implementation uses exec() and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters

field_name (*str*) – Name of the field to use in the representation

Return type

str

get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns

the data structure associated with this virtual point

Return type

tuple

names = ['curvature', 'second_derivative', 'extrapolate']

identifiers used to specify the given boundary class

Type

list

class DirichletBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *ConstBC1stOrderBase*

Represents a boundary condition imposing the value.

Warning

This implementation uses exec() and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters

field_name (*str*) – Name of the field to use in the representation

Return type

str

get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns

the data structure associated with this virtual point

Return type

tuple

names = ['value', 'dirichlet']

identifiers used to specify the given boundary class

Type

list

class ExpressionBC(*grid*, *axis*, *upper*, ***, *rank=0*, *value=0*, *const=0*, *target='virtual_point'*, *user_funcs=None*, *value_cell=None*)

Bases: *BCBase*

Represents a boundary whose virtual point is calculated from an expression.

The expression is given as a string that can be parsed by `sympy` or as a function. The expression can contain typical mathematical operators and may depend on the value at the last support point next to the boundary (*value*), spatial coordinates defined by the grid marking the boundary point (e.g., *x* or *r*), and time *t*.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or callable*) – An expression that determines the value of the boundary condition. Alternatively, this can be a function with signature *(value, dx, *coords, t)* that determines the value of *target* from the field value *value* (the value of the adjacent cell unless *value_cell* is specified), the spatial discretization *dx* in the direction perpendicular to the wall, the spatial coordinates of the wall point, and time *t*. Ideally, this function should be numba-compilable since simulations might otherwise be very slow.
- **const** (*float or str or callable*) – An expression similar to *value*, which is only used for mixed (Robin) boundary conditions. Note that the implementation currently does

not support that one argument is given as a callable function while the other is defined via an expression, so both need to have the same type.

- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, *mixed*, and *virtual_point*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **value_cell** (*int*) – Determines which cells is read to determine the field value that is used as *value* in the expression or the function call. The default (*None*) specifies the adjacent cell.

copy(*upper=None*, *rank=None*)

Return a copy of itself, but with a reference to the same grid.

Parameters

- **upper** (*bool*) – The upper flag of the returned object
- **rank** (*int*) – The rank of the returned object
- **self** (ExpressionBC)

Return type
ExpressionBC

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
str

get_sparse_matrix_data(*idx*)

Parameters
idx (*tuple[int, ...]*)

Return type
tuple[float, dict[int, float]]

get_virtual_point(*arr*, *idx=None*)

Parameters
idx (*tuple[int, ...] | None*)

Return type
float

names: list[str] = ['virtual_point']

identifiers used to specify the given boundary class

Type
list

set_ghost_cells(*data_full*, ***, *args=None*)

Set the ghost cell values for this boundary.

Parameters

- **data_full** (ndarray) – The full field data including ghost points

- **args** (ndarray) – Determines what boundary conditions are set. *args* should be set to `{TARGET: value}`. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Return type
None

to_subgrid(*subgrid*)

Converts this boundary condition to one valid for a given subgrid.

Parameters

- **subgrid** (GridBase) – Grid of the new boundary conditions
- **self** (ExpressionBC)

Returns
Boundary conditions valid on the subgrid

Return type
BCBase

class ExpressionDerivativeBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*, *target='derivative'*, *user_funcs=None*, *value_cell=None*)

Bases: *ExpressionBC*

Represents a boundary whose outward derivative is calculated from an expression.

The expression is given as a string and will be parsed by `sympy`. The expression can contain typical mathematical operators and may depend on the value at the last support point next to the boundary (*value*), spatial coordinates defined by the grid marking the boundary point (e.g., *x* or *r*), and time *t*.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or callable*) – An expression that determines the value of the boundary condition. Alternatively, this can be a function with signature *(value, dx, *coords, t)* that determines the value of *target* from the field value *value* (the value of the adjacent cell unless *value_cell* is specified), the spatial discretization *dx* in the direction perpendicular to the wall, the spatial coordinates of the wall point, and time *t*. Ideally, this function should be numba-compilable since simulations might otherwise be very slow.
- **const** (*float or str or callable*) – An expression similar to *value*, which is only used for mixed (Robin) boundary conditions. Note that the implementation currently does

not support that one argument is given as a callable function while the other is defined via an expression, so both need to have the same type.

- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, *mixed*, and *virtual_point*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **value_cell** (*int*) – Determines which cells is read to determine the field value that is used as *value* in the expression or the function call. The default (*None*) specifies the adjacent cell.

names: list[str] = ['derivative_expression', 'derivative_expr']

identifiers used to specify the given boundary class

Type
list

class ExpressionMixedBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*, *const=0*, *target='mixed'*, *user_funcs=None*, *value_cell=None*)

Bases: *ExpressionBC*

Represents a boundary whose outward derivative is calculated from an expression.

The expression is given as a string and will be parsed by `sympy`. The expression can contain typical mathematical operators and may depend on the value at the last support point next to the boundary (*value*), spatial coordinates defined by the grid marking the boundary point (e.g., *x* or *r*), and time *t*.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or callable*) – An expression that determines the value of the boundary condition. Alternatively, this can be a function with signature *(value, dx, *coords, t)* that determines the value of *target* from the field value *value* (the value of the adjacent cell unless *value_cell* is specified), the spatial discretization *dx* in the direction perpendicular to the wall, the spatial coordinates of the wall point, and time *t*. Ideally, this function should be numba-compilable since simulations might otherwise be very slow.
- **const** (*float or str or callable*) – An expression similar to *value*, which is only used for mixed (Robin) boundary conditions. Note that the implementation currently does not support that one argument is given as a callable function while the other is defined via an expression, so both need to have the same type.
- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, *mixed*, and *virtual_point*.

- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression
- **value_cell** (*int*) – Determines which cells is read to determine the field value that is used as *value* in the expression or the function call. The default (*None*) specifies the adjacent cell.

names: list[str] = ['mixed_expression', 'mixed_expr', 'robin_expression', 'robin_expr']

identifiers used to specify the given boundary class

Type
list

class ExpressionValueBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*, *target='value'*, *user_funcs=None*, *value_cell=None*)

Bases: *ExpressionBC*

Represents a boundary whose value is calculated from an expression.

The expression is given as a string and will be parsed by `sympy`. The expression can contain typical mathematical operators and may depend on the value at the last support point next to the boundary (*value*), spatial coordinates defined by the grid marking the boundary point (e.g., *x* or *r*), and time *t*.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or callable*) – An expression that determines the value of the boundary condition. Alternatively, this can be a function with signature *(value, dx, *coords, t)* that determines the value of *target* from the field value *value* (the value of the adjacent cell unless *value_cell* is specified), the spatial discretization *dx* in the direction perpendicular to the wall, the spatial coordinates of the wall point, and time *t*. Ideally, this function should be numba-compilable since simulations might otherwise be very slow.
- **const** (*float or str or callable*) – An expression similar to *value*, which is only used for mixed (Robin) boundary conditions. Note that the implementation currently does not support that one argument is given as a callable function while the other is defined via an expression, so both need to have the same type.
- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, *mixed*, and *virtual_point*.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression

- **value_cell** (*int*) – Determines which cells is read to determine the field value that is used as *value* in the expression or the function call. The default (*None*) specifies the adjacent cell.

names: list[str] = ['value_expression', 'value_expr']

identifiers used to specify the given boundary class

Type
list

class MixedBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*, *const=0*)

Bases: *ConstBC1stOrderBase*

represents a mixed (or Robin) boundary condition imposing a derivative in the outward normal direction of the boundary that is given by an affine function involving the actual value:

$$\partial_n c + \gamma c = \beta$$

Here, c is the field to which the condition is applied, γ quantifies the influence of the field and β is the constant term. Note that $\gamma = 0$ corresponds to Dirichlet conditions imposing β as the derivative. Conversely, $\gamma \to \infty$ corresponds to imposing a zero value on c.

This condition can be enforced by using one of the following variants

```
bc = {"mixed": VALUE}
bc = {"type": "mixed", "value": VALUE, "const": CONST}
```

where *VALUE* corresponds to γ and *CONST* to β.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or array*) – The parameter γ quantifying the influence of the field onto its normal derivative. If *value* is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.
- **const** (float or ndarray or str) – The parameter β determining the constant term for the boundary condition. Supports the same input as *value*.

copy(*upper=None*, *rank=None*, *value=None*, *const=None*)

Return a copy of itself, but with a reference to the same grid.

Parameters

- **upper** (*bool*) – The upper flag of the returned object
- **rank** (*int*) – The rank of the returned object
- **value** (float or ndarray or str) – The value of the returned object
- **const** (float or ndarray or str) – The constant value of the returned object
- **self** (MixedBC)

Return type
MixedBC

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
str

get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns
the data structure associated with this virtual point

Return type
tuple

names = ['mixed', 'robin']

identifiers used to specify the given boundary class

Type
list

to_subgrid(*subgrid*)

Converts this boundary condition to one valid for a given subgrid.

Parameters

- **subgrid** (GridBase) – Grid of the new boundary conditions
- **self** (MixedBC)

Returns
Boundary conditions valid on the subgrid

Return type
ConstBCBase

class NeumannBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *ConstBC1stOrderBase*

Represents a boundary condition imposing the derivative in the outward normal direction of the boundary.

Warning

This implementation uses exec() and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.

- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters

field_name (*str*) – Name of the field to use in the representation

Return type

str

get_virtual_point_data()

Return data suitable for calculating virtual points.

Returns

the data structure associated with this virtual point

Return type

tuple

names = ['derivative', 'neumann']

identifiers used to specify the given boundary class

Type

list

class NormalCurvatureBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *CurvatureBC*

Represents a boundary condition imposing the 2nd normal derivative onto the normal components at the boundary.

Warning

This implementation uses exec() and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

names = ['normal_curvature']

identifiers used to specify the given boundary class

Type
list

normal = True

determines whether the boundary condition only affects normal components.

If this flag is *False*, boundary conditions must specify values for all components of the field. If *True*, only the normal components at the boundary are specified.

Type
bool

class NormalDirichletBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *DirichletBC*

Represents a boundary condition imposing the value on normal components.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or `ndarray`) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

names = ['normal_value', 'normal_dirichlet', 'dirichlet_normal']

identifiers used to specify the given boundary class

Type
list

normal = True

determines whether the boundary condition only affects normal components.

If this flag is *False*, boundary conditions must specify values for all components of the field. If *True*, only the normal components at the boundary are specified.

Type
bool

class NormalMixedBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*, *const=0*)

Bases: *MixedBC*

represents a mixed (or Robin) boundary condition setting the derivative of the normal components in the outward normal direction of the boundary using an affine function involving the actual value:

$$\partial_n c + \gamma c = \beta$$

Here, c is the field to which the condition is applied, γ quantifies the influence of the field and β is the constant term. Note that $\gamma = 0$ corresponds to Dirichlet conditions imposing β as the derivative. Conversely, $\gamma \to \infty$ corresponds to imposing a zero value on c.

This condition can be enforced by using one of the following variants

```
bc = {"mixed": VALUE}
bc = {"type": "mixed", "value": VALUE, "const": CONST}
```

where *VALUE* corresponds to γ and *CONST* to β.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (*float or str or array*) – The parameter γ quantifying the influence of the field onto its normal derivative. If *value* is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.
- **const** (float or ndarray or str) – The parameter β determining the constant term for the boundary condition. Supports the same input as *value*.

names = ['normal_mixed', 'normal_robin']

identifiers used to specify the given boundary class

Type
list

normal = True

determines whether the boundary condition only affects normal components.

If this flag is *False*, boundary conditions must specify values for all components of the field. If *True*, only the normal components at the boundary are specified.

Type
bool

class NormalNeumannBC(*grid*, *axis*, *upper*, *, *rank=0*, *value=0*)

Bases: *NeumannBC*

Represents a boundary condition imposing the derivative of normal components in the outward normal direction of the boundary.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur. However, the function is safe when *value* cannot be an arbitrary string.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated
- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **value** (float or str or `ndarray`) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

names = ['normal_derivative', 'normal_neumann', 'neumann_normal']

identifiers used to specify the given boundary class

Type
list

normal = True

determines whether the boundary condition only affects normal components.

If this flag is *False*, boundary conditions must specify values for all components of the field. If *True*, only the normal components at the boundary are specified.

Type
bool

class UserBC(*grid*, *axis*, *upper*, *, *rank=0*)

Bases: *BCBase*

Represents a boundary whose virtual points are set by the user.

Boundary conditions will only be set when a dictionary `{TARGET: value}` is supplied as argument *args* to *set_ghost_cells()* or the numba equivalent. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

> **Warning**
>
> This implies that the boundary conditions are never enforced automatically, e.g., when evaluating an operator. It is thus the user's responsibility to ensure virtual points are set correctly before operators are applied.

Parameters

- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated

- **upper** (*bool*) – Flag indicating whether this boundary condition is associated with the upper side of an axis or not. In essence, this determines the direction of the local normal vector of the boundary.
- **rank** (*int*) – The tensorial rank of the field for this boundary condition

copy(*upper=None*, *rank=None*)

Return a copy of itself, but with a reference to the same grid.

Parameters

- **upper** (*bool*) – The upper flag of the returned object
- **rank** (*int*) – The rank of the returned object

Return type
Self

get_mathematical_representation(*field_name='C'*)

Return mathematical representation of the boundary condition.

Parameters
field_name (*str*) – Name of the field to use in the representation

Return type
str

names: list[str] = ['user']

identifiers used to specify the given boundary class

Type
list

set_ghost_cells(*data_full*, ***, *args=None*)

Set the ghost cell values for this boundary.

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** (ndarray) – Determines what boundary conditions are set. *args* should be set to `{TARGET: value}`. Here, *TARGET* determines how the *value* is interpreted and what boundary condition is actually enforced: the value of the virtual points directly (*virtual_point*), the value of the field at the boundary (*value*) or the outward derivative of the field at the boundary (*derivative*).

Return type
None

to_subgrid(*subgrid*)

Converts this boundary condition to one valid for a given subgrid.

Parameters
subgrid (GridBase) – Grid of the new boundary conditions

Returns
Boundary conditions valid on the subgrid

Return type
BCBase

registered_boundary_condition_classes()

Returns all boundary condition classes that are currently defined.

Returns
a dictionary with the names of the boundary condition classes

Return type
dict

registered_boundary_condition_names()

Returns all named boundary conditions that are currently defined.

Returns
a dictionary with the names of the boundary conditions that can be used

Return type
dict

4.3.2 pde.grids.coordinates package

Package collecting classes representing orthonormal coordinate systems.

BipolarCoordinates	2-dimensional bipolar coordinates.
BisphericalCoordinates	3-dimensional bispherical coordinates.
CartesianCoordinates	N-dimensional Cartesian coordinates.
CylindricalCoordinates	3-dimensional cylindrical coordinates.
PolarCoordinates	2-dimensional polar coordinates.
SphericalCoordinates	3-dimensional spherical coordinates.

class BipolarCoordinates(*scale_parameter=1*)

Bases: *CoordinatesBase*

2-dimensional bipolar coordinates.

Parameters
scale_parameter (*float*)

axes: list[str] = ['σ', 'τ']
name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, 6.283185307179586), (-inf, inf)]
the limits of each coordinate axis

Type
list of tuple

dim: int = 2
spatial dimension of the coordinate system

Type
int

class BisphericalCoordinates(*scale_parameter=1*)

Bases: *CoordinatesBase*

3-dimensional bispherical coordinates.

Parameters
scale_parameter (*float*)

axes: list[str] = ['σ', 'τ', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, 3.141592653589793), (-inf, inf), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

class CartesianCoordinates(*dim*)

Bases: *CoordinatesBase*

N-dimensional Cartesian coordinates.

Parameters
dim (*int*) – Dimension of the Cartesian coordinate system

class CylindricalCoordinates

Bases: *CoordinatesBase*

3-dimensional cylindrical coordinates.

axes: list[str] = ['r', 'ψ', 'z']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 6.283185307179586), (-inf, inf)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

class PolarCoordinates

Bases: *CoordinatesBase*

2-dimensional polar coordinates.

axes: list[str] = ['r', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 2

spatial dimension of the coordinate system

Type
int

class SphericalCoordinates

Bases: *CoordinatesBase*

3-dimensional spherical coordinates.

axes: list[str] = ['r', 'θ', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 3.141592653589793), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

major_axis = 0

pde.grids.coordinates.base module

Base classes for coordinate systems.

CoordinatesBase	Base class for orthonormal coordinate systems.
DimensionError	Exception indicating that dimensions were inconsistent.

class CoordinatesBase

Bases: object

Base class for orthonormal coordinate systems.

axes: list[str]

name of each coordinate axis

Type
list

basis_rotation(*points*)

Returns rotation matrix rotating basis vectors to Cartesian coordinates.

Parameters
points (ndarray) – Coordinates of the point(s)

Returns
Rotation matrices for all points. The returned array has the shape *(dim, dim) + points_shape*, assuming *points* has the shape *points_shape + (dim,)*.

Return type
ndarray

cell_volume(*c_low*, *c_high*)

Calculate the volume between coordinate lines.

Parameters

- **c_low** (ndarray) – Lower values of the coordinate lines enclosing the volume
- **c_high** (ndarray) – Upper values of the coordinate lines enclosing the volume

Returns
Enclosed volumes for all given cells

Return type
ndarray

coordinate_limits: list[tuple[float, float]]

the limits of each coordinate axis

Type
list of tuple

dim: int

spatial dimension of the coordinate system

Type
int

distance(*p1*, *p2*, *, *axis=-1*)

Calculate the distance between two points.

Parameters

- **p1** (ndarray) – First position
- **p2** (ndarray) – Second position
- **axis** (*int*) – Determines the axis along which the coordinates of the points are given

Returns
Distance between the two positions

Return type
float

mapping_jacobian(*points*)

Returns the Jacobian matrix of the coordinate mapping.

Parameters
points (ndarray) – Coordinates of the point(s)

Returns
The Jacobian

Return type
ndarray

metric(*points*)

Calculate the metric tensor at coordinate points.

Parameters
points (ndarray) – The coordinates of the points

Returns
Metric tensor at the points

Return type
ndarray

pos_from_cart(*points*, *, *axis=-1*)

Convert Cartesian coordinates to coordinates in this system.

Parameters

- **points** (ndarray) – Points given in Cartesian coordinates.
- **axis** (*int*) – Determines the axis along which the coordinates of the points are given

Returns
Points given in the coordinates of this system

Return type
ndarray

pos_to_cart(*points*, *, *axis=-1*)

Convert coordinates to Cartesian coordinates.

Parameters

- **points** (ndarray) – The coordinates of points in the current coordinate system
- **axis** (*int*) – Determines the axis along which the coordinates of the points are given

Returns
Cartesian coordinates of the points

Return type
ndarray

scale_factors(*points*)

Calculate the scale factors at various points.

Parameters
points (ndarray) – The coordinates of the points

Returns
Scale factors at the points

Return type
ndarray

vec_to_cart(*points*, *components*)
Convert the vectors at given points to a Cartesian basis.

Parameters
- **points** (ndarray) – The coordinates of the point(s) where the vectors are specified.
- **components** (ndarray) – The components of the vectors at the given points

Returns
The vectors specified at the same position but with components given in Cartesian coordinates.

Return type
ndarray

volume_factor(*points*)
Calculate the volume factors at various points.

Parameters
points (ndarray) – Coordinates of the point(s)

Returns
Volume factors at the points

Return type
ndarray

exception DimensionError
Bases: ValueError

Exception indicating that dimensions were inconsistent.

pde.grids.coordinates.bipolar module

class BipolarCoordinates(*scale_parameter=1*)
Bases: *CoordinatesBase*

2-dimensional bipolar coordinates.

Parameters
scale_parameter (*float*)

axes: list[str] = ['σ', 'τ']
name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, 6.283185307179586), (-inf, inf)]
the limits of each coordinate axis

Type
list of tuple

dim: int = 2

spatial dimension of the coordinate system

Type
int

pde.grids.coordinates.bispherical module

class BisphericalCoordinates(*scale_parameter=1*)

Bases: *CoordinatesBase*

3-dimensional bispherical coordinates.

Parameters
scale_parameter (*float*)

axes: list[str] = ['σ', 'τ', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, 3.141592653589793), (-inf, inf), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

pde.grids.coordinates.cartesian module

class CartesianCoordinates(*dim*)

Bases: *CoordinatesBase*

N-dimensional Cartesian coordinates.

Parameters
dim (*int*) – Dimension of the Cartesian coordinate system

pde.grids.coordinates.cylindrical module

class CylindricalCoordinates

Bases: *CoordinatesBase*

3-dimensional cylindrical coordinates.

axes: list[str] = ['r', 'ψ', 'z']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 6.283185307179586), (-inf, inf)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

pde.grids.coordinates.polar module

class PolarCoordinates

Bases: *CoordinatesBase*

2-dimensional polar coordinates.

axes: list[str] = ['r', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 2

spatial dimension of the coordinate system

Type
int

pde.grids.coordinates.spherical module

class SphericalCoordinates

Bases: *CoordinatesBase*

3-dimensional spherical coordinates.

axes: list[str] = ['r', 'θ', 'ψ']

name of each coordinate axis

Type
list

coordinate_limits: list[tuple[float, float]] = [(0, inf), (0, 3.141592653589793), (0, 6.283185307179586)]

the limits of each coordinate axis

Type
list of tuple

dim: int = 3

spatial dimension of the coordinate system

Type
int

major_axis = 0

4.3.3 pde.grids.base module

Defines the base class for all grids.

GridBase	Base class for all grids defining common methods and interfaces.
DomainError	Exception indicating that point lies outside domain.
PeriodicityError	Exception indicating that the grid periodicity is inconsistent.
discretize_interval	Construct a list of equidistantly placed intervals.
registered_grids	Returns all grids available in the package.
registered_operators	Returns all operators that are currently defined.

exception DimensionError

Bases: ValueError

Exception indicating that dimensions were inconsistent.

exception DomainError

Bases: ValueError

Exception indicating that point lies outside domain.

class GridBase

Bases: object

Base class for all grids defining common methods and interfaces.

Initialize the grid.

assert_grid_compatible(*other*)

Checks whether *other* is compatible with the current grid.

Parameters
other (*GridBase*) – The grid compared to this one

Raises
ValueError – if grids are not compatible

Return type
None

axes: list[str]

Names of all axes that are described by the grid

Type
list

property axes_bounds: tuple[tuple[float, float], ...]
lower and upper bounds of each axis

Type
tuple

property axes_coords: tuple[FloatingArray, ...]
coordinates of the cells for each axis

Type
tuple

axes_symmetric: list[str] = []
The names of the additional axes that the fields do not depend on, e.g. along which they are constant.

Type
list

boundary_names: dict[str, tuple[int, bool]] = {}
Names of boundaries to select them conveniently

Type
dict

c: *CoordinatesBase*
Coordinates of the grid.

Type
CoordinatesBase

cell_coords
coordinate values for all axes of each cell.

Type
ndarray

cell_volume_data: Sequence[FloatOrArray] | None
Information about the size of discretization cells

Type
list

cell_volumes
volume of each cell.

Type
ndarray

compatible_with(*other*)
Tests whether this grid is compatible with other grids.

Grids are compatible when they cover the same area with the same discretization. The difference to equality is that compatible grids do not need to have the same periodicity in their boundaries.

Parameters
other (*GridBase*) – The other grid to test against

Returns
Whether the grid is compatible

Return type
bool

contains_point(*points*, *, *coords='cartesian'*)

Check whether the point is contained in the grid.

Parameters

- **points** (ndarray) – Coordinates of the point(s)
- **coords** (*str*) – The coordinate system in which the points are given

Returns

A boolean array indicating which points lie within the grid

Return type

ndarray

coordinate_arrays

for each axes: coordinate values for all cells

Type

tuple

coordinate_constraints: list[int] = []

axes that not described explicitly

Type

list

copy()

Return a copy of the grid.

Return type

GridBase

difference_vector(*p1*, *p2*, *, *coords='grid'*)

Return Cartesian vector(s) pointing from p1 to p2.

In case of periodic boundary conditions, the shortest vector is returned.

Parameters

- **p1** (ndarray) – First point(s)
- **p2** (ndarray) – Second point(s)
- **coords** (*str*) – Coordinate system in which points are specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.

Returns

The difference vectors between the points with periodic boundary conditions applied.

Return type

ndarray

property dim: int

The spatial dimension in which the grid is embedded

Type

int

property discretization: FloatingArray

the linear size of a cell along each axis.

Type

numpy.array

distance(*p1*, *p2*, *, *coords='grid'*)

Calculate the distance between two points given in real coordinates.

This takes periodic boundary conditions into account if necessary.

Parameters

- **p1** (ndarray) – First position
- **p2** (ndarray) – Second position
- **coords** (*str*) – Coordinate system in which points are specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.

Returns
Distance between the two positions

Return type
float

classmethod from_bounds(*bounds*, *shape*, *periodic*)

Parameters

- **bounds** (*Sequence[tuple[float, float]]*)
- **shape** (*Sequence[int]*)
- **periodic** (*Sequence[bool]*)

Return type
GridBase

classmethod from_state(*state*)

Create a field from a stored *state*.

Parameters
state (*str* or *dict*) – The state from which the grid is reconstructed. If *state* is a string, it is decoded as JSON, which should yield a *dict*.

Returns
Grid re-created from the state

Return type
GridBase

get_axis_index(*key*, *allow_symmetric=True*)

Return the index belonging to an axis.

Parameters

- **key** (*int or str*) – The index or name of an axis
- **allow_symmetric** (*bool*) – Whether axes with assumed symmetry are included

Returns
The index of the axis

Return type
int

get_boundary_conditions(*bc='auto_periodic_neumann'*, *rank=0*)

Constructs boundary conditions from a flexible data format.

Parameters

- **bc** (*str or list or tuple or dict*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **rank** (*int*) – The tensorial rank of the value associated with the boundary conditions.

Returns
The boundary conditions for all axes.

Return type
BoundariesBase

Raises

- **ValueError** – If the data given in *bc* cannot be read
- ***PeriodicityError*** – If the boundaries are not compatible with the periodic axes of the grid.

get_image_data(*data*)

Return a 2d-image of the data.

Parameters
data (ndarray) – The values at the grid points

Returns
A dictionary with information about the data convenient for plotting.

Return type
dict

get_line_data(*data*, *extract='auto'*)

Return a line cut through the grid.

Parameters

- **data** (ndarray) – The values at the grid points
- **extract** (*str*) – Determines which cut is done through the grid. Possible choices depend on the actual grid.

Returns
A dictionary with information about the line cut, which is convenient for plotting.

Return type
dict

get_random_point(*, *boundary_distance=0*, *coords='cartesian'*, *rng=None*)

Return a random point within the grid.

Parameters

- **boundary_distance** (*float*) – The minimal distance this point needs to have from all boundaries.

- **coords** (*str*) – Determines the coordinate system in which the point is specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.
- **rng** (Generator) – Random number generator (default: default_rng())

Returns
The coordinates of the random point

Return type
ndarray

get_vector_data(*data*, ***kwargs*)

Return data to visualize vector field.

Parameters

- **data** (ndarray) – The vectorial values at the grid points
- ****kwargs** – Arguments forwarded to *get_image_data()*.

Returns
A dictionary with information about the data convenient for plotting.

Return type
dict

integrate(*data*, *axes=None*)

Integrates the discretized data over the grid.

Parameters

- **data** (ndarray) – The values at the support points of the grid that need to be integrated.
- **axes** (*list of int, optional*) – The axes along which the integral is performed. If omitted, all axes are integrated over.

Returns
The values integrated over the entire grid

Return type
ndarray

iter_mirror_points(*point*, *with_self=False*, *only_periodic=True*)

Generates all mirror points corresponding to *point*

Parameters

- **point** (ndarray) – The point within the grid
- **with_self** (*bool*) – Whether to include the point itself
- **only_periodic** (*bool*) – Whether to only mirror along periodic axes

Returns
A generator yielding the coordinates that correspond to mirrors

Return type
Iterator[FloatingArray]

make_cell_volume_compiled(*flat_index=False*)

Return a compiled function returning the volume of a grid cell.

Parameters
flat_index (*bool*) – When True, cell_volumes are indexed by a single integer into the flattened array.

Returns
returning the volume of the chosen cell

Return type
function

make_inserter_compiled(*, *with_ghost_cells=False*)

Return a compiled function to insert values at interpolated positions.

Parameters
with_ghost_cells (*bool*) – Flag indicating that the interpolator should work on the full data array that includes values for the grid points. If this is the case, the boundaries are not checked and the coordinates are used as is.

Returns
A function with signature (data, position, amount), where *data* is the numpy array containing the field data, position is denotes the position in grid coordinates, and *amount* is the that is to be added to the field.

Return type
callable

make_integrator(*backend='numpy'*)

Return function that can be used to integrates discretized data over the grid.

If this function is used in a multiprocessing run (using MPI), the integrals are performed on all subgrids and then accumulated. Each process then receives the same value representing the global integral.

Parameters
backend (*str*) – The backend to use for making the operator

Returns
A function that takes a numpy array and returns the integral with the correct weights given by the cell volumes.

Return type
callable

make_normalize_point_compiled(*reflect=True*)

Return a compiled function that normalizes a point.

Here, the point is assumed to be specified by the physical values along the non-symmetric axes of the grid. Normalizing points is useful to make sure they lie within the domain of the grid. This function respects periodic boundary conditions and can also reflect points off the boundary.

Parameters
reflect (*bool*) – Flag determining whether coordinates along non-periodic axes are reflected to lie in the valid range. If *False*, such coordinates are left unchanged and only periodic boundary conditions are enforced.

Returns
A function that takes a `ndarray` as an argument, which describes the coordinates of the points. This array is modified in-place!

Return type
callable

make_operator(*operator*, *bc*, *, *backend='default'*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator with boundary conditions.

Parameters

- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **bc** (*str or list or tuple or dict*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **backend** (*str*) – The backend to use for making the operator
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Return type
OperatorType

The returned function takes the discretized data on the grid as an input and returns the data to which the operator *operator* has been applied. The function only takes the valid grid points and allocates memory for the ghost points internally to apply the boundary conditions specified as *bc*. Note that the function supports an optional argument *out*, which if given should provide space for the valid output array without the ghost cells. The result of the operator is then written into this output array.

The function also accepts an optional parameter *args*, which is forwarded to *set_ghost_cells*. This allows setting boundary conditions based on external parameters, like time. Note that the *numba* backend requires a special calling convention to make it compatible with Numba. The following example shows how to pass the current time *t*:

```
from pde.backends.numba.utils import numba_dict

operator = grid.make_operator(
    "laplace", bc={"value_expression": "t"}, backend="numba"
)
operator(field.data, args=numba_dict(t=t))
```

Returns
the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray = None, args=None).

Return type
callable

Parameters

- **operator** (*str | OperatorInfo*)
- **bc** (*BoundariesData*)
- **backend** (*str | BackendBase[TNativeArray]*)
- **dtype** (*DTypeLike | None*)

make_operator_no_bc(*operator*, *, *backend='default'*, *dtype=None*, ***kwargs*)

Return a compiled function applying an operator without boundary conditions.

A function that takes the discretized full data as an input and an array of valid data points to which the result of applying the operator is written.

> **Note**
>
> The resulting function does not check whether the ghost cells of the input array have been supplied with sensible values. It is the responsibility of the user to set the values of the ghost cells beforehand. Use this function only if you absolutely know what you're doing. In all other cases, *make_operator()* is probably the better choice.

Parameters

- **operator** (*str*) – Identifier for the operator. Some examples are 'laplace', 'gradient', or 'divergence'. The registered operators for this grid can be obtained from the *operators* attribute.
- **backend** (*str*) – The backend to use for making the operator
- **dtype** (*numpy dtype*) – The data type of the field.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

Returns

the function that applies the operator. This function has the signature (arr: NumericArray, out: NumericArray), so they *out* array need to be supplied explicitly.

Return type

callable

normalize_point(*point*, *, *reflect=False*)

Normalize grid coordinates by applying periodic boundary conditions.

Here, points are assumed to be specified by the physical values along the non-symmetric axes of the grid, e.g., by grid coordinates. Normalizing points is useful to make sure they lie within the domain of the grid. This function respects periodic boundary conditions and can also reflect points off the boundary if *reflect* = *True*.

Parameters

- **point** (ndarray) – Coordinates of a single point or an array of points, where the last axis denotes the point coordinates (e.g., a list of points).
- **reflect** (*bool*) – Flag determining whether coordinates along non-periodic axes are reflected to lie in the valid range. If *False*, such coordinates are left unchanged and only periodic boundary conditions are enforced.

Returns

The respective coordinates with periodic boundary conditions applied.

Return type

ndarray

num_axes: int

Number of axes that are *not* assumed symmetrically

Type

int

property num_cells: int

the number of cells in this grid

Type
int

property numba_type: str

represents type of the grid data in numba signatures

Type
str

operators: set[str] = {}

names of all operators defined for this grid

Type
set

property periodic: list[bool]

Flags that describe which axes are periodic

Type
list

plot()

Visualize the grid.

Return type
None

point_from_cartesian(*points*)

Convert points given in Cartesian coordinates to grid coordinates.

Parameters
points (ndarray) – Points given in Cartesian coordinates.

Returns
Points given in the coordinates of the grid

Return type
ndarray

point_to_cartesian(*points*)

Convert coordinates of a point in grid coordinates to Cartesian coordinates.

Parameters
points (ndarray) – The grid coordinates of the points

Returns
The Cartesian coordinates of the point

Return type
ndarray

property shape: tuple[int, ...]

the number of support points of each axis

Type
tuple of int

slice(*indices*)

Return a subgrid of only the specified axes.

Parameters
indices (*list*) – Indices indicating the axes that are retained in the subgrid

Returns
The subgrid

Return type
GridBase

abstract property state: dict[str, Any]

all information required for reconstructing the grid

Type
dict

property state_serialized: str

JSON-serialized version of the state of this grid

Type
str

transform(*coordinates*, *source*, *target*)

Converts coordinates from one coordinate system to another.

Supported coordinate systems include the following:

- ***cartesian*:**
  Cartesian coordinates where each point carries *dim* values. These are the true physical coordinates in space.
- ***grid*:**
  Coordinates values in the coordinate system defined by the grid. A point is thus characterized by *grid.num_axes* values.
- ***cell*:**
  Normalized grid coordinates based on indexing the discretization cells. A point is characterized by *grid.num_axes* values and the range of values for a given axis is between *0* and *N*, where *N* is the number of grid points. Consequently, the integral part of the cell coordinate denotes the cell, while the fractional part denotes the relative position within the cell. In particular, the cell center is located at *i + 0.5* with *i = 0, …, N-1*.

Note

Some conversion might involve projections if the coordinate system imposes symmetries. For instance, converting 3d Cartesian coordinates to grid coordinates in a spherically symmetric grid will only return the radius from the origin. Conversely, converting these grid coordinates back to 3d Cartesian coordinates will only return coordinates along a particular ray originating at the origin.

Parameters

- **coordinates** (*ndarray*) – The coordinates to convert
- **source** (*str*) – The source coordinate system
- **target** (*str*) – The target coordinate system

Returns
The transformed coordinates

Return type
ndarray

property typical_discretization: float
the average side length of the cells

Type
float

uniform_cell_volumes
returns True if all cell volumes are the same

Type
bool

property volume: float
total volume of the grid

Type
float

exception PeriodicityError
Bases: RuntimeError

Exception indicating that the grid periodicity is inconsistent.

registered_grids()
Returns all grids available in the package.

Returns
a dictionary with the names of the grids and the associated class

Return type
dict

registered_operators()
Returns all operators that are currently defined.

Returns
a dictionary with the names of the operators defined for each grid class

Return type
dict

4.3.4 pde.grids.cartesian module

Cartesian grids of arbitrary dimension.

CartesianGrid	D-dimensional Cartesian grid with uniform discretization for each axis.
UnitGrid	D-dimensional Cartesian grid with unit discretization in all directions.

class CartesianGrid(*bounds*, *shape*, *periodic=False*)

Bases: *GridBase*

D-dimensional Cartesian grid with uniform discretization for each axis.

The grids can be thought of as a collection of n-dimensional boxes, called cells, of equal length in each dimension. The bounds then defined the total volume covered by these cells, while the cell coordinates give the location of the box centers. We index the boxes starting from 0 along each dimension. Consequently, the cell $i - \frac{1}{2}$ corresponds to the left edge of the covered interval and the index $i + \frac{1}{2}$ corresponds to the right edge, when the dimension is covered by d boxes.

In particular, the discretization along dimension k is defined as

$$x_i^{(k)} = x_{\text{min}}^{(k)} + \left(i + \frac{1}{2}\right) \Delta x^{(k)} \quad \text{for} \quad i = 0, \ldots, N^{(k)} - 1$$
$$\Delta x^{(k)} = \frac{x_{\text{max}}^{(k)} - x_{\text{min}}^{(k)}}{N^{(k)}}$$

where $N^{(k)}$ is the number of cells along this dimension. Consequently, cells have dimension $\Delta x^{(k)}$ and cover the interval $[x_{\text{min}}^{(k)}, x_{\text{max}}^{(k)}]$.

Parameters

- **bounds** (*list of tuple*) – Give the coordinate range for each axis. This should be a tuple of two number (lower and upper bound) for each axis. The length of *bounds* thus determines the grid dimension.
- **shape** (*list*) – The number of support points for each axis. The length of *shape* needs to match the grid dimension.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. This is either a list of booleans defining periodicity for each individual axis or a single boolean value specifying the same periodicity for all axes.

property cell_volume_data

Size associated with each cell.

cuboid: *Cuboid*

difference_vector(*p1*, *p2*, *, *coords='grid'*)

Return Cartesian vector(s) pointing from p1 to p2.

In case of periodic boundary conditions, the shortest vector is returned.

Parameters

- **p1** (ndarray) – First point(s)
- **p2** (ndarray) – Second point(s)
- **coords** (*str*) – Coordinate system in which points are specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.

Returns

The difference vectors between the points with periodic boundary conditions applied.

Return type

ndarray

classmethod from_bounds(*bounds*, *shape*, *periodic*)

Parameters

- **bounds** (*tuple*) – Give the coordinate range for each axis. This should be a tuple of two number (lower and upper bound) for each axis. The length of *bounds* thus determines the grid dimension.
- **shape** (*tuple*) – The number of support points for each axis. The length of *shape* needs to match the grid dimension.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. This is either a list of booleans defining periodicity for each individual axis or a single boolean value specifying the same periodicity for all axes.

Returns
representing the region chosen by bounds

Return type
CartesianGrid

classmethod from_state(*state*)

Create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

Returns
the grid re-created from the state data

Return type
CartesianGrid

get_image_data(*data*)

Return a 2d-image of the data.

Parameters
data (ndarray) – The values at the grid points

Returns
A dictionary with information about the data convenient for plotting.

Return type
dict

get_line_data(*data*, *extract='auto'*)

Return a line cut through the given data.

Parameters

- **data** (ndarray) – The values at the grid points
- **extract** (*str*) – Determines which cut is done through the grid. Possible choices are (default is *cut_0*):
  - *cut_#*: return values along the axis specified by # and use the mid point along all other axes.
  - *project_#*: average values for all axes, except axis #.

  Here, # can either be a zero-based index (from 0 to dim-1) or a letter denoting the axis.

Returns
A dictionary with information about the line cut, which is convenient for plotting.

Return type
dict

get_random_point(*, *boundary_distance=0*, *coords='cartesian'*, *rng=None*)

Return a random point within the grid.

Parameters

- **boundary_distance** (*float*) – The minimal distance this point needs to have from all boundaries.
- **coords** (*str*) – Determines the coordinate system in which the point is specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.
- **rng** (Generator) – Random number generator (default: default_rng())

Returns

The coordinates of the point

Return type

ndarray

get_vector_data(*data*, ***kwargs*)

Return data to visualize vector field.

Parameters

- **data** (ndarray) – The vectorial values at the grid points
- ****kwargs** – Arguments forwarded to *get_image_data()*.

Returns

A dictionary with information about the data convenient for plotting.

Return type

dict

iter_mirror_points(*point*, *with_self=False*, *only_periodic=True*)

Generates all mirror points corresponding to *point*

Parameters

- **point** (ndarray) – The point within the grid
- **with_self** (*bool*) – Whether to include the point itself
- **only_periodic** (*bool*) – Whether to only mirror along periodic axes

Returns

A generator yielding the coordinates that correspond to mirrors

Return type

Generator

plot(**args*, *title=None*, *filename=None*, *action='auto'*, *ax_style=None*, *fig_style=None*, *ax=None*, ***kwargs*)

Visualize the grid.

Parameters

- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically.
- **filename** (*str, optional*) – If given, the plot is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, matplotlib.pyplot.show() will be called to show the plot. If the value is *auto* or *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given.

- **ax_style** (*dict*) – Dictionary with properties that will be changed on the axis after the plot has been drawn by calling `matplotlib.pyplot.setp()`. A special item i this dictionary is *use_offset*, which is flag that can be used to control whether offset are shown along the axes of the plot.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **ax** (`matplotlib.axes.Axes`) – Figure axes to be used for plotting. The special value "create" creates a new figure, while "reuse" attempts to reuse an existing figure, which is the default.
- ****kwargs** – Extra arguments are passed on the to the matplotlib plotting routines, e.g., to set the color of the lines

slice(*indices*)

Return a subgrid of only the specified axes.

Parameters

indices (*list*) – Indices indicating the axes that are retained in the subgrid

Returns

The subgrid

Return type

CartesianGrid

property state: dict[str, Any]

the state of the grid

Type

dict

property volume: float

total volume of the grid

Type

float

class UnitGrid(*shape*, *periodic=False*)

Bases: *CartesianGrid*

D-dimensional Cartesian grid with unit discretization in all directions.

The grids can be thought of as a collection of d-dimensional cells of unit length. The *shape* parameter determines how many boxes there are in each direction. The cells are enumerated starting with 0, so the last cell has index $n - 1$ if there are n cells along a dimension. A given cell i extends from coordinates i to $i + 1$, so the midpoint is at $i + \frac{1}{2}$, which is the cell coordinate. Taken together, the cells covers the interval $[0, n]$ along this dimension.

Parameters

- **shape** (*list*) – The number of support points for each axis. The dimension of the grid is given by *len(shape)*.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. This is either a list of booleans defining periodicity for each individual axis or a single boolean value specifying the same periodicity for all axes.

classmethod from_state(*state*)

Create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

Return type
UnitGrid

slice(*indices*)

Return a subgrid of only the specified axes.

Parameters
indices (*list*) – Indices indicating the axes that are retained in the subgrid

Returns
The subgrid

Return type
UnitGrid

property state: dict[str, Any]

the state of the grid

Type
dict

to_cartesian()

Convert unit grid to *CartesianGrid*

Returns
The equivalent cartesian grid

Return type
CartesianGrid

4.3.5 pde.grids.cylindrical module

Cylindrical grids with azimuthal symmetry.

class CylindricalSymGrid(*radius*, *bounds_z*, *shape*, *periodic_z=False*)

Bases: *GridBase*

3-dimensional cylindrical grid assuming polar symmetry.

The polar symmetry implies that states only depend on the radial and axial coordinates r and z, respectively. These are discretized uniformly,

$$r_i = R_{\text{inner}} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N_r - 1 \quad \text{with} \quad \Delta r = \frac{R_{\text{outer}} - R_{\text{inner}}}{N_r}$$

$$z_j = z_{\text{min}} + \left(j + \frac{1}{2}\right)\Delta z \quad \text{for} \quad j = 0, \ldots, N_z - 1 \quad \text{with} \quad \Delta z = \frac{z_{\text{max}} - z_{\text{min}}}{N_z}$$

where R_{outer} is the outer radius of the grid, R_{inner} corresponds to a possible inner radius (which is zero by default), and z_{min} and z_{max} denote the respective lower and upper bounds of the axial direction, so that $z_{\text{max}} - z_{\text{min}}$ is the total height. The two axes are discretized by N_r and N_z support points, respectively.

Warning

The order of components in the vector and tensor fields defined on this grid are still (r, ϕ, z). To avoid any confusion it might thus be best to access components by name instead of index, e.g., use `field['z']` instead of `field[2]`.

Parameters

- **radius** (*float or tuple of floats*) – radius R_{outer} in case a simple float is given. If a tuple is supplied it is interpreted as the inner and outer radius, $(R_{\text{inner}}, R_{\text{outer}})$.
- **bounds_z** (*tuple*) – The lower and upper bound of the z-axis
- **shape** (*tuple*) – The number of support points in r and z direction, respectively. The same number is used for both if a single value is given.
- **periodic_z** (*bool*) – Determines whether the z-axis has periodic boundary conditions.

boundary_names = {'bottom': (1, False), 'inner': (0, False), 'outer': (0, True), 'top': (1, True)}

Names of boundaries to select them conveniently

Type
dict

c = CylindricalCoordinates()

Coordinates of the grid.

Type
CoordinatesBase

cell_volume_data

Information about the size of discretization cells

Type
list

coordinate_constraints = [0, 1]

axes that not described explicitly

Type
list

difference_vector(*p1*, *p2*, *, *coords='grid'*)

Return Cartesian vector(s) pointing from p1 to p2.

In case of periodic boundary conditions, the shortest vector is returned.

Parameters

- **p1** (ndarray) – First point(s)
- **p2** (ndarray) – Second point(s)
- **coords** (*str*) – Coordinate system in which points are specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.

Returns
The difference vectors between the points with periodic boundary conditions applied.

Return type
ndarray

classmethod from_bounds(*bounds*, *shape*, *periodic*)

Parameters

- **bounds** (*tuple*) – Give the coordinate range for each axis. This should be a tuple of two number (lower and upper bound) for each axis. The length of *bounds* must be 2.

- **shape** (*tuple*) – The number of support points for each axis. The length of *shape* needs to be 2.
- **periodic** (*bool or list*) – Specifies which axes possess periodic boundary conditions. The first entry is ignored.

Returns
grid representing the region chosen by bounds

Return type
CylindricalSymGrid

classmethod from_state(*state*)

Create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

Return type
CylindricalSymGrid

get_cartesian_grid(*mode='valid'*)

Return a Cartesian grid for this Cylindrical one.

Parameters
mode (*str*) – Determines how the grid is determined. Setting it to 'valid' only returns points that are fully resolved in the cylindrical grid, e.g., the cylinder is circumscribed. Conversely, 'full' returns all data, so the cylinder is inscribed.

Returns
The requested grid

Return type
pde.grids.cartesian.CartesianGrid

get_image_data(*data*)

Return a 2d-image of the data.

Parameters
data (ndarray) – The values at the grid points

Returns
A dictionary with information about the image, which is convenient for plotting.

Return type
dict

get_line_data(*data*, *extract='auto'*)

Return a line cut for the cylindrical grid.

Parameters

- **data** (ndarray) – The values at the grid points
- **extract** (*str*) – Determines which cut is done through the grid. Possible choices are (default is *cut_axial*):
  - *cut_z* or *cut_axial*: values along the axial coordinate for $r = 0$.
  - *project_z* or *project_axial*: average values for each axial position (radial average).
  - *project_r* or *project_radial*: average values for each radial position (axial average)

Returns
A dictionary with information about the line cut, which is convenient for plotting.

Return type
dict

get_random_point(*, *boundary_distance=0*, *avoid_center=False*, *coords='cartesian'*, *rng=None*)

Return a random point within the grid.

Parameters

- **boundary_distance** (*float*) – The minimal distance this point needs to have from all boundaries.
- **avoid_center** (*bool*) – Determines whether the boundary distance should also be kept from the center, i.e., whether points close to the center are returned.
- **coords** (*str*) – Determines the coordinate system in which the point is specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.
- **rng** (Generator) – Random number generator (default: default_rng())

Returns
The coordinates of the point

Return type
ndarray

property has_hole: bool

whether the inner radius is larger than zero

Type
bool

iter_mirror_points(*point*, *with_self=False*, *only_periodic=True*)

Generates all mirror points corresponding to *point*

Parameters

- **point** (ndarray) – The point within the grid
- **with_self** (*bool*) – Whether to include the point itself
- **only_periodic** (*bool*) – Whether to only mirror along periodic axes

Returns
A generator yielding the coordinates that correspond to mirrors

Return type
Generator

property length: float

length of the cylinder

Type
float

property radius: float | tuple[float, float]

radius of the sphere

Type
float

slice(*indices*)

Return a subgrid of only the specified axes.

Parameters

indices (*list*) – Indices indicating the axes that are retained in the subgrid

Returns

CartesianGrid or *PolarSymGrid*: The subgrid

Return type

CartesianGrid | *PolarSymGrid*

property state: dict[str, Any]

all information required for reconstructing the grid

Type

dict

property volume: float

total volume of the grid

Type

float

4.3.6 pde.grids.spherical module

Spherically-symmetric grids in 2 and 3 dimensions. These are grids that only discretize the radial direction, assuming symmetry with respect to all angles. This choice implies that differential operators might not be applicable to all fields. For instance, the divergence of a vector field on a spherical grid can only be represented as a scalar field on the same grid if the θ-component of the vector field vanishes.

PolarSymGrid	2-dimensional polar grid assuming angular symmetry.
SphericalSymGrid	3-dimensional spherical grid assuming spherical symmetry.
volume_from_radius	Return the volume of a sphere with a given radius.

class PolarSymGrid(*radius*, *shape*)

Bases: *SphericalSymGridBase*

2-dimensional polar grid assuming angular symmetry.

The angular symmetry implies that states only depend on the radial coordinate r, which is discretized uniformly as

$$r_i = R_{\text{inner}} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N-1 \quad \text{with} \quad \Delta r = \frac{R_{\text{outer}} - R_{\text{inner}}}{N}$$

where R_{outer} is the outer radius of the grid and R_{inner} corresponds to a possible inner radius, which is zero by default. The radial direction is discretized by N support points.

Parameters

- **radius** (*float or tuple of floats*) – Radius R_{outer} in case a simple float is given. If a tuple is supplied it is interpreted as the inner and outer radius, $(R_{\text{inner}}, R_{\text{outer}})$.
- **shape** (*tuple or int*) – The number N of support points along the radial coordinate.

c = PolarCoordinates()

Coordinates of the grid.

Type
CoordinatesBase

coordinate_constraints = [0, 1]
axes that not described explicitly

Type
list

class SphericalSymGrid(*radius*, *shape*)

Bases: *SphericalSymGridBase*

3-dimensional spherical grid assuming spherical symmetry.

The symmetry implies that states only depend on the radial coordinate r, which is discretized as follows:

$$r_i = R_{\text{inner}} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N-1 \quad \text{with} \quad \Delta r = \frac{R_{\text{outer}} - R_{\text{inner}}}{N}$$

where R_{outer} is the outer radius of the grid and R_{inner} corresponds to a possible inner radius, which is zero by default. The radial direction is discretized by N support points.

Warning

Not all results of differential operators on vectorial and tensorial fields can be expressed in terms of fields that only depend on the radial coordinate r. In particular, the gradient of a vector field can only be calculated if the azimuthal component of the vector field vanishes. Similarly, the divergence of a tensor field can only be taken in special situations.

Parameters

- **radius** (*float or tuple of floats*) – Radius R_{outer} in case a simple float is given. If a tuple is supplied it is interpreted as the inner and outer radius, $(R_{\text{inner}}, R_{\text{outer}})$.
- **shape** (*tuple or int*) – The number N of support points along the radial coordinate.

c = SphericalCoordinates()
Coordinates of the grid.

Type
CoordinatesBase

coordinate_constraints = [0, 1, 2]
axes that not described explicitly

Type
list

class SphericalSymGridBase(*radius*, *shape*)

Bases: *GridBase*

Base class for d-dimensional spherical grids with angular symmetry.

The angular symmetry implies that states only depend on the radial coordinate r, which is discretized uniformly as

$$r_i = R_{\text{inner}} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N-1 \quad \text{with} \quad \Delta r = \frac{R_{\text{outer}} - R_{\text{inner}}}{N}$$

where R_{outer} is the outer radius of the grid and R_{inner} corresponds to a possible inner radius, which is zero by default. The radial direction is discretized by N support points.

Parameters

- **radius** (*float or tuple of floats*) – Radius R_{outer} in case a simple float is given. If a tuple is supplied it is interpreted as the inner and outer radius, $(R_{\text{inner}}, R_{\text{outer}})$.
- **shape** (*tuple or int*) – The number N of support points along the radial coordinate.

boundary_names = {'inner': (0, False), 'outer': (0, True)}

Names of boundaries to select them conveniently

Type
dict

cell_volume_data

Information about the size of discretization cells

Type
list

classmethod from_bounds(*bounds*, *shape*, *periodic*)

Parameters

- **bounds** (*tuple*) – Give the coordinate range for the radial axis.
- **shape** (*tuple*) – The number of support points for the radial axis
- **periodic** (*tuple*) – Flag indicating whether the grid is periodic (not used here)

Returns
represents the region chosen by bounds

Return type
SphericalGridBase

classmethod from_state(*state*)

Create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

Return type
SphericalSymGridBase

get_cartesian_grid(*mode='valid'*, *num=None*)

Return a Cartesian grid for this spherical one.

Parameters

- **mode** (*str*) – Determines how the grid is determined. Setting it to 'valid' (or 'inscribed') only returns points that are fully resolved in the spherical grid, e.g., the Cartesian grid is inscribed in the sphere. Conversely, 'full' (or 'circumscribed') returns all data, so the Cartesian grid is circumscribed.
- **num** (*int*) – Number of support points along each axis of the returned grid.

Returns
The requested grid

Return type
pde.grids.cartesian.CartesianGrid

get_image_data(*data*, *, *performance_goal='speed'*, *fill_value=0*, *masked=True*)

Return a 2d-image of the data.

Parameters

- **data** (ndarray) – The values at the grid points
- **performance_goal** (*str*) – Determines the method chosen for interpolation. Possible options are *speed* and *quality*.
- **fill_value** (*float*) – The value assigned to invalid positions (those inside the hole or outside the region).
- **masked** (*bool*) – Whether a numpy.ma.MaskedArray is returned for the data instead of the normal ndarray.

Returns

dict: A dictionary with information about the image, which is convenient for plotting.

Return type

dict[str, *Any*]

get_line_data(*data*, *extract='auto'*)

Return a line cut along the radial axis.

Parameters

- **data** (ndarray) – The values at the grid points
- **extract** (*str*) – Determines which cut is done through the grid. This parameter is mainly supplied for a consistent interface and has no effect for polar grids.

Returns

A dictionary with information about the line cut, which is convenient for plotting.

Return type

dict[str, *Any*]

get_random_point(*, *boundary_distance=0*, *avoid_center=False*, *coords='cartesian'*, *rng=None*)

Return a random point within the grid.

Note that these points will be uniformly distributed in the volume, implying they are not uniformly distributed on the radial axis.

Parameters

- **boundary_distance** (*float*) – The minimal distance this point needs to have from all boundaries.
- **avoid_center** (*bool*) – Determines whether the boundary distance should also be kept from the center, i.e., whether points close to the center are returned.
- **coords** (*str*) – Determines the coordinate system in which the point is specified. Valid values are *cartesian*, *cell*, and *grid*; see *transform()*.
- **rng** (Generator) – Random number generator (default: default_rng())

Returns

The coordinates of the point

Return type

ndarray

property has_hole: bool

Returns whether the inner radius is larger than zero.

plot(**args*, *title=None*, *filename=None*, *action='auto'*, *ax_style=None*, *fig_style=None*, *ax=None*, ***kwargs*)

Visualize the spherically symmetric grid in two dimensions.

Parameters

- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically.
- **filename** (*str, optional*) – If given, the plot is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *auto* or *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given.
- **ax_style** (*dict*) – Dictionary with properties that will be changed on the axis after the plot has been drawn by calling `matplotlib.pyplot.setp()`. A special item i this dictionary is *use_offset*, which is flag that can be used to control whether offset are shown along the axes of the plot.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **ax** (`matplotlib.axes.Axes`) – Figure axes to be used for plotting. The special value "create" creates a new figure, while "reuse" attempts to reuse an existing figure, which is the default.
- ****kwargs** – Extra arguments are passed on the to the matplotlib plotting routines, e.g., to set the color of the lines

property radius: float | tuple[float, float]

radius of the sphere

Type
float

property state: dict[str, Any]

the state of the grid

Type
state

property volume: float

total volume of the grid

Type
float

volume_from_radius(*radius*, *dim*)

Return the volume of a sphere with a given radius.

Parameters

- **radius** (float or `ndarray`) – Radius of the sphere
- **dim** (*int*) – Dimension of the space

Returns
Volume of the sphere

Return type
float or ndarray

4.4 pde.pdes package

Package that defines PDEs describing physical systems.

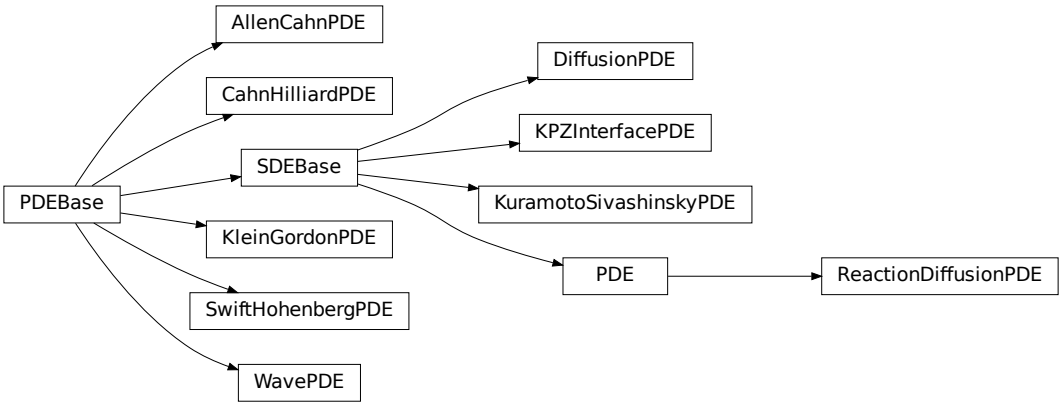
The examples in this package are often simple version of classical PDEs to demonstrate various aspects of the *py-pde* package. Clearly, not all extensions to these PDEs can be covered here, but this should serve as a starting point for custom investigations.

Publicly available methods should take fields with grid information and also only return such methods. There might be corresponding private methods that deal with raw data for faster simulations.

PDEBase	Base class for defining deterministic partial differential equations (PDEs)
SDEBase	Base class for defining stochastic partial differential equations (SDEs)
PDE	PDE defined by mathematical expressions.
AllenCahnPDE	A simple Allen-Cahn equation.
CahnHilliardPDE	A simple Cahn-Hilliard equation.
DiffusionPDE	A simple diffusion equation.
KleinGordonPDE	The Klein-Gordon equation.
KPZInterfacePDE	The Kardar–Parisi–Zhang (KPZ) equation.
KuramotoSivashinskyPDE	The Kuramoto-Sivashinsky equation.
ReactionDiffusionPDE	Reaction-diffusion equation
SwiftHohenbergPDE	The Swift-Hohenberg equation.
WavePDE	A simple wave equation.

Additionally, we offer two solvers for typical elliptical PDEs:

solve_laplace_equation	Solve Laplace's equation on a given grid.
solve_poisson_equation	Solve Poisson's equation on a given grid.
helmholtz_decomposition	Return Helmholtz decomposition of a vector field.



4.4.1 pde.pdes.allen_cahn module

A Allen-Cahn equation.

class AllenCahnPDE(*interface_width=1*, *mobility=1*, *, *bc=None*)

Bases: *PDEBase*

A simple Allen-Cahn equation.

The mathematical definition is

$$\partial_t c = \gamma \nabla^2 c - c^3 + c$$

where c is a scalar field and γ sets the (squared) interfacial width.

Parameters

- **interface_width** (*float*) – The (squared) interfacial width γ of the Allen-Cahn equation
- **mobility** (*float*) – The rate at which the structures evolve
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns
Scalar field describing the evolution rate of the PDE

Return type
ScalarField

explicit_time_dependence: bool | None = False
Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type
bool

property expression: str
the expression of the right hand side of this PDE

Type
str

interface_width: float

make_evolution_rate(*state*, *backend*)
Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.4.2 pde.pdes.base module

Base class for defining partial differential equations.

PDEBase	Base class for defining deterministic partial differential equations (PDEs)
SDEBase	Base class for defining stochastic partial differential equations (SDEs)

NOISE_INTERPRETATIONS: dict[str, float] = {'anti-ito': 1.0, 'anti-itô': 1.0, 'hanggi-klimontovich': 1.0, 'hänggi-klimontovich': 1.0, 'ito': 0.0, 'itô': 0.0, 'stratonovich': 0.5}
dictionary translating noise interpretations to the respective fraction.

Type
dict

class PDEBase(*, *rng=None*)

Bases: object

Base class for defining deterministic partial differential equations (PDEs)

Custom PDEs can be implemented by subclassing *PDEBase* to specify the evolution rate. In the simplest case, only the *PDEBase.evolution_rate()* needs to be implemented to support the *numpy* backend. Other backends require overwriting the *PDEBase.make_evolution_rate()*.

Parameters

rng (Generator) – Random number generator (default: default_rng()) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while other backends might not use it. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

cache_rhs: bool = False

Flag indicating whether the right hand side of the equation should be cached. If True, the same implementation is used in subsequent calls to *solve*. Note that the cache is only invalidated when the grid of the underlying state changes. Consequently, the simulation might lead to wrong results if the parameters of the PDE are changed after the first call. This option is thus disabled by default and should be used with care.

Warning: This flag is deprecated since 2025-12-13 and caching is not implemented anymore.

Type
bool

check_implementation: bool = True

Flag determining whether numba-compiled functions should be checked against their numpy counter-parts. This can help with implementing a correct compiled version for a PDE class.

Warning: This flag is deprecated since 2025-12-13 and this check will not be performed automatically anymore.

Type
bool

check_rhs_consistency(*rhs_implementation*, *state*, *t=0*, *, *tol=1e-07*)

Checks an implementation of the right hand side versus the numpy variant.

Parameters

- **rhs_implementation** (*callable*) – The implementation of the numba variant that is to be checked.
- **state** (FieldBase) – The state for which the evolution rates should be compared
- **t** (*float*) – The associated time point
- **tol** (*float*) – Acceptance tolerance. The check passes if the evolution rates differ by less then this value

Return type
None

complex_valued: bool = False

Flag indicating whether the right hand side is a complex-valued PDE, which requires all involved variables to have complex data type.

Type
bool

determine_backend(*state*, *backend='auto'*, *, *use_mpi=False*)

Returns backend that will be chosen automatically for this PDE.

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted.
- **backend** (*str*) – Information about which backend to choose. The special value *auto* tries various backends and returns one for which the evolution rate is implemented for this PDE.
- **use_mpi** (*bool*) – Request backend with MPI support for parallel simulation.

Returns

The backend used automatically

Return type

str

diagnostics: dict[str, Any]

Diagnostic information (available after the PDE has been solved)

Type

dict

abstractmethod evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*FieldBase*) – The field at the current time point
- **t** (*float*) – The current time point

Returns

Field describing the evolution rate of the PDE

Return type

FieldBase

explicit_time_dependence: bool | None = None

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property is_sde: bool

flag indicating whether this is a stochastic differential equation

Type

bool

make_evolution_rate(*state*, *backend*)

Return function evaluating right hand side of the PDE using given backend.

Note

This factory function must return a function that processes fields stored in the native format of the backend. For instance, a function returned for the *jax* backend must deal with jax.Array objects.

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted.
- **backend** (*str*) – Determines the backend.

Returns
Function determining the right hand side of the PDE

Return type
callable

make_pde_rhs(*state*, *backend='auto'*)

Return a function for evaluating the right hand side of the PDE.

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted.
- **backend** (*str*) – The backend that is used to create the function. The special value *numpy* uses the method *evaluation_rate*. Other backends are only available if *make_evolution_rate* is defined for the PDE. If this is the case, the special value *auto* selects the *numba* backend, otherwise it defaults to *numpy*.

Returns
Function determining the right hand side of the PDE

Return type
callable

make_post_step_hook(*state*, *backend='numpy'*)

Returns a function that is called after each step.

This function receives three arguments: the current state as a numpy array, the current time point, and a numpy array that can store data for the hook function. The function must return the state data and the hook data, which it can both modify in place.

The hook can also be used to abort the simulation when a user-defined condition is met by raising *StopIteration*. Note that this interrupts the inner-most loop, so that some final information might be still reflect the values they assumed at the last tracker interrupt. Additional information (beside the current state) should be returned by the *post_step_data*. Note that raising *StopIteration* only works for some backends.

Example

The following code provides an example that creates a hook function that limits the state to a maximal value of 1 and keeps track of the total correction that is applied. This is achieved using *post_step_data*, which is initialized with the second value (0) returned by the method and incremented each time the hook is called.

```
def make_post_step_hook(self, state, backend):
    def post_step_hook(state_data, t, post_step_data):
        i = state_data > 1  # get violating entries
        overshoot = (state_data[i] - 1).sum()  # get total correction
        state_data[i] = 1  # limit data entries
        post_step_data += overshoot  # accumulate total correction
        return state_data, post_step_data

    return post_step_hook, 0.0  # hook function and initial value
```

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted
- **backend** (*str*) – Determines how the function is created (like 'numpy' and 'numba')

Returns

The first entry is the function that implements the hook. The second entry gives the initial data that is used as auxiliary data in the hook. This can be *None* if no data is used.

Return type
tuple

solve(*state*, *t_range*, *dt=None*, *tracker='auto'*, ***, *backend='auto'*, *solver='euler'*, *ret_info=False*, ***kwargs*)

Solves the partial differential equation.

The method constructs a suitable solver (*SolverBase*) and controller (Controller) to advance the state over the temporal range specified by *t_range*. This method only exposes the most common functions, so explicit construction of these classes might offer more flexibility.

Parameters

- **state** (*FieldBase*) – The initial state (which also defines the spatial grid).
- **t_range** (*float or tuple*) – Sets the time range for which the PDE is solved. This should typically be a tuple of two numbers, *(t_start, t_end)*, specifying the initial and final time of the simulation. If only a single value is given, it is interpreted as *t_end* and the time range is *(0, t_end)*.
- **dt** (*float*) – Time step of the chosen stepping scheme. If *None*, the solver chooses a default value when constructing the stepping function. If adaptive stepping is enabled (e.g., supported by EulerSolver), *dt* sets the initial time step.
- **tracker** (*TrackerCollectionDataType*) – Defines trackers that process the state of the simulation at specified times. A tracker is either an instance of *TrackerBase* or a string identifying a tracker (possible identifiers can be obtained by calling registered_trackers()). Multiple trackers can be specified as a list. The default value *auto* checks the state for consistency (tracker 'consistency') and displays a progress bar (tracker 'progress') when tqdm is installed. More general trackers are defined in *trackers*, where all options are explained in detail. In particular, the time points where the tracker analyzes data can be chosen when creating a tracker object explicitly.
- **backend** (*str*) – Determines how the function is created. Accepted values are 'numpy' and 'numba'. Alternatively, 'auto' lets the code pick the optimal backend.
- **solver** (*SolverBase* or str) – Specifies the persistent numerical strategy used for solving the differential equation. This can either be a solver factory or a descriptive name like 'explicit' or 'scipy'. The valid names are given by pde.solvers.registered_solvers(). Details of the solver classes and additional features (like adaptive stepping) are explained in *solvers*.
- **ret_info** (*bool*) – Flag determining whether diagnostic information about the solver and stepping process should be returned. Note that the same information is also available as the *diagnostics* attribute.
- ****kwargs** – Additional keyword arguments are forwarded to the solver chosen with the *solver* argument. In particular, adaptive stepping can often be enabled using adaptive=True.

Returns

The state at the final time point. If *ret_info* == *True*, a tuple with the final state and a dictionary with additional information is returned. Note that *None* instead of a field is returned in multiprocessing simulations if the current node is not the main MPI node.

Return type

FieldBase

use_noise_realization: bool = False

Flag indicating that a stochastic differential equation should be solved with noise given via `make_noise_realization()`.

Type

bool

use_noise_variance: bool = False

Flag indicating that a stochastic differential equation should be solved with noise given via *make_noise_variance()*.

Type

bool

class SDEBase(*, *noise=0*, *noise_interpretation='ito'*, *rng=None*)

Bases: *PDEBase*

Base class for defining stochastic partial differential equations (SDEs)

Custom PDEs can be implemented by subclassing *SDEBase* to specify the evolution rate and an associated noise variance (or realization). Overwrite *make_noise_variance()* (together with *PDEBase.make_evolution_rate()*) to support all backends.

Parameters

- **noise** (float or `ndarray`) – Variance of the additive Gaussian white noise that is supported for all PDEs by default. If set to zero, a deterministic partial differential equation will be solved. Different noise magnitudes can be supplied for each field in coupled PDEs.
- **noise_interpretation** (*str*) – Interpretation of the stochastic differential equation. Possible values are *ito*, *stratonovich*, and *anti-ito*. Solvers can use this information to implement drift terms that appear for multiplicative noise, which typically only works when *make_noise_variance()* also returns the derivative of the variance.
- **rng** (`Generator`) – Random number generator (default: `default_rng()`) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while other backends might not use it. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

Note

If more complicated noise structures are required, overwrite *SDEBase.make_noise_variance()* to provide a custom noise variance for all backends.

property is_sde: bool

flag indicating whether this is a stochastic differential equation

The *SDEBase* class supports additive Gaussian white noise, whose magnitude is controlled by the *noise* property. In this case, *is_sde* is *True* if *self.noise != 0*. Sub-classes might need to set *is_sde* explicitly to signal that they define a stochastic partial differential equation.

Type
bool

make_noise_variance(*state: TField*, *, *backend:* BackendBase*[TNativeArray]*) → Callable[[TNativeArray, float], TNativeArray]

make_noise_variance(*state: TField*, *, *backend:* BackendBase*[TNativeArray]*, *ret_diff: Literal[False]*) → Callable[[TNativeArray, float], TNativeArray]

make_noise_variance(*state: TField*, *, *backend:* BackendBase*[TNativeArray]*, *ret_diff: Literal[True]*) → Callable[[TNativeArray, float], tuple[TNativeArray, TNativeArray]]

Make function that calculates noise variance.

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted.
- **backend** (*str*) – Determines the backend.
- **ret_diff** (*bool*) – Determines whether only the noise variance or also its derivative with respect to the field at this position is returned.

Returns
A function with signature (state_data, t) that either returns just the noise variance or also its derivative, depending on *ret_diff*.

Return type
Callable[[TNativeArray, float], TNativeArray | tuple[TNativeArray, TNativeArray]]

use_noise_realization: bool = False

Flag indicating that a stochastic differential equation should be solved with noise given via `make_noise_realization()`.

Type
bool

use_noise_variance: bool = True

Flag indicating that a stochastic differential equation should be solved with noise given via *make_noise_variance()*.

Type
bool

expr_prod(*factor*, *expression*)

Helper function for building an expression with an (optional) pre-factor.

Parameters

- **factor** (*float*) – The value of the prefactor
- **expression** (*str*) – The remaining expression

Returns
The expression with the factor appended if necessary

Return type
str

4.4.3 pde.pdes.cahn_hilliard module

A Cahn-Hilliard equation.

class CahnHilliardPDE(*interface_width=1*, ***, *bc_c=None*, *bc_mu=None*)

Bases: *PDEBase*

A simple Cahn-Hilliard equation.

The mathematical definition is

$$\partial_t c = \nabla^2 \left(c^3 - c - \gamma \nabla^2 c\right)$$

where c is a scalar field taking values on the interval $[-1, 1]$ and γ sets the (squared) interfacial width.

Parameters

- **interface_width** (*float*) – The width of the interface between the separated phases. This defines a characteristic length in the simulation. The grid needs to resolve this length of a stable simulation.
- **bc_c** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_mu** (*BoundariesData | None*) – The boundary conditions applied to the chemical potential associated with the scalar field c. Supports the same options as *bc_c*.

default_bc_c = 'auto_periodic_neumann'

Default boundary condition for order parameter.

default_bc_mu = 'auto_periodic_neumann'

Default boundary condition for chemical potential.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns

Scalar field describing the evolution rate of the PDE

Return type

ScalarField

explicit_time_dependence = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expression: str

the expression of the right hand side of this PDE

Type
str

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.4.4 pde.pdes.diffusion module

A simple diffusion equation.

class DiffusionPDE(*diffusivity=1*, *, *bc=None*, *noise=0*, *rng=None*)

Bases: *SDEBase*

A simple diffusion equation.

The mathematical definition is

$$\partial_t c = D \nabla^2 c$$

where c is a scalar field and D denotes the diffusivity.

Parameters

- **diffusivity** (*float*) – The diffusivity of the described species
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **noise** (*float*) – Variance of the (additive) noise term
- **rng** (*Generator*) – Random number generator (default: *default_rng()*) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while compiled numba code uses the random number generator of numba. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns

Scalar field describing the evolution rate of the PDE

Return type

ScalarField

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expression: str

the expression of the right hand side of this PDE

Type

str

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns

A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type

Callable[[TNativeArray, float], TNativeArray]

4.4.5 pde.pdes.klein_gordon module

The Klein-Gordon equation.

class KleinGordonPDE(*speed=1*, *mass=1*, *, *bc=None*)

Bases: *PDEBase*

The Klein-Gordon equation.

The mathematical definition, $\partial_t^2 u = c^2 \nabla^2 u - \mu^2 u$, is implemented as two first-order equations,

$$\partial_t u = v$$
$$\partial_t v = c^2 \nabla^2 u - \mu^2 u$$

where c sets the wave speed, μ is the mass parameter, and v is an auxiliary field. Note that the class expects an initial condition specifying both fields, which can be created using the *KleinGordonPDE.get_initial_condition()* method. The result will also return two fields.

The Klein-Gordon equation describes relativistic scalar fields and reduces to the standard wave equation when $\mu = 0$.

Parameters

- **speed** (*float*) – The speed c of the wave
- **mass** (*float*) – The mass parameter μ
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field u. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like x- and y+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*FieldCollection*) – The fields u and v
- **t** (*float*) – The current time point

Returns

Fields describing the evolution rates of the PDE

Return type

FieldCollection

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expressions: dict[str, str]

the expressions of the right hand side of this PDE

Type

dict

get_initial_condition(*u*, *v=None*)

Create a suitable initial condition.

Parameters

- **u** (*ScalarField*) – The initial field amplitude on the grid
- **v** (*ScalarField*, optional) – The initial rate of change. This is assumed to be zero if the value is omitted.

Returns

The combined fields u and v, suitable for the simulation

Return type
FieldCollection

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.4.6 pde.pdes.kpz_interface module

The Kardar–Parisi–Zhang (KPZ) equation describing the evolution of an interface.

class KPZInterfacePDE(*nu=0.5*, *lmbda=1*, *, *bc=None*, *noise=0*, *rng=None*)

Bases: *SDEBase*

The Kardar–Parisi–Zhang (KPZ) equation.

The mathematical definition is

$$\partial_t h = \nu \nabla^2 h + \frac{\lambda}{2} (\nabla h)^2 + \eta(\boldsymbol{r}, t)$$

where h is the height of the interface in Monge parameterization. The dynamics are governed by the two parameters ν and λ, while η is Gaussian white noise, whose strength is controlled by the *noise* argument.

Parameters

- **nu** (*float*) – Parameter ν for the strength of the diffusive term
- **lmbda** (*float*) – Parameter λ for the strength of the gradient term
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **noise** (*float*) – Variance of the (additive) noise term
- **rng** (*Generator*) – Random number generator (default: `default_rng()`) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while compiled numba code uses the random number generator of numba. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns

Scalar field describing the evolution rate of the PDE

Return type

ScalarField

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expression: str

the expression of the right hand side of this PDE

Type

str

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns

A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type

Callable[[TNativeArray, float], TNativeArray]

4.4.7 pde.pdes.kuramoto_sivashinsky module

The Kuramoto-Sivashinsky equation.

class KuramotoSivashinskyPDE(*nu=1*, ***, *bc=None*, *bc_lap=None*, *noise=0*, *rng=None*)

Bases: *SDEBase*

The Kuramoto-Sivashinsky equation.

The mathematical definition is

$$\partial_t u = -\nu \nabla^4 u - \nabla^2 u - \frac{1}{2} (\nabla u)^2 + \eta(\boldsymbol{r}, t)$$

where u is the height of the interface in Monge parameterization. The dynamics are governed by the parameter ν, while η is Gaussian white noise, whose strength is controlled by the *noise* argument.

Parameters

- **nu** (*float*) – Parameter ν for the strength of the fourth-order term
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_lap** (*BoundariesData | None*) – The boundary conditions applied to the second derivative of the scalar field *c*. If *None*, the same boundary condition as *bc* is chosen. Otherwise, this supports the same options as *bc*.
- **noise** (*float*) – Variance of the (additive) noise term
- **rng** (*Generator*) – Random number generator (default: `default_rng()`) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while compiled numba code uses the random number generator of numba. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns

Scalar field describing the evolution rate of the PDE

Return type

ScalarField

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expression: str

the expression of the right hand side of this PDE

Type

str

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types

- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.4.8 pde.pdes.laplace module

Solvers for Poisson's and Laplace's equation.

solve_poisson_equation	Solve Poisson's equation on a given grid.
solve_laplace_equation	Solve Laplace's equation on a given grid.
helmholtz_decomposition	Return Helmholtz decomposition of a vector field.

helmholtz_decomposition(*field*, *bc*)

Return Helmholtz decomposition of a vector field.

For a vector field $\boldsymbol{u}$, we return a scalar potential ϕ and a solenoidal (divergence-free) vector field $\boldsymbol{v}$, which obey

$$\boldsymbol{u} = \nabla\phi + \boldsymbol{v}$$

Parameters

- **field** (*VectorField*) – The vector field u that needs to be decomposed.
- **bc** (*BoundariesData*) – The boundary conditions applied to the field. Note that the same boundary conditions are also applied to when solving the Poisson equation to determine the potential. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti- periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

Returns
The two fields of the Helmholtz decomposition

Return type
ScalarField, *VectorField*

solve_laplace_equation(*grid*, *bc*, *, *backend='scipy'*, *label="Solution to Laplace's equation"*)

Solve Laplace's equation on a given grid.

Parameters

- **grid** (*GridBase*) – The grid on which the equation is solved
- **bc** (*BoundariesData*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti- periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a

value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

- **backend** (*str*) – The name of the backend to use to implement this operator.
- **label** (*str*) – The label of the returned field.

Returns
The field that solves the equation.

Return type
ScalarField

solve_poisson_equation(*rhs*, *bc*, *, *backend='scipy'*, *label="Solution to Poisson's equation"*, ***kwargs*)

Solve Poisson's equation on a given grid.

Denoting the current field by u, we thus solve for f, defined by the equation

$$\nabla^2 u(\boldsymbol{r}) = -f(\boldsymbol{r})$$

with boundary conditions specified by *bc*.

Note

In case of periodic or Neumann boundary conditions, the right hand side $f(\boldsymbol{r})$ needs to satisfy the following condition

$$\int f \,\mathrm{d}V = \oint g \,\mathrm{d}S \;,$$

where g denotes the function specifying the outwards derivative for Neumann conditions. Note that for periodic boundaries g vanishes, so that this condition implies that the integral over f must vanish for neutral Neumann or periodic conditions.

Parameters

- **rhs** (*ScalarField*) – The scalar field f describing the right hand side
- **bc** (*BoundariesData*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti- periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **backend** (*str*) – The name of the backend to use to implement this operator.
- **label** (*str*) – The label of the returned field.
- ****kwargs** – Additional parameters influence how the Laplace operator is constructed.

Returns
Field u solving the equation.

Return type
ScalarField

4.4.9 pde.pdes.pde module

Defines a PDE class whose right hand side is given as a string.

class PDE(*rhs*, ***, *bc=None*, *bc_ops=None*, *post_step_hook=None*, *user_funcs=None*, *consts=None*, *noise=0*, *noise_interpretation='ito'*, *rng=None*)

Bases: *SDEBase*

PDE defined by mathematical expressions.

variables

The name of the variables (i.e., fields) in the order they are expected to appear in the *state*.

Type
tuple

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **rhs** (*dict*) – The expressions defining the evolution rate. The dictionary keys define the name of the fields whose evolution is considered, while the values specify their evolution rate as a string that can be parsed by `sympy`. These expression may contain variables (i.e., the fields themselves, spatial coordinates of the grid, and *t* for the time), standard local mathematical operators defined by sympy, and the operators defined in the *pde* package. Note that operators need to be specified with their full name, i.e., *laplace* for a scalar Laplacian and *vector_laplace* for a Laplacian operating on a vector field. Moreover, the dot product between two vector fields can be denoted by using *dot(field1, field2)* in the expression, an outer product is calculated using *outer(field1, field2)*, and *integral(field)* denotes an integral over a field. More information can be found in the *expression documentation*.
- **bc** (*BoundariesData | None*) – General boundary conditions for all operators that do not have a specialized condition given in *bc_ops*. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_ops** (*dict*) – Special boundary conditions for specific operators. The keys in this dictionary specify where the boundary condition will be applied. The keys follow the format "VARIABLE:OPERATOR", where VARIABLE specifies the expression in *rhs* where the boundary condition is applied to the operator specified by OPERATOR. For both identifiers, the wildcard symbol "*" denotes that all fields and operators are affected, respectively. For instance, the identifier "c:*" allows specifying a condition for all operators of the field named *c*.

- **post_step_hook** (*callable*) – A function with signature *(state_data, t)* that will be called after every time step. The function must return *state_data*, which can be modified in place. The hook can also abort the simulation immediately by raising *StopIteration* (might not work with all backends).
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expressions in *rhs*.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. These can be either scalar numbers or fields defined on the same grid as the actual simulation.
- **noise** (float, ndarray, or dict) – Variance of additive Gaussian white noise. The default value of zero implies deterministic partial differential equations will be solved. Different noise magnitudes can be supplied for each field in coupled PDEs by either specifying a sequence of numbers or a dictionary with values for each field.
- **noise_interpretation** (*str*) – Interpretation of the stochastic differential equation. Possible values are *ito*, *stratonovich*, and *anti-ito*. Solvers can use this information to implement drift terms that appear for multiplicative noise, which typically only works when *make_noise_variance()* also returns the derivative of the variance.
- **rng** (Generator) – Random number generator (default: default_rng()) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while compiled numba code uses the random number generator of numba. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

Note

The order in which the fields are given in *rhs* defines the order in which they need to appear in the *state* variable when the evolution rate is calculated. Note that *dict* keep the insertion order since Python version 3.7, so a normal dictionary can be used to define the equations.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0.0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (FieldBase) – The field describing the state of the PDE
- **t** (*float*) – The current time point

Returns

Field describing the evolution rate of the PDE

Return type

FieldBase

property expressions: dict[str, str]

Show the expressions of the PDE.

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

make_post_step_hook(*state*, *backend='numpy'*)

Returns a function that is called after each step.

Parameters

- **state** (FieldBase) – An example for the state from which the grid and other information can be extracted
- **backend** (*str*) – Determines how the function is created (like 'numpy' and 'numba')

Returns

The first entry is the function that implements the hook. The second
entry gives the initial data that is used as auxiliary data in the hook. This can be *None* if no data is used.

Return type
tuple

Raises
NotImplementedError – When post_step_hook is *None*.

4.4.10 pde.pdes.reaction_diffusion module

Defines a PDE class implementing a reaction-diffusion system.

class ReactionDiffusionPDE(*variables*, *diffusivity*, *sources*, *, *bc=None*, *bc_ops=None*, *post_step_hook=None*, *user_funcs=None*, *consts=None*, *noise=0*, *rng=None*)

Bases: *PDE*

Reaction-diffusion equation

The equation being solved reads

$$\partial_t c_i = D_i \partial_\alpha^2 c_i + s_i(\{c_j\}, t)$$

where *c_i* are the concentration fields, D_i are the diffusivities, and s_i are sink/source terms that account for chemical reactions.

Parameters

- **variables** (*list of strings*) – The names and order of the variables c_i in the system
- **diffusivity** (ndarray) – Diffusivities D_i of all species. A scalar sets the same diffusivity for all species.
- **sources** (*list of str or dict of str*) – Specifies the source terms s_i of each species. Must be a list with an entry for each variable. Alternatively, a dictionary may be used to only specify the source term of a few variables, while all others are assumed to not have any sources.

- **bc** (*BoundariesData | None*) – General boundary conditions for all operators that do not have a specialized condition given in *bc_ops*. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_ops** (*dict*) – Special boundary conditions for specific operators. The keys in this dictionary specify where the boundary condition will be applied. The keys follow the format "VARIABLE:OPERATOR", where VARIABLE specifies the expression in *rhs* where the boundary condition is applied to the operator specified by OPERATOR. For both identifiers, the wildcard symbol "*" denotes that all fields and operators are affected, respectively. For instance, the identifier "c:*" allows specifying a condition for all operators of the field named *c*.
- **post_step_hook** (*callable*) – A function with signature *(state_data, t)* that will be called after every time step. The function must return *state_data*, which can be modified in place. The hook can also abort the simulation immediately by raising *StopIteration* (might not work with all backends).
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expressions in *rhs*.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. These can be either scalar numbers or fields defined on the same grid as the actual simulation.
- **noise** (float, ndarray, or dict) – Variance of additive Gaussian white noise. The default value of zero implies deterministic partial differential equations will be solved. Different noise magnitudes can be supplied for each field in coupled PDEs by either specifying a sequence of numbers or a dictionary with values for each field.
- **rng** (Generator) – Random number generator (default: default_rng()) used for stochastic simulations. Note that this random number generator is only used for numpy functions, while compiled numba code uses the random number generator of numba. Moreover, in simulations using multiprocessing, setting the same generator in all processes might yield unintended correlations in the simulation results.

4.4.11 pde.pdes.swift_hohenberg module

The Swift-Hohenberg equation.

class SwiftHohenbergPDE(*rate=0.1*, *kc2=1.0*, *delta=1.0*, ***, *bc=None*, *bc_lap=None*)

Bases: *PDEBase*

The Swift-Hohenberg equation.

The mathematical definition is

$$\partial_t c = \left[\epsilon - \left(k_c^2 + \nabla^2\right)^2\right] c + \delta\, c^2 - c^3$$

where c is a scalar field and ϵ, k_c^2, and δ are parameters of the equation.

Parameters

- **rate** (*float*) – The bifurcation parameter ϵ

- **kc2** (*float*) – Squared wave vector k_c^2 of the linear instability
- **delta** (*float*) – Parameter δ of the non-linearity
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_lap** (*BoundariesData | None*) – The boundary conditions applied to the second derivative of the scalar field *c*. If *None*, the same boundary condition as *bc* is chosen. Otherwise, this supports the same options as *bc*.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point

Returns

Scalar field describing the evolution rate of the PDE

Return type

ScalarField

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type

bool

property expression: str

the expression of the right hand side of this PDE

Type

str

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns

A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.4.12 pde.pdes.wave module

A simple wave equation.

class WavePDE(*speed=1*, *, *bc=None*)

Bases: *PDEBase*

A simple wave equation.

The mathematical definition, $\partial_t^2 u = c^2 \nabla^2 u$, is implemented as two first-order equations,

$$\partial_t u = v$$
$$\partial_t v = c^2 \nabla^2 u$$

where c sets the wave speed and v is an auxiliary field. Note that the class expects an initial condition specifying both fields, which can be created using the *WavePDE.get_initial_condition()* method. The result will also return two fields.

Parameters

- **speed** (*float*) – The speed c of the wave
- **bc** (*BoundariesData | None*) – The boundary conditions applied to the field u. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti-periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x-* and *y+*). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

default_bc = 'auto_periodic_neumann'

Default boundary condition used when no specific conditions are chosen.

evolution_rate(*state*, *t=0*)

Evaluate the right hand side of the PDE.

Parameters

- **state** (*FieldCollection*) – The fields u and v
- **t** (*float*) – The current time point

Returns
Fields describing the evolution rates of the PDE

Return type
FieldCollection

explicit_time_dependence: bool | None = False

Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type
bool

property expressions: dict[str, str]

the expressions of the right hand side of this PDE

Type
dict

get_initial_condition(*u*, *v=None*)

Create a suitable initial condition.

Parameters

- **u** (*ScalarField*) – The initial density on the grid
- **v** (*ScalarField*, optional) – The initial rate of change. This is assumed to be zero if the value is omitted.

Returns
The combined fields u and v, suitable for the simulation

Return type
FieldCollection

make_evolution_rate(*state*, *backend*)

Create a compiled function evaluating the right hand side of the PDE.

Parameters

- **state** (*ScalarField*) – An example for the state defining the grid and data types
- **backend** (str or *BackendBase*) – The backend used for numerical operations

Returns
A function with signature *(state_data, t)*, which can be called with an instance of the state data and time to obtain the associated evolution rate.

Return type
Callable[[TNativeArray, float], TNativeArray]

4.5 pde.solvers package

Solver classes define the strategy for evolving PDE states in time.

A solver object stores the numerical method and configuration and is used to construct an executable stepping function that advances the current state.

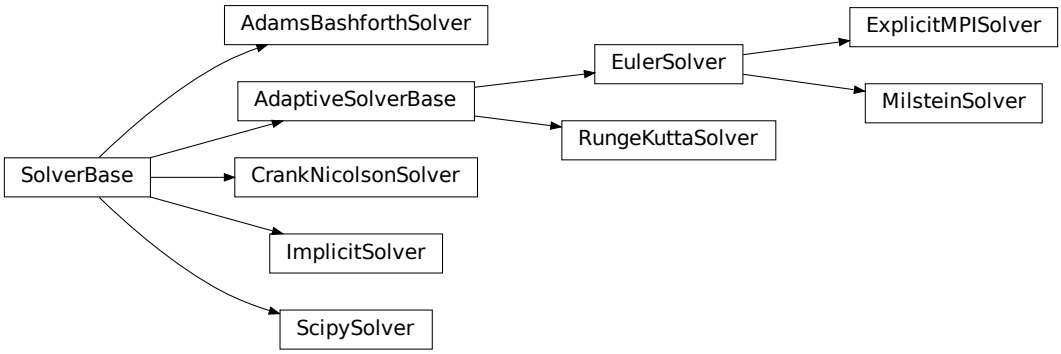
AdamsBashforthSolver	Explicit Adams-Bashforth multi-step solver.
AdaptiveSolverBase	Base class for solvers that can produce adaptive stepping functions.
SolverBase	Base class for persistent PDE solver strategy objects.
Controller	Class controlling a simulation.
CrankNicolsonSolver	Crank-Nicolson solver.
EulerSolver	Explicit Euler solver.
ExplicitMPISolver	Explicit Euler solver using MPI.
ImplicitSolver	Implicit (backward) Euler PDE solver.
MilsteinSolver	Milstein method for stochastic differential equations.
RungeKuttaSolver	Explicit Runge-Kutta PDE solver of order 5(4).
ScipySolver	PDE solver using `scipy.integrate.solve_ivp()`.

continues on next page

Table 48 – continued from previous page

registered_solvers	Returns all solvers that are currently registered.

Inheritance structure of the classes:



4.5.1 pde.solvers.adams_bashforth module

Defines an explicit Adams-Bashforth solver.

class AdamsBashforthSolver(*pde*, *, *backend='auto'*)

Bases: *SolverBase*

Explicit Adams-Bashforth multi-step solver.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (str or *BackendBase*) – The backend used for numerical operations

name = 'adams-bashforth'

4.5.2 pde.solvers.base module

Package that contains base classes for solvers.

Beside the abstract base class *SolverBase* defining the interfaces, we also provide *AdaptiveSolverBase*, which contains methods for adaptive solvers.

SolverBase	Base class for persistent PDE solver strategy objects.
AdaptiveSolverBase	Base class for solvers that can produce adaptive stepping functions.
ConvergenceError	Indicates that an implicit step did not converge.

class AdaptiveSolverBase(*pde*, *, *backend='auto'*, *adaptive=False*, *tolerance=0.0001*)

Bases: *SolverBase*

Base class for solvers that can produce adaptive stepping functions.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations
- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping

dt_max: float = 10000000000.0

maximal time step that the adaptive solver will use

Type
float

dt_min: float = 1e-10

minimal time step that the adaptive solver will use

Type
float

make_stepper(*state*, *dt=None*)

Create the executable stepping function produced by this solver.

Parameters

- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **dt** (*float*) – Time step used (Uses *SolverBase.dt_default* if *None*). This sets the initial time step for adaptive solvers.

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type
StepperType

exception ConvergenceError

Bases: RuntimeError

Indicates that an implicit step did not converge.

class SolverBase(*pde*, ***, *backend='auto'*)

Bases: object

Base class for persistent PDE solver strategy objects.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (str or *BackendBase*) – The backend used for numerical operations

property backend: *BackendBase*

The backend for this solver.

Type
BackendBase

property backend_name: str

The name of the backend used for this solver.

Type
str

dt_default: float = 0.001

default time step used when no initial value was specified

Type
float

classmethod from_name(*name*, *pde*, ***kwargs*)

Create a solver object from its registered name.

Solver classes are automatically registered when they inherit from *SolverBase*. Note that this also requires that the respective python module containing the solver has been loaded before it is attempted to be used.

Parameters

- **name** (*str*) – The name of the solver to construct
- **pde** (*PDEBase*) – The partial differential equation that should be solved
- ****kwargs** – Additional arguments for the constructor of the solver

Returns
An instance of a subclass of *SolverBase*

Return type
SolverBase

info: dict[str, Any]

make_stepper(*state*, *dt=None*)

Create the executable stepping function produced by this solver.

Parameters

- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **dt** (*float*) – Initial time step used (Uses *SolverBase.dt_default* if *None*)

Returns
Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type
StepperType

mpi_run: bool = False

Whether this solver runs using MPI

Type
bool

registered_solvers()

Returns all solvers that are currently registered.

Returns
a dictionary with the names of the solvers and the associated class

Return type
dict

4.5.3 pde.solvers.controller module

Defines a class controlling the simulations of PDEs.

class Controller(*solver*, *t_range*, *tracker='auto'*)

Bases: object

Class controlling a simulation.

The controller calls a solver to advance the simulation into the future and it takes care of trackers that analyze and modify the state periodically. The controller also handles errors in the simulations and the trackers, as well as user-induced interrupts, e.g., by hitting Ctrl-C or Cmd-C to cause a KeyboardInterrupt. In case of problems, the Controller writes additional information into *diagnostics*, which can help to diagnose problems.

Parameters

- **solver** (*SolverBase*) – Solver instance that is used to advance the simulation in time
- **t_range** (*float or tuple*) – Sets the time range for which the simulation is run. If only a single value *t_end* is given, the time range is assumed to be *[0, t_end]*.
- **tracker** (*TrackerCollectionDataType*) – Defines trackers that process the state of the simulation at specified times. A tracker is either an instance of *TrackerBase* or a string identifying a tracker (possible identifiers can be obtained by calling registered_trackers()). Multiple trackers can be specified as a list. The default value *auto* checks the state for consistency (tracker 'consistency') and displays a progress bar (tracker 'progress') when tqdm is installed. More general trackers are defined in *trackers*, where all options are explained in detail. In particular, the time points where the tracker analyzes data can be chosen when creating a tracker object explicitly.

diagnostics: dict[str, Any]

diagnostic information (available after simulation finished)

Type
dict

run(*initial_state*, *dt=None*)

Run the simulation.

Diagnostic information about the solver are available in the *diagnostics* property after this function has been called.

Parameters

- **initial_state** (*FieldBase*) – The initial state of the simulation. This state will be copied and thus not modified by the simulation. Instead, the final state will be returned and trackers can be used to record intermediate states.
- **dt** (*float*) – Initial time step of the chosen stepping scheme. If *None*, a default value based on the solver configuration will be chosen.

Returns
The state at the final time point. If multiprocessing is used, only the main node will return the state. All other nodes return None.

Return type
TField | None

property t_range: tuple[float, float]

start and end time of the simulation

Type

tuple

4.5.4 pde.solvers.crank_nicolson module

Defines a Crank-Nicolson solver.

class CrankNicolsonSolver(*pde*, *, *maxiter=100*, *maxerror=0.0001*, *explicit_fraction=0*, *backend='auto'*)

Bases: *SolverBase*

Crank-Nicolson solver.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **maxiter** (*int*) – Maximum number of iterations for the implicit solver
- **maxerror** (*float*) – Maximum error tolerance for the implicit solver
- **explicit_fraction** (*float*) – Fraction of explicit time stepping (0 for fully implicit)
- **backend** (*str*) – The backend used for numerical operations

name = 'crank-nicolson'

4.5.5 pde.solvers.euler module

Defines an explicit solver for the Euler scheme.

EulerSolver	Explicit Euler solver.

class EulerSolver(*pde*, *, *backend='auto'*, *adaptive=False*, *tolerance=0.0001*)

Bases: *AdaptiveSolverBase*

Explicit Euler solver.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations
- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping

name = 'euler'

class ExplicitSolver(*pde*, *scheme='euler'*, ***kwargs*)

Bases: *AdaptiveSolverBase*

Various explicit PDE solvers.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations

- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping
- **scheme** (*Literal['euler', 'runge-kutta', 'rk', 'rk45']*)

name = 'explicit'

4.5.6 pde.solvers.explicit_mpi module

Defines an explicit solver using multiprocessing via MPI.

TODO: Implement this not as a separate solver but as a separate numba_mpi backend

class ExplicitMPISolver(*pde*, *decomposition='auto'*, ***, *backend='auto'*, *adaptive=False*, *tolerance=0.0001*)

Bases: *EulerSolver*

Explicit Euler solver using MPI.

Warning

This solver can only be used if MPI is properly installed. In particular, python scripts then need to be started using `mpirun` or `mpiexec`. Please refer to the documentation of your MPI distribution for details.

The main idea of the solver is to take the full initial state in the main node (ID 0) and split the grid into roughly equal subgrids. The main node then distributes these subfields to all other nodes and each node creates the right hand side of the PDE for itself (and independently). Each node then advances the PDE independently, ensuring proper coupling to neighboring nodes via special boundary conditions, which exchange field values between sub grids. This is implemented by the `get_boundary_conditions()` method of the sub grids, which takes the boundary conditions for the full grid and creates conditions suitable for the specific sub grid on the given node. The trackers (and thus all input and output) are only handled on the main node.

Warning

The function providing the right hand side of the PDE needs to support MPI. This is automatically the case for local evaluations (which only use the field value at the current position), for the differential operators provided by *pde*, and integration of fields. Similarly, *post_step_hook* can only be used to do local modifications since the field data supplied to the function is local to each MPI node.

Example

A minimal example using the MPI solver is

```
from pde import DiffusionPDE, pde.fields.scalar.ScalarField, UnitGrid

grid = UnitGrid([64, 64])
state = pde.fields.scalar.ScalarField.random_uniform(grid, 0.2, 0.3)

eq = DiffusionPDE(diffusivity=0.1)
result = eq.solve(state, t_range=10, dt=0.1, solver="explicit_mpi")

if result is not None:  # restrict the output to the main node
    result.plot()
```

Saving this script as *multiprocessing.py*, a parallel simulation is started by

```
mpiexec -n 2 python3 multiprocessing.py
```

Here, the number *2* determines the number of cores that will be used. Note that macOS might require an additional hint on how to connect the processes even when they are run on the same machine (e.g., your workstation). It might help to run `mpiexec -n 2 -host localhost python3 multiprocessing.py` in this case

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **decomposition** (*str, int, or list of ints*) – Number of subdivision in each direction. Should be a list of length *grid.num_axes* specifying the number of nodes for this axis. If one value is *-1*, its value will be determined from the number of available nodes. A single integer is interpreted as the number of subdivisions along one axis. The default value *auto* tries to determine an optimal decomposition by minimizing communication between nodes.
- **backend** (*str*) – The backend used for numerical operations
- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping

make_stepper(*state*, *dt=None*)

Create the executable stepping function produced by this solver.

Parameters

- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **dt** (*float*) – Initial time step. If *None*, this solver specifies 1e-3 as a default value.

Returns

Function that can be called to advance the *state* from time *t_start* to time *t_end*. The function call signature is *(state: numpy.ndarray, t_start: float, t_end: float)*

Return type

StepperType

mpi_run = True

Whether this solver runs using MPI

Type

bool

name = 'explicit_mpi'

4.5.7 pde.solvers.implicit module

Defines an implicit Euler solver.

class ImplicitSolver(*pde*, *, *maxiter=100*, *maxerror=0.0001*, *backend='auto'*)

Bases: *SolverBase*

Implicit (backward) Euler PDE solver.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **maxiter** (*int*) – Maximum number of iterations for the implicit solver
- **maxerror** (*float*) – Maximum error tolerance for the implicit solver
- **backend** (*str*) – The backend used for numerical operations

name = 'implicit'

4.5.8 pde.solvers.milstein module

Defines an explicit Milstein solver for stochastic differential equations.

MilsteinSolver	Milstein method for stochastic differential equations.

class MilsteinSolver(*pde*, *, *backend='auto'*, *adaptive=False*, *tolerance=0.0001*)

Bases: *EulerSolver*

Milstein method for stochastic differential equations.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations
- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping

name = 'milstein'

4.5.9 pde.solvers.runge_kutta module

Defines an explicit solver using a Runge-Kutta method.

RungeKuttaSolver	Explicit Runge-Kutta PDE solver of order 5(4).

class RungeKuttaSolver(*pde*, *, *backend='auto'*, *adaptive=False*, *tolerance=0.0001*)

Bases: *AdaptiveSolverBase*

Explicit Runge-Kutta PDE solver of order 5(4).

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations
- **adaptive** (*bool*) – Whether to use adaptive time stepping
- **tolerance** (*float*) – Error tolerance for adaptive time stepping

name = 'runge-kutta'

4.5.10 pde.solvers.scipy module

Defines a solver using `scipy.integrate`

class ScipySolver(*pde*, *, *backend='auto'*, ***kwargs*)

Bases: *SolverBase*

PDE solver using `scipy.integrate.solve_ivp()`.

This class is a thin wrapper around `scipy.integrate.solve_ivp()`. In particular, it supports all the methods implemented by this function and exposes its arguments, so details can be controlled.

Parameters

- **pde** (*PDEBase*) – The partial differential equation that should be solved
- **backend** (*str*) – The backend used for numerical operations
- ****kwargs** – All extra arguments are forwarded to `scipy.integrate.solve_ivp()`.

make_stepper(*state*, *dt=None*)

Create the executable stepping function produced by this solver.

Parameters

- **state** (`FieldBase`) – An example for the state from which the grid and other information can be extracted.
- **dt** (*float*) – Initial time step for the simulation. If *None*, the solver will choose a suitable initial value.

Returns

Function that can be called to advance the *state* from time *t_start* to time *t_end*.

Return type

StepperType

name = 'scipy'

exception ScipySolverError

Bases: `RuntimeError`

4.6 pde.storage package

Module defining classes for storing simulation data.

StorageBase	Base class for storing time series of discretized fields.
FileStorage	Store discretized fields in a hdf5 file.
get_memory_storage	A context manager that can be used to create a MemoryStorage.
MemoryStorage	Store discretized fields in memory.
ModelrunnerStorage	Store discretized fields in a `modelrunner` storage.
MovieStorage	Store discretized fields in a movie file.

class FileStorage(*filename*, *, *info=None*, *write_mode='truncate_once'*, *max_length=None*, *compression=True*, *keep_opened=True*, *check_mpi=True*)

Bases: *StorageBase*

Store discretized fields in a hdf5 file.

Parameters

- **filename** (*str*) – The path to the hdf5-file where the data is stored
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.
- **max_length** (*int, optional*) – Maximal number of entries that will be stored in the file. This can be used to preallocate data, which can lead to smaller files, but is also less flexible. Giving *max_length* = *None*, allows for arbitrarily large data, which might lead to larger files.
- **compression** (*bool*) – Whether to store the data in compressed form. Automatically enabled chunked storage.
- **keep_opened** (*bool*) – Flag indicating whether the file should be kept opened after each writing. If *False*, the file will be closed after writing a dataset. This keeps the file in a consistent state, but also requires more work before data can be written.
- **check_mpi** (*bool*) – If True, files will only be opened in the main node for an parallel simulation using MPI. This flag has no effect in serial code.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

close()

Close the currently opened file.

Return type

None

property data

The actual data for all time.

Type

ndarray

end_writing()

Finalize the storage after writing.

This makes sure the data is actually written to a file when self.keep_opened == False

Return type

None

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type

None

property times

The times at which data is available.

Type
ndarray

class MemoryStorage(*times=None*, *data=None*, ***, *info=None*, *field_obj=None*, *write_mode='truncate_once'*)

Bases: *StorageBase*

Store discretized fields in memory.

Parameters

- **times** (ndarray) – Sequence of times for which data is known
- **data** (list of ndarray) – The field data at the given times
- **field_obj** (*FieldBase*) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

Return type
None

classmethod from_collection(*storages*, *label=None*, ***, *rtol=1e-05*, *atol=1e-08*)

Combine multiple memory storages into one.

This method can be used to combine multiple time series of different fields into a single representation. This requires that all time series contain data at the same time points.

Parameters

- **storages** (*list*) – A collection of instances of *StorageBase* whose data will be concatenated into a single MemoryStorage
- **label** (*str, optional*) – The label of the instances of *FieldCollection* that represent the concatenated data
- **rtol** (*float*) – Relative tolerance used when checking times for merging
- **atol** (*float*) – Absolute tolerance used when checking times for merging

Returns
Storage containing all the data.

Return type
MemoryStorage

classmethod from_fields(*times=None*, *fields=None*, *info=None*, *write_mode='truncate_once'*)

Create MemoryStorage from a list of fields.

Parameters

- **times** (ndarray) – Sequence of times for which data is known

- **fields** (list of FieldBase) – The fields at all given time points
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

Return type
MemoryStorage

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type
None

class ModelrunnerStorage(*storage*, *, *loc='trajectory'*, *info=None*, *write_mode='truncate_once'*)

Bases: *StorageBase*

Store discretized fields in a modelrunner storage.

This storage class acts as a wrapper for the trajectory module, which allows handling time-dependent data in modelrunner storages. In principle, all backends are supported, but it is advisable to use binary formats like HDFStorage or ZarrStorage to write large amounts of data.

```
from modelrunner import Result

r = Result.from_file("data.hdf5")
r.result.plot()  # plots the final state
r.storage["trajectory"]  # allows accessing the stored trajectory
```

Parameters

- **storage** (StorageGroup) – Modelrunner storage used for storing the trajectory
- **loc** (*str or list of str*) – The location in the storage where the trajectory data is written.
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

close()

Close the currently opened trajectory writer.

Return type
None

property data

The actual data for all time.

Type
ndarray

end_writing()

Finalize the storage after writing.

This makes sure the data is actually written to a file when self.keep_opened == False

Return type
None

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type
None

property times

The times at which data is available.

Type
ndarray

class MovieStorage(*filename*, ***, *vmin=0*, *vmax=1*, *bits_per_channel=16*, *video_format='auto'*, *bitrate=-1*, *info=None*, *write_mode='truncate_once'*, *write_times=False*, *loglevel='warning'*)

Bases: *StorageBase*

Store discretized fields in a movie file.

This storage only works when the *ffmpeg* program and `ffmpeg` is installed. The default codec is FFV1, which supports lossless compression for various configurations. Not all video players support this codec, but VLC usually works quite well.

Note that important meta information is stored as a comment in the movie, so this data must not be deleted or altered if the video should be read again.

Warning

This storage potentially compresses data and can thus lead to loss of some information. The data quality depends on many parameters, but most important are the bits per channel of the video format and the range that is encoded (determined by *vmin* and *vmax*).

Note also that selecting individual time points might be quite slow since the video needs to be read from the beginning each time. Instead, it is much more efficient to process entire videos (by iterating over them or using *items()*).

Parameters

- **filename** (*str*) – The path where the movie is stored. The file extension determines the container format of the movie. The standard codec FFV1 plays well with the ".avi", ".mkv", and ".mov" container format.
- **vmin** (*float or array*) – Lowest values that are encoded (per field). Smaller values are clipped to this value.
- **vmax** (*float or array*) – Highest values that are encoded (per field). Larger values are clipped to this value.
- **bits_per_channel** (*int*) – The number of bits used per color channel. Typical values are 8 and 16. The relative accuracy of stored values is 0.01 and 0.0001, respectively.
- **video_format** (*str*) – Identifier for a video format from *formats*, which determines the number of channels, the bit depth of individual colors, and the codec. The special value *auto* tries to find a suitable format automatically, taking *bits_per_channel* into account.
- **bitrate** (*float*) – The bitrate of the movie (in kilobits per second). The default value of -1 let's the encoder choose an appropriate bit rate.
- **info** (*dict*) – Supplies extra information that is stored in the storage alongside additional information necessary to reconstruct fields and grids.
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.
- **write_times** (*bool*) – Flag determining whether timestamps are written to a file. If True, a separate file with name `filename + ".times"` is created where the times are written as plain text. Without these timestamps, the time information might be inaccurate.
- **loglevel** (*str*) – FFmpeg log level determining the amount of data sent to stdout. The default only emits warnings and errors, but setting this to *"info"* can be useful to get additional information about the encoding.

clear()

Truncate the storage by removing all stored data.

close()

Close the currently opened file.

Return type
None

property data

The actual data for all times.

Type
ndarray

end_writing()

Finalize the storage after writing.

Return type
None

items()

Iterate over all times and stored fields, returning pairs.

Return type
Iterator[tuple[float, *FieldBase*]]

start_writing(*field*, *info=None*)
Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type
None

times
The times at which data is available.

Type
ndarray

tracker(*interrupts=1*, ***, *transformation=None*)
Create object that can be used as a tracker to fill this storage.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then stored. This allows to store derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

Returns
The tracker that fills the current storage

Return type
StorageTracker

Example

The *transformation* argument allows storing additional fields:

```
def add_to_state(state):
    transformed_field = state.smooth(1)
    return field.append(transformed_field)


storage = pde.MemoryStorage()
tracker = storage.tracker(1, transformation=add_to_state)
eq.solve(..., tracker=tracker)
```

In this example, `storage` will contain a trajectory of the fields of the simulation as well as the smoothed fields. Other transformations are possible by defining appropriate `add_to_state()`

get_memory_storage(*field*, *info=None*)

A context manager that can be used to create a MemoryStorage.

Example

This can be used to quickly store data:

```
with get_memory_storage(field_class) as storage:
    storage.append(numpy_array0, 0)
    storage.append(numpy_array1, 1)

# use storage thereafter
```

Parameters

- **field** (FieldBase) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage

Yields

MemoryStorage

4.6.1 pde.storage.base module

Base classes for storing data.

StorageBase	Base class for storing time series of discretized fields.
StorageTracker	Tracker that stores data in special storage classes.

class StorageBase(*, *info=None*, *write_mode='truncate_once'*)

Bases: object

Base class for storing time series of discretized fields.

These classes store time series of *FieldBase*, i.e., they store the values of the fields at particular time points. Iterating of the storage will return the fields in order and individual time points can also be accessed.

Parameters

- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing one. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but subsequent data using the same instances are appended). Alternatively, specifying 'readonly' will disable writing completely.

append(*field*, *time=None*)

Add field to the storage.

Parameters

- **field** (*FieldBase*) – The field that is added to the storage
- **time** (*float, optional*) – The time point

Return type

None

apply(*func*, *out=None*, *, *progress=False*)

Applies function to each field in a storage.

Parameters

- **func** (*callable*) – The function to apply to each stored field. The function must either take as a single argument the field or as two arguments the field and the associated time point. In both cases, it should return a field.
- **out** (*StorageBase*) – Storage to which the output is written. If omitted, a new *MemoryStorage* is used and returned
- **progress** (*bool*) – Flag indicating whether the progress is shown during the calculation

Returns

The new storage that contains the data after the function *func* has been applied

Return type

StorageBase

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

Return type

None

copy(*out=None*, *, *progress=False*)

Copies all fields in a storage to a new one.

Parameters

- **out** (*StorageBase*) – Storage to which the output is written. If omitted, a new *MemoryStorage* is used and returned
- **progress** (*bool*) – Flag indicating whether the progress is shown during the calculation

Returns

The new storage that contains the copied data

Return type

StorageBase

data: Any

property data_shape: tuple[int, ...]

The current data shape.

Raises

RuntimeError – if data_shape was not set

property dtype: DTypeLike

The current data type.

Raises

RuntimeError – if data_type was not set

end_writing()

Finalize the storage after writing.

Return type
None

extract_field(*field_id*, *label=None*)

Extract the time course of a single field from a collection.

This method makes a copy of the underlying data.

Parameters

- **field_id** (*int or str*) – The index into the field collection. This determines which field of the collection is returned. Instead of a numerical index, the field label can also be supplied. If there are multiple fields with the same label, only the first field is returned.
- **label** (*str*) – The label of the returned field. If omitted, the stored label is used.

Returns
a storage instance that contains the data for the single field

Return type
MemoryStorage

extract_time_range(*t_range=None*)

Extract a particular time interval.

> **Note**
>
> This might return a view into the original data, so modifying the returned data can also change the underlying original data.

Parameters
t_range (*float or tuple*) – Determines the range of time points included in the result. If only a single number is given, all data up to this time point are included.

Returns
a storage instance that contains the extracted data.

Return type
MemoryStorage

property grid: *GridBase* | None

the grid associated with this storage

This returns *None* if grid was not stored in *self.info*.

Type
GridBase

property has_collection: bool

whether the storage is storing a collection

Type
bool

items()

Iterate over all times and stored fields, returning pairs.

Return type
Iterator[tuple[float, *FieldBase*]]

property shape: tuple[int, ...] | None

The shape of the stored data.

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type

None

times: Sequence[float]

tracker(*interrupts=1*, ***, *transformation=None*)

Create object that can be used as a tracker to fill this storage.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then stored. This allows to store derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

Returns

The tracker that fills the current storage

Return type

StorageTracker

Example

The *transformation* argument allows storing additional fields:

```
def add_to_state(state):
    transformed_field = state.smooth(1)
    return field.append(transformed_field)


storage = pde.MemoryStorage()
tracker = storage.tracker(1, transformation=add_to_state)
eq.solve(..., tracker=tracker)
```

In this example, `storage` will contain a trajectory of the fields of the simulation as well as the smoothed fields. Other transformations are possible by defining appropriate `add_to_state()`

view_field(*field_id*)

Returns a view into this storage focusing on a particular field.

> **Note**
>
> Modifying data returned by the view will modify the underlying storage

Parameters

field_id (*int or str*) – The index into the field collection. This determines which field of the collection is returned. Instead of a numerical index, the field label can also be supplied. If there are multiple fields with the same label, only the first field is returned.

Returns

A view into the storage only returning a single field

Return type

StorageView

write_mode: WriteModeType

class StorageTracker(*storage*, *interrupts=1*, ***, *transformation=None*)

Bases: *TransformedTrackerBase*

Tracker that stores data in special storage classes.

storage

The underlying storage class through which the data can be accessed

Type

StorageBase

Parameters

- **storage** (*StorageBase*) – Storage instance to which the data is written
- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then stored. This allows to store derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters

info (*dict*) – Extra information from the simulation

Return type

None

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation

- **t** (*float*) – The associated time

Return type
None

initialize(*field*, *info=None*)

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

class StorageView(*storage*, *, *field*)

Bases: object

Represents a view into a storage that extracts a particular field.

Parameters

- **storage** (*StorageBase*) – The storage providing the basic data
- **field** (*int or str*) – The index into the field collection determining which field of the collection is returned. Instead of a numerical index, the field label can also be supplied. If there are multiple fields with the same label, only the first field is returned.

property grid: *GridBase* | None

has_collection: bool = False

items()

Iterate over all times and stored fields, returning pairs.

Return type
Iterator[tuple[float, *DataFieldBase*]]

property times: Sequence[float]

4.6.2 pde.storage.file module

Defines a class storing data on the file system using the hierarchical data format (hdf)

class FileStorage(*filename*, *, *info=None*, *write_mode='truncate_once'*, *max_length=None*, *compression=True*, *keep_opened=True*, *check_mpi=True*)

Bases: *StorageBase*

Store discretized fields in a hdf5 file.

Parameters

- **filename** (*str*) – The path to the hdf5-file where the data is stored
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

- **max_length** (*int, optional*) – Maximal number of entries that will be stored in the file. This can be used to preallocate data, which can lead to smaller files, but is also less flexible. Giving *max_length = None*, allows for arbitrarily large data, which might lead to larger files.
- **compression** (*bool*) – Whether to store the data in compressed form. Automatically enabled chunked storage.
- **keep_opened** (*bool*) – Flag indicating whether the file should be kept opened after each writing. If *False*, the file will be closed after writing a dataset. This keeps the file in a consistent state, but also requires more work before data can be written.
- **check_mpi** (*bool*) – If True, files will only be opened in the main node for an parallel simulation using MPI. This flag has no effect in serial code.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

close()

Close the currently opened file.

Return type

None

property data

The actual data for all time.

Type

ndarray

end_writing()

Finalize the storage after writing.

This makes sure the data is actually written to a file when self.keep_opened == False

Return type

None

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type

None

property times

The times at which data is available.

Type

ndarray

4.6.3 pde.storage.memory module

Defines a class storing data in memory.

class MemoryStorage(*times=None*, *data=None*, ***, *info=None*, *field_obj=None*, *write_mode='truncate_once'*)

Bases: *StorageBase*

Store discretized fields in memory.

Parameters

- **times** (ndarray) – Sequence of times for which data is known
- **data** (list of ndarray) – The field data at the given times
- **field_obj** (*FieldBase*) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

Return type
None

classmethod from_collection(*storages*, *label=None*, ***, *rtol=1e-05*, *atol=1e-08*)

Combine multiple memory storages into one.

This method can be used to combine multiple time series of different fields into a single representation. This requires that all time series contain data at the same time points.

Parameters

- **storages** (*list*) – A collection of instances of *StorageBase* whose data will be concatenated into a single MemoryStorage
- **label** (*str, optional*) – The label of the instances of *FieldCollection* that represent the concatenated data
- **rtol** (*float*) – Relative tolerance used when checking times for merging
- **atol** (*float*) – Absolute tolerance used when checking times for merging

Returns
Storage containing all the data.

Return type
MemoryStorage

classmethod from_fields(*times=None*, *fields=None*, *info=None*, *write_mode='truncate_once'*)

Create MemoryStorage from a list of fields.

Parameters

- **times** (ndarray) – Sequence of times for which data is known
- **fields** (list of FieldBase) – The fields at all given time points

- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

Return type
MemoryStorage

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type
None

get_memory_storage(*field*, *info=None*)

A context manager that can be used to create a MemoryStorage.

Example

This can be used to quickly store data:

```
with get_memory_storage(field_class) as storage:
    storage.append(numpy_array0, 0)
    storage.append(numpy_array1, 1)

# use storage thereafter
```

Parameters

- **field** (FieldBase) – An instance of the field class store data for a single time point.
- **info** (*dict*) – Supplies extra information that is stored in the storage

Yields
MemoryStorage

4.6.4 pde.storage.modelrunner module

Defines a class storing data using modelrunner.

class ModelrunnerStorage(*storage*, *, *loc='trajectory'*, *info=None*, *write_mode='truncate_once'*)

Bases: *StorageBase*

Store discretized fields in a modelrunner storage.

This storage class acts as a wrapper for the trajectory module, which allows handling time-dependent data in modelrunner storages. In principle, all backends are supported, but it is advisable to use binary formats like HDFStorage or ZarrStorage to write large amounts of data.

```
from modelrunner import Result

r = Result.from_file("data.hdf5")
r.result.plot()  # plots the final state
r.storage["trajectory"]  # allows accessing the stored trajectory
```

Parameters

- **storage** (StorageGroup) – Modelrunner storage used for storing the trajectory
- **loc** (*str or list of str*) – The location in the storage where the trajectory data is written.
- **info** (*dict*) – Supplies extra information that is stored in the storage
- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.

clear(*clear_data_shape=False*)

Truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

close()

Close the currently opened trajectory writer.

Return type

None

property data

The actual data for all time.

Type

ndarray

end_writing()

Finalize the storage after writing.

This makes sure the data is actually written to a file when self.keep_opened == False

Return type

None

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type

None

property times

The times at which data is available.

Type
ndarray

4.6.5 pde.storage.movie module

Defines a class storing data on the file system as a compressed movie.

This package requires the optional `ffmpeg-python` package to use FFmpeg for reading and writing movies.

class MovieStorage(*filename*, *, *vmin=0*, *vmax=1*, *bits_per_channel=16*, *video_format='auto'*, *bitrate=-1*, *info=None*, *write_mode='truncate_once'*, *write_times=False*, *loglevel='warning'*)

Bases: *StorageBase*

Store discretized fields in a movie file.

This storage only works when the *ffmpeg* program and `ffmpeg` is installed. The default codec is FFV1, which supports lossless compression for various configurations. Not all video players support this codec, but VLC usually works quite well.

Note that important meta information is stored as a comment in the movie, so this data must not be deleted or altered if the video should be read again.

Warning

This storage potentially compresses data and can thus lead to loss of some information. The data quality depends on many parameters, but most important are the bits per channel of the video format and the range that is encoded (determined by *vmin* and *vmax*).

Note also that selecting individual time points might be quite slow since the video needs to be read from the beginning each time. Instead, it is much more efficient to process entire videos (by iterating over them or using *items()*).

Parameters

- **filename** (*str*) – The path where the movie is stored. The file extension determines the container format of the movie. The standard codec FFV1 plays well with the ".avi", ".mkv", and ".mov" container format.
- **vmin** (*float or array*) – Lowest values that are encoded (per field). Smaller values are clipped to this value.
- **vmax** (*float or array*) – Highest values that are encoded (per field). Larger values are clipped to this value.
- **bits_per_channel** (*int*) – The number of bits used per color channel. Typical values are 8 and 16. The relative accuracy of stored values is 0.01 and 0.0001, respectively.
- **video_format** (*str*) – Identifier for a video format from *formats*, which determines the number of channels, the bit depth of individual colors, and the codec. The special value *auto* tries to find a suitable format automatically, taking *bits_per_channel* into account.
- **bitrate** (*float*) – The bitrate of the movie (in kilobits per second). The default value of -1 let's the encoder choose an appropriate bit rate.
- **info** (*dict*) – Supplies extra information that is stored in the storage alongside additional information necessary to reconstruct fields and grids.

- **write_mode** (*str*) – Determines how new data is added to already existing data. Possible values are: 'append' (data is always appended), 'truncate' (data is cleared every time this storage is used for writing), or 'truncate_once' (data is cleared for the first writing, but appended subsequently). Alternatively, specifying 'readonly' will disable writing completely.
- **write_times** (*bool*) – Flag determining whether timestamps are written to a file. If True, a separate file with name `filename + ".times"` is created where the times are written as plain text. Without these timestamps, the time information might be inaccurate.
- **loglevel** (*str*) – FFmpeg log level determining the amount of data sent to stdout. The default only emits warnings and errors, but setting this to *"info"* can be useful to get additional information about the encoding.

clear()

Truncate the storage by removing all stored data.

close()

Close the currently opened file.

Return type
None

property data

The actual data for all times.

Type
ndarray

end_writing()

Finalize the storage after writing.

Return type
None

items()

Iterate over all times and stored fields, returning pairs.

Return type
Iterator[tuple[float, *FieldBase*]]

start_writing(*field*, *info=None*)

Initialize the storage for writing data.

Parameters

- **field** (FieldBase) – An example of the data that will be written to extract the grid and the data_shape
- **info** (*dict*) – Supplies extra information that is stored in the storage

Return type
None

times

The times at which data is available.

Type
ndarray

tracker(*interrupts=1*, ***, *transformation=None*)

Create object that can be used as a tracker to fill this storage.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then stored. This allows to store derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

Returns

The tracker that fills the current storage

Return type

StorageTracker

Example

The *transformation* argument allows storing additional fields:

```
def add_to_state(state):
    transformed_field = state.smooth(1)
    return field.append(transformed_field)


storage = pde.MemoryStorage()
tracker = storage.tracker(1, transformation=add_to_state)
eq.solve(..., tracker=tracker)
```

In this example, `storage` will contain a trajectory of the fields of the simulation as well as the smoothed fields. Other transformations are possible by defining appropriate `add_to_state()`

4.7 pde.tools package

Package containing several tools required in py-pde.

cache	Functions, classes, and decorators for managing caches.
config	Handles configuration variables of the package.
cuboid	An n-dimensional, axes-aligned cuboid.
docstrings	Methods for automatic transformation of docstrings.
expressions	Handling mathematical expressions with sympy.
ffmpeg	Functions for interacting with FFmpeg.
math	Auxiliary mathematical functions.
misc	Miscellaneous python functions.
modelrunner	Establishes hooks for the interplay between *pde* and *modelrunner*

continues on next page

Table 55 – continued from previous page

mpi	Auxiliary functions and variables for dealing with MPI multiprocessing.
nested_dict	Provides a nested dictionary that stores hierarchical mappings.
output	Python functions for handling output.
parse_duration	Parsing time durations from strings.
plotting	Tools for plotting and controlling plot output using context managers.
spectral	Functions making use of spectral decompositions.
typing	Provides support for mypy type checking of the package.

4.7.1 pde.tools.cache module

Functions, classes, and decorators for managing caches.

cached_property	Decorator to use a method as a cached property.
cached_method	Decorator to enable caching of a method.
hash_mutable	Return hash also for (nested) mutable objects.
hash_readable	Return human readable hash also for (nested) mutable objects.
make_serializer	Returns a function that serialize data with the given method.
make_unserializer	Returns a function that unserialize data with the given method.
DictFiniteCapacity	Cache with a limited number of items.
SerializedDict	A key value database which is stored on the disk.

class DictFiniteCapacity(**args*, ***kwargs*)

Bases: OrderedDict

Cache with a limited number of items.

check_length()

Ensures that the dictionary does not grow beyond its capacity.

default_capacity: int = 100

update([*E*,]**F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class SerializedDict(*key_serialization='pickle'*, *value_serialization='pickle'*, *storage_dict=None*)

Bases: MutableMapping

A key value database which is stored on the disk.

This class provides hooks for converting arbitrary keys and values to strings, which are then stored in the database.

Provides a dictionary whose keys and values are serialized.

Parameters

- **key_serialization** (*str*) – Determines the serialization method for keys
- **value_serialization** (*str*) – Determines the serialization method for values

- **storage_dict** (*dict*) – Can be used to chose a different dictionary for the underlying storage mechanism (e.g., PersistentDict())

class cached_method(*factory=None*, *extra_args=None*, *ignore_args=None*, *hash_function='hash_mutable'*, *doc=None*, *name=None*)

Bases: `_class_cache`

Decorator to enable caching of a method.

The function is only called the first time and each successive call returns the cached result of the first call.

Example

The decorator can be used like so:

```
class Foo:
    @cached_method
    def bar(self):
        return "Cached"


foo = Foo()
result = foo.bar()
```

The data is stored in a dictionary named `_cache_methods` attached to the instance of each object. The cache can thus be cleared by setting `self._cache_methods = {}`. The cache of specific property can be cleared using `self._cache_methods[property_name] = {}`, where *property_name* is the name of the property.

Decorator that caches calls in a dictionary attached to the instances. This can be used with most classes.

Example

An example for using the class is:

```
class Foo:
    @cached_property()
    def property(self):
        return "Cached property"

    @cached_method()
    def method(self):
        return "Cached method"


foo = Foo()
foo.property
foo.method()
```

The cache can be cleared by setting `foo._cache_methods = {}` if the cache factory is a simple dict, i.e, if `factory == None`. Alternatively, each cached method has a `clear_cache_of_obj()` method, which clears the cache of this particular method. In the example above we could thus call `foo.bar.clear_cache_of_obj(foo)` to clear the cache.

Note that the object instance has to be passed as a parameter, since the method `bar()` is defined on the class, not

the instance, i.e., we could also call `Foo.bar.clear_cache_of_obj(foo)`. To clear the cache from within a method, one can thus call `self.method_name.clear_cache_of_obj(self)`, where *method_name* is the name of the method whose cache is cleared

Example

An advanced example is:

```
class Foo:
    def get_cache(self, name):
        # `name` is the name of the method to cache
        return DictFiniteCapacity()

    @cached_method(factory="get_cache")
    def foo(self):
        return "Cached"
```

Parameters

- **factory** (*callable*) – Function/class creating an empty cache. *dict* by default. This can be used with user-supplied storage backends by. The cache factory should return a dict-like object that handles the cache for the given method.
- **extra_args** (*list*) – List of attributes of the class that are included in the cache key. They are then treated as if they are supplied as arguments to the method. This is important to include when the result of a method depends not only on method arguments but also on instance attributes.
- **ignore_args** (*list*) – List of keyword arguments that are not included in the cache key. These should be arguments that do not influence the result of a method, e.g., because they only affect how intermediate results are displayed.
- **hash_function** (*str*) – An identifier determining what hash function is used on the arguments
- **doc** (*str*) – Optional string giving the docstring of the decorated method
- **name** (*str*) – Optional string giving the name of the decorated method

class cached_property(*factory=None*, *extra_args=None*, *ignore_args=None*, *hash_function='hash_mutable'*, *doc=None*, *name=None*)

Bases: `_class_cache`

Decorator to use a method as a cached property.

The function is only called the first time and each successive call returns the cached result of the first call.

Example

Here is an example for how to use the decorator:

```
class Foo:
    @cached_property
    def bar(self):
        return "Cached"
```

```
foo = Foo()
result = foo.bar
```

The data is stored in a dictionary named *_cache_methods* attached to the instance of each object. The cache can thus be cleared by setting `self._cache_methods = {}`. The cache of specific property can be cleared using `self._cache_methods[property_name] = {}`, where *property_name* is the name of the property.

Adapted from <https://wiki.python.org/moin/PythonDecoratorLibrary>.

Decorator that caches calls in a dictionary attached to the instances. This can be used with most classes.

Example

An example for using the class is:

```
class Foo:
    @cached_property()
    def property(self):
        return "Cached property"

    @cached_method()
    def method(self):
        return "Cached method"


foo = Foo()
foo.property
foo.method()
```

The cache can be cleared by setting `foo._cache_methods = {}` if the cache factory is a simple dict, i.e, if `factory == None`. Alternatively, each cached method has a `clear_cache_of_obj()` method, which clears the cache of this particular method. In the example above we could thus call `foo.bar.clear_cache_of_obj(foo)` to clear the cache.

Note that the object instance has to be passed as a parameter, since the method `bar()` is defined on the class, not the instance, i.e., we could also call `Foo.bar.clear_cache_of_obj(foo)`. To clear the cache from within a method, one can thus call `self.method_name.clear_cache_of_obj(self)`, where *method_name* is the name of the method whose cache is cleared

Example

An advanced example is:

```
class Foo:
    def get_cache(self, name):
        # `name` is the name of the method to cache
        return DictFiniteCapacity()

    @cached_method(factory="get_cache")
    def foo(self):
        return "Cached"
```

Parameters

- **factory** (*callable*) – Function/class creating an empty cache. *dict* by default. This can be used with user-supplied storage backends by. The cache factory should return a dict-like object that handles the cache for the given method.
- **extra_args** (*list*) – List of attributes of the class that are included in the cache key. They are then treated as if they are supplied as arguments to the method. This is important to include when the result of a method depends not only on method arguments but also on instance attributes.
- **ignore_args** (*list*) – List of keyword arguments that are not included in the cache key. These should be arguments that do not influence the result of a method, e.g., because they only affect how intermediate results are displayed.
- **hash_function** (*str*) – An identifier determining what hash function is used on the arguments
- **doc** (*str*) – Optional string giving the docstring of the decorated method
- **name** (*str*) – Optional string giving the name of the decorated method

hash_mutable(*obj*)

Return hash also for (nested) mutable objects.

Notes

This function might be a bit slow, since it iterates over all containers and hashes objects recursively. Moreover, the returned value might change with each run of the python interpreter, since the hash values of some basic objects, like *None*, change with each instance of the interpreter.

Parameters

obj – A general python object

Returns

A hash value associated with the data of *obj*

Return type

int

hash_readable(*obj*)

Return human readable hash also for (nested) mutable objects.

This function returns a JSON-like representation of the object. The function might be a bit slow, since it iterates over all containers and hashes objects recursively. Note that this hash function tries to return the same value for equivalent objects, but it does not ensure that the objects can be reconstructed from this data.

Parameters

obj – A general python object

Returns

A hash value associated with the data of *obj*

Return type

str

make_serializer(*method*)

Returns a function that serialize data with the given method. Note that some of the methods destroy information and cannot be reverted.

Parameters
method (*str*) – An identifier determining the serializer that will be returned

Returns
A function that serializes objects

Return type
callable

make_unserializer(*method*)

Returns a function that unserialize data with the given method.

This is the inverse function of *make_serializer()*.

Parameters
method (*str*) – An identifier determining the unserializer that will be returned

Returns
A function that serializes objects

Return type
callable

objects_equal(*a*, *b*)

Compares two objects to see whether they are equal.

In particular, this uses `numpy.array_equal()` to check for numpy arrays

Parameters

- **a** – The first object
- **b** – The second object

Returns
Whether the two objects are considered equal

Return type
bool

4.7.2 pde.tools.config module

Handles configuration variables of the package.

Parameter	Class representing a single parameter.
Config	Class handling general (nested) configurations.
get_package_versions	Tries to load certain python packages and returns their version.
parse_version_str	Helper function converting a version string into a list of integers.
check_package_version	Checks whether a package has a sufficient version.
packages_from_requirements	Read package names from a requirements file.
get_ffmpeg_version	Read version number of ffmpeg program.
is_hpc_environment	Check whether the code is running in a high-performance computing environment.
environment	Obtain information about the compute environment.

exception AccessError

Bases: RuntimeError

Raised when a configuration change violates the active access mode.

class Config(*items=None*, *, *mode='update'*)

Bases: *NestedDict*[*Parameter*]

Class handling general (nested) configurations.

Configurations are basically (nested) dictionaries with string keys that hold *Parameter* values, which contain a value with some extra information. Moreover, configurations have a *mode* that controls whether the configuration is writeable or not.

Parameters

- **items** (*dict, optional*) – Configuration values that should be added or overwritten to initialize the configuration.
- **mode** (*str*) – Defines the mode in which the configuration is used. Possible values are
  - *insert*: any new configuration key can be inserted
  - *update*: only the values of pre-existing items can be updated
  - *locked*: no values can be changed

  Note that the items specified by *items* will always be inserted, independent of the *mode*.

changed_mode(***kwargs*)

Temporarily switch to *mode* and restore the previous mode afterwards.

Parameters

****kwargs** – Keyword arguments forwarded to `ConfigMode._setstate()`, such as *node*, *leaf*, and *delete*.

Yields

ConfigMode – The mode controller with the temporary mode applied.

copy()

Creates a structural copy with copied nested mappings.

Child dictionaries and child *NestedDict* instances are copied, while non-mapping leaf values are reused by reference.

Returns

New instance containing copied nested structure.

Return type

NestedDict

property mode: *ConfigMode*

Current mutable mode descriptor shared across the whole config tree.

replace_recursive(*other*, *delete_extra=False*)

Recursively replaces data of the current instance by another mapping.

Parameters

- **other** (*MutableMapping[str, Any]*) – Mapping whose entries are will end up in this object.
- **delete_extra** (*bool*)

Return type
None

to_dict(*flatten=False*, *values=False*)

Convert the configuration to a simple dictionary.

Parameters

- **flatten** (*bool*) – Return flat or nested dictionary.
- **values** (*bool*) – Whether to return only values (and not *Parameter* instances)

Returns
A representation of the configuration in a normal dict.

Return type
dict

class ConfigMode(*node=Modes.UPDATE*, *leaf=Modes.INSERT*, *delete=False*)

Bases: object

Mutable object storing the current configuration mode.

Parameters

- **mode** – Initial mode controlling whether items can be inserted, updated, or modified at all.
- **node** (Modes)
- **leaf** (Modes)
- **delete** (*bool*)

delete: bool = False

classmethod from_str(*value*)

Create a mode descriptor from a textual mode name.

Parameters
value (*str*) – Mode name. Supported values are "insert", "update", and "locked".

Returns
Newly created mode descriptor.

Return type
ConfigMode

Raises
ValueError – If *value* is not one of the supported mode names.

leaf: *Modes* = 'insert'

node: *Modes* = 'update'

class Modes(*value*)

Bases: Enum

Access modes controlling how configuration entries can be modified.

INSERT = 'insert'

READONLY = 'readonly'

UPDATE = 'update'

class Parameter(*value*, *, *default_value=<_OMITTED_TYPE object>*, *cls=<_OMITTED_TYPE object>*, *description=''*, *hidden=False*, *extra=None*)

Bases: object

Class representing a single parameter.

Parameters

- **value** (*str | float | int | bool | None*) – The current value of the parameter.
- **default_value** (*str | float | int | bool | None | _OMITTED_TYPE*) – The fallback value used when the parameter is reset. If omitted, the current value is used.
- **cls** (*object | _OMITTED_TYPE*) – Type used to convert assigned values.
- **description** (*str*) – Human-readable explanation of the parameter.
- **hidden** (*bool*) – Flag indicating whether the parameter should be hidden in summaries.
- **extra** (*dict[str, Any] | None*) – Optional metadata stored alongside the parameter.

cls: object | _OMITTED_TYPE = <pde.tools.config._OMITTED_TYPE object>

convert(*value=<_OMITTED_TYPE object>*)

Converts a *value* into the correct type for this parameter. If *value* is not given, the current value is converted.

Note that this does not make a copy of the values, which could lead to unexpected effects where the default value is changed by an instance.

Parameters

value (*str | float | int | bool | None | _OMITTED_TYPE*) – The value to convert

Returns

The converted value, which is of type *self.cls*

Return type

str | float | int | bool | None

default_value: str | float | int | bool | None | _OMITTED_TYPE = <pde.tools.config._OMITTED_TYPE object>

description: str = ''

extra: dict[str, Any] | None = None

hidden: bool = False

reset()

Reset parameter to default value.

Return type

None

value: str | float | int | bool | None

check_package_version(*package_name*, *min_version*)

Checks whether a package has a sufficient version.

Parameters

- **package_name** (*str*) – The name of the package to check
- **min_version** (*str*) – The minimum required version

Returns
The function only emits warnings and does not return a value.

Return type
None

environment()

Obtain information about the compute environment.

Returns
information about the python installation and packages

Return type
dict

get_ffmpeg_version()

Read version number of ffmpeg program.

Returns
Detected version string, or *None* if ffmpeg is unavailable or the version could not be parsed.

Return type
str | None

get_package_versions(*packages*, *, *na_str='not available'*)

Tries to load certain python packages and returns their version.

Parameters

- **packages** (*list*) – The names of all packages
- **na_str** (*str*) – Text to return if package is not available

Returns
Dictionary with version for each package name

Return type
dict

is_hpc_environment()

Check whether the code is running in a high-performance computing environment.

Returns
True if running in an HPC environment, False otherwise.

Return type
bool

packages_from_requirements(*requirements_file*)

Read package names from a requirements file.

Parameters
requirements_file (str or Path) – The file from which everything is read

Returns
list of package names

Return type
list[str]

parse_version_str(*ver_str*)

Helper function converting a version string into a list of integers.

Parameters
ver_str (*str*) – The version string to parse

Returns
List of version numbers as integers

Return type
list[int]

4.7.3 pde.tools.cuboid module

An n-dimensional, axes-aligned cuboid.

This module defines the *Cuboid* class, which represents an n-dimensional cuboid that is aligned with the axes of a Cartesian coordinate system.

class Cuboid(*pos*, *size*, *mutable=True*)

Bases: object

Class that represents a cuboid in n dimensions.

Defines a cuboid from a position and a size vector.

Parameters

- **pos** (*list*) – The position of the lower left corner. The length of this list determines the dimensionality of space
- **size** (*list*) – The size of the cuboid along each dimension.
- **mutable** (*bool*) – Flag determining whether the cuboid parameters can be changed

property bounds: tuple[tuple[float, float], ...]

buffer(*amount=0*, *inplace=False*)

Dilate the cuboid by a certain amount in all directions.

Parameters

- **amount** (*FloatOrArray*) – The amount to dilate (can be a scalar or array)
- **inplace** (*bool*) – Whether to modify in place or return a new cuboid

Return type
Cuboid

property centroid

contains_point(*points*)

Returns a True when *points* are within the Cuboid.

Parameters
points (ndarray) – List of point coordinates

Returns
list of booleans indicating which points are inside

Return type
ndarray

copy()

Return type
Cuboid

property corners: tuple[NumericArray, NumericArray]

Return coordinates of two extreme corners defining the cuboid.

property diagonal: float

Returns the length of the diagonal.

property dim: int

classmethod from_bounds(*bounds*, ***kwargs*)

Create cuboid from bounds.

Parameters

- **bounds** (*list*) – Two dimensional array of axes bounds
- ****kwargs** – Additional keyword arguments passed to the constructor

Returns

cuboid with positive size

Return type

Cuboid

classmethod from_centerpoint(*centerpoint*, *size*, ***kwargs*)

Create cuboid from two points.

Parameters

- **centerpoint** (*list*) – Coordinates of the center
- **size** (*list*) – Size of the cuboid
- ****kwargs** – Additional keyword arguments passed to the constructor

Returns

cuboid with positive size

Return type

Cuboid

classmethod from_points(*p1*, *p2*, ***kwargs*)

Create cuboid from two points.

Parameters

- **p1** (*list*) – Coordinates of first corner point
- **p2** (*list*) – Coordinates of second corner point
- ****kwargs** – Additional keyword arguments passed to the constructor

Returns

cuboid with positive size

Return type

Cuboid

property mutable: bool

property size: NumericArray

property surface_area: float

Surface area of a cuboid in n dimensions.

The surface area is the volume of the $(n-1)$-dimensional hypercubes that bound the current cuboid:

- $n = 1$: the number of end points (2)
- $n = 2$: the perimeter of the rectangle
- $n = 3$: the surface area of the cuboid

property vertices: list[list[float]]

Return the coordinates of all the corners.

property volume: float

asanyarray_flags(*data*, *dtype=None*, *writeable=True*)

Turns data into an array and sets the respective flags.

A copy is only made if necessary

Parameters

- **data** (ndarray) – numpy array that whose flags are adjusted
- **dtype** (*DTypeLike | None*) – The resultant dtype
- **writeable** (*bool*) – Flag determining whether the results is writable

Returns

array with same data as *data* but with flags adjusted.

Return type

ndarray

4.7.4 pde.tools.docstrings module

Methods for automatic transformation of docstrings.

get_text_block	Return a single text block.
replace_in_docstring	Replace a text in a docstring using the correct indentation.
fill_in_docstring	Decorator that replaces text in the docstring of a function.

fill_in_docstring(*f*)

Decorator that replaces text in the docstring of a function.

Parameters

f (*TFunc*) – The function to decorate

Return type

TFunc

get_text_block(*identifier*)

Return a single text block.

Parameters

identifier (*str*) – The name of the text block

Returns

the text block as one long line.

Return type
str

replace_in_docstring(*f*, *token*, *value*, *docstring=None*)

Replace a text in a docstring using the correct indentation.

Parameters

- **f** (*callable*) – The function with the docstring to handle
- **token** (*str*) – The token to search for
- **value** (*str*) – The replacement string
- **docstring** (*str*) – A docstring that should be used instead of f.__doc__

Returns
The function with the modified docstring

Return type
callable

4.7.5 pde.tools.expressions module

Handling mathematical expressions with sympy.

This module provides classes representing expressions that can be provided as human-readable strings and are converted to `numpy` and `numba` representations using `sympy`.

parse_number	Return a number compiled from an expression.
ScalarExpression	Describes a mathematical expression of a scalar quantity.
TensorExpression	Describes a mathematical expression of a tensorial quantity.
evaluate	Evaluate an expression involving fields.

class ExpressionBase(*expression*, *signature=None*, *, *user_funcs=None*, *consts=None*, *repl=None*)

Bases: `object`

Abstract base class for handling expressions.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **expression** (`sympy.core.basic.Basic`) – A sympy expression or array. This could for instance be an instance of `Expr` or `NDimArray`.
- **signature** (*list of str, optional*) – The signature defines which variables are expected in the expression. This is typically a list of strings identifying the variable names. Individual names can be specified as list, in which case any of these names can be used. The first item in such a list is the definite name and if another name of the list is used, the associated variable is renamed to the definite name. If signature is *None*, all variables in *expressions* are allowed.

- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or ndarray.
- **repl** (*dict, optional*) – Replacements that are applied to symbols before turning the expression into a python equivalent.

property complex: bool

whether the expression contains the imaginary unit I

Type
bool

property constant: bool

whether the expression is a constant

Type
bool

depends_on(*variable*)

Determine whether the expression depends on *variable*

Parameters
variable (*str*) – the name of the variable to check for

Returns
whether the variable appears in the expression

Return type
bool

property expression: str

the expression in string form

Type
str

get_compiled(*single_arg=False*)

Return numba function evaluating expression.

Parameters
single_arg (*bool*) – Determines whether the function takes all variables in a single argument as an array or whether all variables need to be supplied separately.

Returns
the compiled function

Return type
function

get_function(*backend='numpy'*, ***, *single_arg=False*, *user_funcs=None*)

Return a function evaluating expression for a particular backend.

Parameters

- **backend** (*str*) – The backend (e.g., *numba* or *numpy*) for constructing the function
- **single_arg** (*bool*) – Determines whether the returned function accepts all variables in a single argument as an array or whether all variables need to be supplied separately.
- **user_funcs** (*dict*) – Additional functions that can be used in the expression.

Returns
the function

Return type
function

property rank: int
the rank of the expression

Type
int

abstract property shape: tuple[int, ...]
the shape of the tensor

Type
tuple

class ScalarExpression(*expression=0*, *signature=None*, ***, *user_funcs=None*, *consts=None*, *repl=None*, *explicit_symbols=None*, *allow_indexed=False*)

Bases: *ExpressionBase*

Describes a mathematical expression of a scalar quantity.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **expression** (*str or float*) – The expression, either a number or a string that sympy can parse.
- **signature** (*list of str*) – The signature defines which variables are expected in the expression. This is typically a list of strings identifying the variable names. Individual names can be specified as lists, in which case any of these names can be used. The first item in such a list is the definite name and if another name of the list is used, the associated variable is renamed to the definite name. If signature is *None*, all variables in *expressions* are allowed.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **repl** (*dict, optional*) – Replacements that are applied to symbols before turning the expression into a python equivalent.
- **explicit_symbols** (*list of str*) – List of symbols that need to be interpreted as general sympy symbols
- **allow_indexed** (*bool*) – Whether to allow indexing of variables. If enabled, array variables are allowed to be indexed using square bracket notation.

copy()
Return a copy of the current expression.

Return type
ScalarExpression

derivatives
Differentiate the expression with respect to all variables.

differentiate(*var*)
Return the expression differentiated with respect to var.

Parameters
var (*str*) – The variable to differentiate with respect to

Return type
ScalarExpression

property is_zero: bool
returns whether the expression is zero

Type
bool

shape: tuple[int, ...] = ()

property value: int | float | *complex* | number
the value for a constant expression

Type
float

class TensorExpression(*expression*, *signature=None*, ***, *user_funcs=None*, *consts=None*, *repl=None*, *explicit_symbols=None*)

Bases: *ExpressionBase*

Describes a mathematical expression of a tensorial quantity.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **expression** (*str or float*) – The expression, either a number or a string that sympy can parse.
- **signature** (*list of str*) – The signature defines which variables are expected in the expression. This is typically a list of strings identifying the variable names. Individual names can be specified as list, in which case any of these names can be used. The first item in such a list is the definite name and if another name of the list is used, the associated variable is renamed to the definite name. If signature is *None*, all variables in *expressions* are allowed.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expression.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. The values of these constants should either be numbers or `ndarray`.
- **repl** (*dict, optional*) – Replacements that are applied to symbols before turning the expression into a python equivalent.

- **explicit_symbols** (*list of str*) – List of symbols that need to be interpreted as general sympy symbols

derivatives

Differentiate the expression with respect to all variables.

differentiate(*var*)

Return the expression differentiated with respect to var.

Parameters

var (*str*) – The variable to differentiate with respect to

Return type

TensorExpression

get_compiled_array(*single_arg=True*)

Compile the tensor expression such that a numpy array is returned.

Parameters

single_arg (*bool*) – Whether the compiled function expects all arguments as a single array or whether they are supplied individually.

Return type

Callable[[NumericArray, NumericArray | None], NumericArray]

property rank: int

rank of the tensor expression

Type

int

property shape: tuple[int, ...]

the shape of the tensor

Type

tuple

property value

The value for a constant expression.

evaluate(*expression*, *fields*, *, *bc='auto_periodic_neumann'*, *bc_ops=None*, *user_funcs=None*, *consts=None*, *backend='numpy'*, *label=None*)

Evaluate an expression involving fields.

Warning

This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **expression** (*str*) – The expression, which is parsed by `sympy`. The expression may contain variables (i.e., fields and spatial coordinates of the grid), standard local mathematical operators defined by sympy, and the operators defined in the `pde` package. Note that operators need to be specified with their full name, i.e., *laplace* for a scalar Laplacian and *vector_laplace* for a Laplacian operating on a vector field. Moreover, the dot product between two vector fields can be denoted by using *dot(field1, field2)* in the expression, and *outer(field1, field2)* calculates an outer product. More information can be found in the *expression documentation*.

- **fields** (dict or *FieldCollection*) – Dictionary of the fields involved in the expression. The dictionary keys specify the field names allowed in *expression*. Alternatively, *fields* can be a *FieldCollection* with unique labels.
- **bc** (*BoundariesData*) – Boundary conditions for the operators used in the expression. The conditions here are applied to all operators that do not have a specialized condition given in *bc_ops*. Boundary conditions are generally given as a dictionary with one condition for each axis side. For periodic axes, only periodic boundary conditions are allowed (indicated by 'periodic' and 'anti- periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (using specific identifiers, like *x*- and *y*+). For instance, Dirichlet conditions enforcing a value NUM (specified by *{'value': NUM}*) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by *{'derivative': DERIV}*) are supported. Note that the special value 'auto_periodic_neumann' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_ops** (*dict*) – Special boundary conditions for some operators. The keys in this dictionary specify the operator to which the boundary condition will be applied.
- **user_funcs** (*dict, optional*) – A dictionary with user defined functions that can be used in the expressions in *rhs*.
- **consts** (*dict, optional*) – A dictionary with user defined constants that can be used in the expression. These can be either scalar numbers or fields defined on the same grid as the actual simulation.
- **backend** (*str*) – The backend used to evaluate the expression
- **label** (*str*) – Name of the field that is returned.

Returns

The resulting field. The rank of the returned field (and thus the precise class) is determined automatically.

Return type

pde.fields.base.DataFieldBase

parse_number(*expression*, *variables=None*)

Return a number compiled from an expression.

> **Warning**
>
> This implementation uses `exec()` and should therefore not be used in a context where malicious input could occur.

Parameters

- **expression** (*str or Number*) – An expression that can be interpreted as a number
- **variables** (*dict*) – A dictionary of values that replace variables in the expression

Returns

the calculated value

Return type

Number

4.7.6 pde.tools.ffmpeg module

Functions for interacting with FFmpeg.

FFmpegFormat	Defines a FFmpeg format used for storing field data in a video.
formats	Dict of pre-defined *FFmpegFormat* formats.
find_format	Find a defined FFmpegFormat that satisfies the requirements.

class FFmpegFormat(*pix_fmt_file*, *pix_fmt_data*, *channels*, *bits_per_channel*, *dtype*, *codec='ffv1'*)

Bases: object

Defines a FFmpeg format used for storing field data in a video.

> **Note**
>
> All pixel formats supported by FFmpeg can be obtained by running `ffmpeg -pix_fmts`. However, not all pixel formats are supported by all codecs. Supported pixel formats are listed in the output of `ffmpeg -h encoder=<ENCODER>`, where *<ENCODER>* is one of the encoders listed in `ffmpeg -codecs`.

Parameters

- **pix_fmt_file** (*str*)
- **pix_fmt_data** (*str*)
- **channels** (*int*)
- **bits_per_channel** (*int*)
- **dtype** (*DTypeLike*)
- **codec** (*str*)

bits_per_channel: int

number of bits per color channel in this pixel format

Type
int

property bytes_per_channel: int

number of bytes per color channel.

Type
Int

channels: int

number of color channels in this pixel format

Type
int

codec: str = 'ffv1'

name of the codec that supports this pixel format

Type
str

data_from_frame(*frame_data*)

Converts data stored in a color channel to normalized data.

Parameters
frame_data (*NumericArray*)

data_to_frame(*normalized_data*)

Converts normalized data to data being stored in a color channel.

Parameters
normalized_data (*NumericArray*)

Return type
NumericArray

dtype: DTypeLike

Numpy dtype corresponding to the data of a single channel.

property max_value: float | int

Maximal value stored in a color channel.

pix_fmt_data: str

name of the pixel format used in the frame data

Type
str

pix_fmt_file: str

name of the pixel format used in the codec

Type
str

find_format(*channels*, *bits_per_channel=8*)

Find a defined FFmpegFormat that satisfies the requirements.

Parameters

- **channels** (*int*) – Minimal number of color channels
- **bits_per_channel** (*int*) – Minimal number of bits per channel

Returns
Identifier for a format that satisfies the requirements (but might have more channels or more bits per channel then requested. *None* is returned if no format can be identified.

Return type
str

```
formats = {'gbrap16le': FFmpegFormat(pix_fmt_file='gbrap16le',
pix_fmt_data='gbrap16le', channels=4, bits_per_channel=16, dtype=dtype('uint16'),
codec='ffv1'), 'gbrp16le': FFmpegFormat(pix_fmt_file='gbrp16le',
pix_fmt_data='gbrp16le', channels=3, bits_per_channel=16, dtype=dtype('uint16'),
codec='ffv1'), 'gray': FFmpegFormat(pix_fmt_file='gray', pix_fmt_data='gray',
channels=1, bits_per_channel=8, dtype=<class 'numpy.uint8'>, codec='ffv1'), 'gray16le':
FFmpegFormat(pix_fmt_file='gray16le', pix_fmt_data='gray16le', channels=1,
bits_per_channel=16, dtype=dtype('uint16'), codec='ffv1'), 'rgb24':
FFmpegFormat(pix_fmt_file='rgb24', pix_fmt_data='rgb24', channels=3,
bits_per_channel=8, dtype=<class 'numpy.uint8'>, codec='ffv1'), 'rgb32':
FFmpegFormat(pix_fmt_file='rgb32', pix_fmt_data='rgb32', channels=4,
bits_per_channel=8, dtype=<class 'numpy.uint8'>, codec='ffv1')}
```

Dict of pre-defined *FFmpegFormat* formats.

4.7.7 pde.tools.math module

Auxiliary mathematical functions.

SmoothData1D	Allows smoothing data in 1d using a Gaussian kernel of defined width.
OnlineStatistics	Class for using an online algorithm for calculating statistics.

class OnlineStatistics

Bases: object

Class for using an online algorithm for calculating statistics.

add(*value*)

Add a value to the accumulator.

Parameters
value (*float*) – The value to add

Return type
None

count: int

recorded number of items

Type
int

mean: float

recorded mean

Type
float

property std: float

recorded standard deviation

Type
float

to_dict()

Return the information as a dictionary.

Return type
dict[str, *Any*]

property var: float

recorded variance

Type
float

class SmoothData1D(*x*, *y*, *sigma=None*)

Bases: object

Allows smoothing data in 1d using a Gaussian kernel of defined width.

The data is given a pairs of *x* and *y*, the assumption being that there is an underlying relation *y = f(x)*.

Initialize with data.

Parameters

- **x** – List of x values
- **y** – List of y values
- **sigma** (*float*) – The size of the smoothing window in units of *x*. If omitted, the average distance of x values multiplied by *sigma_auto_scale* is used.

property bounds: tuple[float, float]

Return minimal and maximal *x* values.

derivative(*xs*)

Return the derivative of the smoothed values for the positions *xs*

Note that this value

Parameters

xs (list of ndarray) – the x-values

Returns

The associated values of the derivative

Return type

ndarray

sigma_auto_scale: float = 10

scale for setting automatic values for sigma

Type

float

4.7.8 pde.tools.misc module

Miscellaneous python functions.

module_available	Check whether a python module is available.
ensure_directory_exists	Creates a folder if it not already exists.
decorator_arguments	make a decorator usable with and without arguments:
import_class	Import a class or module given an identifier.
classproperty	Decorator that can be used to define read-only properties for classes.
hybridmethod	Descriptor that can be used as a decorator to allow calling a method both as a classmethod and an instance method.
estimate_computation_speed	Estimates the computation speed of a function.
hdf_write_attributes	Write (JSON-serialized) attributes to a hdf file.
number	Convert a value into a float or complex number.
get_common_dtype	Returns a dtype in which all arguments can be represented.

continues on next page

Table 62 – continued from previous page

number_array	Convert data into an array, assuming float numbers if no dtype is given.

class classproperty(*fget=None*, *doc=None*)

Bases: property

Decorator that can be used to define read-only properties for classes.

This is inspired by the implementation of astropy, see astropy.org.

Example

The decorator can be used much like the *property* decorator:

```
class Test:
    item: str = "World"

    @classproperty
    def message(cls):
        return "Hello " + cls.item


print(Test.message)
```

deleter(*fdel*)

Descriptor to obtain a copy of the property with a different deleter.

getter(*fget*)

Descriptor to obtain a copy of the property with a different getter.

setter(*fset*)

Descriptor to obtain a copy of the property with a different setter.

decorator_arguments(*decorator*)

make a decorator usable with and without arguments:

The resulting decorator can be used like *@decorator* or *@decorator(*args, **kwargs)*

Inspired by https://stackoverflow.com/a/14412901/932593

Parameters

decorator (*Callable*) – The decorator that needs to be modified

Returns

The decorated function

Return type

_FlexibleDecorator

ensure_directory_exists(*folder*)

Creates a folder if it not already exists.

Parameters

folder (*str*) – path of the new folder

estimate_computation_speed(*func*, **args*, ***kwargs*)

Estimates the computation speed of a function.

Parameters
func (*callable*) – The function to call

Returns
the number of times the function can be calculated in one second. The inverse is thus the runtime in seconds per function call

Return type
float

get_array_namespace(*arr*)

Get the namespace associated with an array.

This function should be used to write general functions that can work with many backends, such as `jax`. We define this as a function, so we can overload this function for the *numba* backend, which does not yet support __array_namespace__.

Parameters
arr (`ndarray` or another array type) – The array whose namespace will be returned

Returns
A python module defining typical functions that can be applied to arrays. In many cases this will be `numpy`.

Return type
types.ModuleType

get_common_dtype(**args*)

Returns a dtype in which all arguments can be represented.

Parameters
***args** – All items (arrays, scalars, etc) to be checked

Returns: numpy.cdouble if any entry is complex, otherwise np.double

hdf_write_attributes(*hdf_path*, *attributes=None*, *raise_serialization_error=False*)

Write (JSON-serialized) attributes to a hdf file.

Parameters

- **hdf_path** – Path to a group or dataset in an open HDF file
- **attributes** (*dict*) – Dictionary with values written as attributes
- **raise_serialization_error** (*bool*) – Flag indicating whether serialization errors are raised or silently ignored

Return type
None

class hybridmethod(*fclass*, *finstance=None*, *doc=None*)

Bases: `object`

Descriptor that can be used as a decorator to allow calling a method both as a classmethod and an instance method.

Adapted from https://stackoverflow.com/a/28238047

classmethod(*fclass*)

instancemethod(*finstance*)

import_class(*identifier*)

Import a class or module given an identifier.

Parameters

identifier (*str*) – The identifier can be a module or a class. For instance, calling the function with the string *identifier == 'numpy.linalg.norm'* is roughly equivalent to running *from numpy.linalg import norm* and would return a reference to *norm*.

module_available(*module_name*, *, *cache=True*, *strict=False*)

Check whether a python module is available.

Parameters

- **module_name** (*str*) – The name of the module to search
- **cache** (*bool*) – Flag determining whether an internal cache is used to speed up the check
- **strict** (*bool*) – If True, we actually try to import the full package

Returns

True if the module can be imported and *False* otherwise

Return type

bool

number(*value*)

Convert a value into a float or complex number.

Parameters

value (*Number or str*) – The value which needs to be converted

Return type

Number

Result:

Number: A complex number or a float if the imaginary part vanishes

number_array(*data*, *dtype=None*, *copy=None*)

Convert data into an array, assuming float numbers if no dtype is given.

Parameters

- **data** (ndarray) – The data that needs to be converted to a number array. This can also be any iterable of numbers.
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be double unless *data* contains complex numbers in which case it will be cdouble.
- **copy** (*bool*) – Whether the data must be copied (in which case the original array is left untouched). The default *None* implies that data is only copied if necessary, e.g., when changing the dtype.

Returns

An array with the correct dtype

Return type

ndarray

4.7.9 pde.tools.modelrunner module

Establishes hooks for the interplay between *pde* and modelrunner

This package defines a function for hook registration, which is usually called automatically during import if modelrunner is available. In this case, grids and fields of *pde* can be directly written to storages from modelrunner.storage

register_modelrunner_hooks()

Register *modelrunner* hooks.

Return type
None

4.7.10 pde.tools.mpi module

Auxiliary functions and variables for dealing with MPI multiprocessing.

Warning

These functions are mostly no-ops unless MPI is properly installed and python code was started using mpirun or mpiexec. Please refer to the documentation of your MPI distribution for details.

mpi_send	Send data to another MPI node.
mpi_recv	Receive data from another MPI node.
mpi_allreduce	Combines data from all MPI nodes.

initialized: bool = True

Flag determining whether mpi was initialized (and is available)

Type
bool

is_main: bool = True

Flag indicating whether the current process is the main process (with ID 0)

Type
bool

mpi_allreduce(*data*, *operator*)

Combines data from all MPI nodes.

Note that complex datatypes and user-defined reduction operators are not properly supported in numba-compiled cases.

Parameters

- **data** – Data being send from this node to all others
- **operator** (*int | str*) – The operator used to combine all data. Possible options are summarized in MPIOperator.

Returns
The accumulated data

mpi_bcast(*data*, *root=0*)

Broadcast data from root node to all other MPI nodes.

Parameters

- **data** (*T*) – The data being sent
- **root** (*int*) – The ID of the sending node

Return type
T

mpi_excepthook(*exc_type*, *exc_value*, *exc_tb*)

Print uncaught exceptions with rank information and abort the MPI job.

This function is intended to be assigned to `sys.excepthook` in MPI runs. It formats the traceback on the local rank, writes the error to stderr, flushes stdio streams, and then calls `MPI.Comm.Abort()` to terminate all ranks.

Parameters

- **exc_type** – Exception class of the uncaught exception.
- **exc_value** – Exception instance.
- **exc_tb** – Traceback object of the uncaught exception.

mpi_recv(*data*, *source*, *tag*)

Receive data from another MPI node.

Parameters

- **data** (*NumericArray*) – A buffer into which the received data is written
- **source** (*int*) – The ID of the sending node
- **tag** (*int*) – A numeric tag identifying the message

Return type
None

mpi_send(*data*, *dest*, *tag*)

Send data to another MPI node.

Parameters

- **data** (*NumericArray*) – The data being send
- **dest** (*int*) – The ID of the receiving node
- **tag** (*int*) – A numeric tag identifying the message

Return type
None

parallel_run: bool = False

Flag indicating whether the current run is using multiprocessing

Type
bool

rank: int = 0

ID of the current process

Type
int

size: int = 1

Total process count

Type
int

4.7.11 pde.tools.nested_dict module

Provides a nested dictionary that stores hierarchical mappings.

NestedDict	Stores hierarchical mappings with string paths as keys.

class NestedDict(*data=None*)

Bases: MutableMapping[str, TValue | NestedDict[TValue]], Generic[TValue]

Stores hierarchical mappings with string paths as keys.

NestedDict wraps nested mappings and supports reading and writing nested values using a separator-based key syntax (for example `"a.b.c"`). It can convert between flat and nested representations and recursively traverses children when requested.

Note

Equivalent entries can overwrite each other during initialization. For instance, `NestedDict({'a.b': 1, 'a': {'b': 2}})` stores only one final value for `a.b`.

Initializes a nested dictionary from an optional mapping.

Parameters
data (*MutableMapping[str, Any] | None*) – Optional mapping used to populate the instance. Nested plain dictionaries are converted into *NestedDict* children.

clear()

Removes all top-level entries from the mapping.

Return type
None

copy()

Creates a structural copy with copied nested mappings.

Child dictionaries and child *NestedDict* instances are copied, while non-mapping leaf values are reused by reference.

Returns
New instance containing copied nested structure.

Return type
NestedDict

create_node(*key*)

Create an empty node at the given location.

Creates all necessary parent nodes recursively. Skips nodes that already exist.

Parameters
key (*str*) – Key or nested key path identifying the node to create.

Returns
The leaf node

Return type
Self

data: MutableMapping[str, TValue | NestedDict[TValue]]
Internal mapping storing top-level keys and values for this instance.

Type
dict

items(*, *flatten: Literal[False] = False*) → Iterator[tuple[str, TValue | *NestedDict*[TValue]]]

items(*, *flatten: Literal[True]*) → Iterator[tuple[str, TValue]]

Iterates over key-value pairs, optionally flattening nested paths.

Parameters
flatten (*bool*) – If *True*, yields *(path, value)* pairs for all descendants. If *False*, yields only top-level pairs.

Returns
Iterator over key-value pairs according to *flatten*.

Return type
Iterator[tuple[str, Any]]

Raises
TypeError – If a key used during flattening is not a string.

keys(*, *flatten=False*)

Iterates over keys, optionally returning flattened key paths.

Parameters
flatten (*bool*) – If *True*, yields separator-joined paths for descendant keys. If *False*, yields only top-level keys.

Returns
Iterator over keys or flattened key paths.

Return type
Iterator[str]

Raises
TypeError – If a key used during flattening is not a string.

pprint(**args*, *flatten=False*, ***kwargs*)

Pretty-prints the current data as nested plain dictionaries.

Parameters

- ***args** (*Any*) – Positional arguments forwarded to *pprint.pprint*.
- **flatten** (*bool*) – If *True*, returns a flat mapping with separator-joined paths as keys. If *False*, returns nested dictionaries.
- ****kwargs** (*Any*) – Keyword arguments forwarded to *pprint.pprint*.

Return type
None

sep: str = '.'
Separator used in key paths to traverse nested levels.

Type
str

to_dict(*, *flatten: Literal[False] = False*) → dict[str, TValue | TDictTree]

to_dict(*, *flatten: Literal[True]*) → dict[str, TValue]

Converts this object to a plain dictionary representation.

Parameters
flatten (*bool*) – If *True*, returns a flat mapping with separator-joined paths as keys. If *False*, returns nested dictionaries.

Returns
Dictionary representation of this object.

Return type
dict[str, Any]

Raises
TypeError – If flattening encounters non-string keys.

update(*other*)

Update this mapping from another mapping recursively.

This method implements `collections.abc.MutableMapping` update semantics for mapping-like inputs and forwards the actual merge to *update_recursive()*.

Parameters
other – Mapping containing keys and values to merge into this instance.

Raises
TypeError – If *other* is not a mutable mapping.

Return type
None

update_recursive(*other*)

Recursively merges another mapping into this instance.

Parameters
other (*MutableMapping[str, Any]*) – Mapping whose entries are merged into this object. If both sides contain nested mappings at a key, values are merged recursively.

Return type
None

values(*, *flatten: Literal[False] = False*) → Iterator[TValue | *NestedDict*[TValue]]

values(*, *flatten: Literal[True]*) → Iterator[TValue]

Iterates over values, optionally recursing into nested children.

Parameters
flatten (*bool*) – If *True*, yields values from all descendant *NestedDict* instances. If *False*, yields only top-level values.

Returns
Iterator over values according to *flatten*.

Return type
Iterator[Any]

4.7.12 pde.tools.numba module

Helper functions for just-in-time compilation with numba.

numba_environment	Return information about the numba setup used.
jit	Apply nb.jit with predefined arguments.
make_array_constructor	Returns an array within a jitted function using basic information.
numba_dict	Converts a python dictionary to a numba typed dictionary.
get_common_numba_dtype	Returns a numba numerical type in which all arrays can be represented.
random_seed	Sets the seed of the random number generator of numpy and numba.

4.7.13 pde.tools.output module

Python functions for handling output.

get_progress_bar_class	Returns a class that behaves as progress bar.
display_progress	Displays a progress bar when iterating.
in_jupyter_notebook	Checks whether we are in a jupyter notebook.
BasicOutput	Class that writes text line to stdout.
JupyterOutput	Class that writes text lines as html in a jupyter cell.

class BasicOutput(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *OutputBase*

Class that writes text line to stdout.

Parameters

stream – The stream where the lines are written

show()

Shows the actual text.

class JupyterOutput(*header=''*, *footer=''*)

Bases: *OutputBase*

Class that writes text lines as html in a jupyter cell.

Parameters

- **header** (*str*) – The html code written before all lines
- **footer** (*str*) – The html code written after all lines

show()

Shows the actual text.

class OutputBase

Bases: object

Base class for output management.

abstractmethod show()

Shows the actual text.

display_progress(*iterator*, *total=None*, *enabled=True*, ***kwargs*)

Displays a progress bar when iterating.

Parameters

- **iterator** (*iter*) – The iterator
- **total** (*int*) – Total number of steps
- **enabled** (*bool*) – Flag determining whether the progress is display
- ****kwargs** – All extra arguments are forwarded to the progress bar class

Returns

A class that behaves as the original iterator, but shows the progress alongside iteration.

get_progress_bar_class(*fancy=True*)

Returns a class that behaves as progress bar.

This either uses classes from the optional *tqdm* package or a simple version that writes dots to stderr, if the class it not available.

Parameters

fancy (*bool*) – Flag determining whether a fancy progress bar should be used in jupyter notebooks (if `ipywidgets` is installed)

in_jupyter_notebook()

Checks whether we are in a jupyter notebook.

Return type

bool

4.7.14 pde.tools.parse_duration module

Parsing time durations from strings.

This module provides a function that parses time durations from strings. It has been copied from the django software, which comes with the following notes:

Copyright (c) Django Software Foundation and individual contributors. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Django nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

parse_duration(*value*)

Parse a duration string and return a datetime.timedelta.

Parameters

value (*str*) – A time duration given as text. The preferred format for durations is '%d %H:%M:%S.%f'. This function also supports ISO 8601 representation and PostgreSQL's day-time interval format.

Returns

An instance representing the duration.

Return type

datetime.timedelta

4.7.15 pde.tools.plotting module

Tools for plotting and controlling plot output using context managers.

add_scaled_colorbar	Add a vertical color bar to an image plot.
disable_interactive	Context manager disabling the interactive mode of matplotlib.
plot_on_axes	Decorator for a plot method or function that uses a single axes.
plot_on_figure	Decorator for a plot method or function that fills an entire figure.
PlotReference	Contains all information to update a plot element.
BasicPlottingContext	Basic plotting using just matplotlib.
JupyterPlottingContext	Plotting in a jupyter widget using the *inline* backend.
get_plotting_context	Returns a suitable plotting context.
napari_add_layers	adds layers to a napari viewer

class BasicPlottingContext(*fig_or_ax=None*, *title=None*, *show=True*)

Bases: *PlottingContextBase*

Basic plotting using just matplotlib.

Parameters

- **fig_or_ax** – If axes are given, they are used. If a figure is given, it is set as active.
- **title** (*str*) – The title shown in the plot
- **show** (*bool*) – Flag determining whether plots are actually shown

class JupyterPlottingContext(*title=None*, *show=True*)

Bases: *PlottingContextBase*

Plotting in a jupyter widget using the *inline* backend.

Parameters

- **title** (*str*) – The shown in the plot
- **show** (*bool*) – Flag determining whether plots are actually shown

close()

Close the plot.

supports_update = False

Flag indicating whether the context supports that plots can be updated with out redrawing the entire plot.

The jupyter backend (*inline*) requires replotting of the entire figure, so an update is not supported.

class PlotReference(*ax*, *element*, *parameters=None*)

Bases: object

Contains all information to update a plot element.

Parameters

- **ax** (matplotlib.axes.Axes) – The axes of the element
- **element** (matplotlib.artist.Artist) – The actual element
- **parameters** (*dict*) – Parameters to recreate the plot element

ax

element

parameters

class PlottingContextBase(*title=None*, *show=True*)

Bases: object

Base class of the plotting contexts.

Example

The context wraps calls to the matplotlib.pyplot interface:

```
context = PlottingContext()
with context:
    plt.plot(...)
    plt.xlabel(...)
```

Parameters

- **title** (*str*) – The shown in the plot
- **show** (*bool*) – Flag determining whether plots are actually shown

close()

Close the plot.

fig: mpl_figure.Figure | None

supports_update: bool = True

Flag indicating whether the context supports that plots can be updated with out redrawing the entire plot.

add_scaled_colorbar(*axes_image*, *ax=None*, *aspect=20*, *pad_fraction=0.5*, *label=''*, ***kwargs*)

Add a vertical color bar to an image plot.

The height of the colorbar is now adjusted to the plot, so that the width determined by *aspect* is now given relative to the height. Moreover, the gap between the colorbar and the plot is now given in units of the fraction of the width by *pad_fraction*.

Inspired by https://stackoverflow.com/a/33505522/932593

Parameters

- **axes_image** (matplotlib.cm.ScalarMappable) – Mappable object, e.g., returned from matplotlib.pyplot.imshow()
- **ax** (matplotlib.axes.Axes) – The current figure axes from which space is taken for the colorbar. If omitted, the axes in which the *axes_image* is shown is taken.
- **aspect** (*float*) – The target aspect ratio of the colorbar
- **pad_fraction** (*float*) – Width of the gap between colorbar and image
- **label** (*str*) – Set a label for the colorbar
- ****kwargs** – Additional parameters are passed to colorbar call

Returns
The resulting Colorbar object

Return type
Colorbar

disable_interactive()

Context manager disabling the interactive mode of matplotlib.

This context manager restores the previous state after it is done. Details of the interactive mode are described in matplotlib.interactive().

get_plotting_context(*context=None*, *title=None*, *show=True*)

Returns a suitable plotting context.

Parameters

- **context** – An instance of *PlottingContextBase* or an instance of matplotlib.axes.Axes or matplotlib.figure.Figure to determine where the plotting will happen. If omitted, the context is determined automatically.
- **title** (*str*) – The title shown in the plot
- **show** (*bool*) – Determines whether the plot is shown while the simulation is running. If *False*, the files are created in the background.

Returns
The plotting context

Return type
PlottingContextBase

in_ipython()

Try to detect whether we are in an ipython shell, e.g., a jupyter notebook.

Return type
bool

napari_add_layers(*viewer*, *layers_data*)

adds layers to a napari viewer

Parameters

- **viewer** (napari.viewer.Viewer) – The napari application
- **layers_data** (*dict*) – Data for all layers that will be added.

napari_viewer(*grid*, *run=None*, *close=False*, ***kwargs*)

Creates an napari viewer for interactive plotting.

Parameters

- **grid** (*pde.grids.base.GridBase*) – The grid defining the space
- **run** (*bool*) – Whether to run the event loop of napari.
- **close** (*bool*) – Whether to close the viewer immediately (e.g. for testing)
- ****kwargs** – Extra arguments are passed to `napari.Viewer`

Return type

Generator[napari.viewer.Viewer]

class nested_plotting_check

Bases: `object`

Context manager that checks whether it is the root plotting call.

Example

The context manager can be used in plotting calls to check for nested plotting calls:

```
with nested_plotting_check() as is_outermost_plot_call:
    make_plot(...)  # could potentially call other plotting methods
    if is_outermost_plot_call:
        plt.show()
```

plot_on_axes(*wrapped=None*, *update_method=None*)

Decorator for a plot method or function that uses a single axes.

This decorator adds typical options for creating plots that fill a single axes. These options are available via keyword arguments. To avoid redundancy in describing these options in the docstring, the placeholder *{PLOT_ARGS}* can be added to the docstring of the wrapped function or method and will be replaced by the appropriate text. Note that the decorator can be used on both functions and methods.

Example

The following example illustrates how this decorator can be used to implement plotting for a given class. In particular, supplying the *update_method* will allow efficient dynamical plotting:

```
class State:
    def __init__(self) -> None:
        self.data = np.arange(8)

    def _update_plot(self, reference):
        reference.element.set_ydata(self.data)

    @plot_on_axes(update_method="_update_plot")
    def plot(self, ax):
        (line,) = ax.plot(np.arange(8), self.data)
        return PlotReference(ax, line)
```

```
@plot_on_axes
def make_plot(ax):
    ax.plot(...)
```

When *update_method* is absent, the method can still be used for plotting, but dynamic updating, e.g., by `pde.trackers.PlotTracker`, is not possible.

Parameters

- **wrapped** (*callable*) – Function to be wrapped
- **update_method** (*callable or str*) – Method to call to update the plot. The argument of the new method will be the result of the initial call of the wrapped method.

plot_on_figure(*wrapped=None*, *update_method=None*)

Decorator for a plot method or function that fills an entire figure.

This decorator adds typical options for creating plots that fill an entire figure. This decorator adds typical options for creating plots that fill a single axes. These options are available via keyword arguments. To avoid redundancy in describing these options in the docstring, the placeholder *{PLOT_ARGS}* can be added to the docstring of the wrapped function or method and will be replaced by the appropriate text. Note that the decorator can be used on both functions and methods.

Example

The following example illustrates how this decorator can be used to implement plotting for a given class. In particular, supplying the *update_method* will allow efficient dynamical plotting:

```
class State:
    def __init__(self) -> None:
        self.data = np.random.random((2, 8))

    def _update_plot(self, reference):
        ref1, ref2 = reference
        ref1.element.set_ydata(self.data[0])
        ref2.element.set_ydata(self.data[1])

    @plot_on_figure(update_method="_update_plot")
    def plot(self, fig):
        ax1, ax2 = fig.subplots(1, 2)
        (l1,) = ax1.plot(np.arange(8), self.data[0])
        (l2,) = ax2.plot(np.arange(8), self.data[1])
        return [PlotReference(ax1, l1), PlotReference(ax2, l2)]


@plot_on_figure
def make_plot(fig): ...
```

When *update_method* is not supplied, the method can still be used for plotting, but dynamic updating, e.g., by `pde.trackers.PlotTracker`, is not possible.

Parameters

- **wrapped** (*callable*) – Function to be wrapped
- **update_method** (*callable or str*) – Method to call to update the plot. The argument of the new method will be the result of the initial call of the wrapped method.

4.7.16 pde.tools.spectral module

Functions making use of spectral decompositions.

make_colored_noise	Return a function creating an array of random values that obey.
make_correlated_noise	Return a function creating random values with specified spatial correlations.

make_colored_noise(*shape*, *dx=1.0*, *exponent=0*, *scale=1*, *rng=None*)

Return a function creating an array of random values that obey.

$$\langle c(\boldsymbol{k}) c(\boldsymbol{k}') \rangle = \Gamma^2 |\boldsymbol{k}|^\nu \delta(\boldsymbol{k} - \boldsymbol{k}')$$

in spectral space on a Cartesian grid. The special case $\nu = 0$ corresponds to white noise. For simplicity, the correlations respect periodic boundary conditions.

Parameters

- **shape** (*tuple of ints*) – Number of supports points in each spatial dimension. The number of the list defines the spatial dimension.
- **dx** (*float or list of floats*) – Discretization along each dimension. A uniform discretization in each direction can be indicated by a single number.
- **exponent** (*float*) – Exponent ν of the power spectrum
- **scale** (*float*) – Scaling factor Γ determining noise strength
- **rng** (*Generator*) – Random number generator (default: `default_rng()`)

Returns

a function returning a random realization

Return type

callable

make_correlated_noise(*shape*, *correlation*, *, *discretization=1.0*, *dtype=<class 'float'>*, *rng=None*, ***kwargs*)

Return a function creating random values with specified spatial correlations.

The returned field f generally obeys a selected correlation function $C(k)$. In Fourier space, we thus have

$$\langle f(\boldsymbol{k}) f(\boldsymbol{k}') \rangle = C(|\boldsymbol{k}|) \delta(\boldsymbol{k} - \boldsymbol{k}')$$

For simplicity, the correlations respect periodic boundary conditions.

Parameters

- **shape** (*tuple of ints*) – Number of supports points in each spatial dimension. The number of the list defines the spatial dimension.
- **correlation** (*str*) – Selects the correlation function used to make the correlated noise. Many of the options (described below) support additional parameters that can be supplied as keyword arguments.

- **discretization** (*float or list of floats*) – Discretization along each dimension. A uniform discretization in each direction can be indicated by a single number.
- **dtype** (numpy.dtype) – Data type of the returned noise array. If a complex dtype is provided, the function returns complex-valued arrays; otherwise the real part is taken and returned as the specified real dtype.
- **rng** (Generator) – Random number generator (default: default_rng())
- ****kwargs** – Additional parameters can affect details of the correlation function

Return type
Callable[[], NumericArray]

Table 69: Supported correlation functions

Identifier	Correlation function
none	No correlation, $C(k) = \delta(k)$
gaussian	$C(k) = \exp(\frac{1}{2}k^2\lambda^2)$ with the length scale λ set by argument length_scale.
power law	$C(k) = k^{\nu/2}$ with exponent ν set by argument exponent.
cosine	$C(k) = \exp\bigl(-s^2(\lambda k - 1)^2\bigr)$ with the length scale λ set by argument length_scale, whereas the sharpness parameter s is set by sharpness and defaults to 10.

Note

The returned field only has unit variance for correlation functions that decrease monotonously. In other cases (i.e., for cosine correlation), the variance depends on details, like the resolution of the grid.

Returns
a function returning a random realization

Return type
callable

Parameters

- **shape** (*tuple[int, ...]*)
- **correlation** (*CorrelationType*)
- **discretization** (*NumberOrArray*)
- **dtype** (*np.typing.DTypeLike*)
- **rng** (*np.random.Generator | None*)

4.7.17 pde.tools.typing module

Provides support for mypy type checking of the package.

OperatorInfo	Stores information about an operator.
OperatorImplType	Represent a PEP 604 union type
OperatorFactory	A factory function that creates an operator for a particular grid.
OperatorType	An operator that acts on an array.
CellVolume	

continues on next page

Table 70 – continued from previous page

VirtualPointEvaluator
GhostCellSetter
DataSetter
StepperHook

class CellVolume(**args*, ***kwargs*)

Bases: Protocol

class DataSetter(**args*, ***kwargs*)

Bases: Protocol

class GhostCellSetter(**args*, ***kwargs*)

Bases: Protocol

class InnerStepperType(**args*, ***kwargs*)

Bases: Protocol

General backend-level stepping-function type working with numpy arrays.

class OperatorFactory(**args*, ***kwargs*)

Bases: Protocol

A factory function that creates an operator for a particular grid.

class OperatorInfo(*factory*, *rank_in*, *rank_out*, *name=''*)

Bases: NamedTuple

Stores information about an operator.

Create new instance of OperatorInfo(factory, rank_in, rank_out, name)

Parameters

- **factory** (OperatorFactory)
- **rank_in** (*int*)
- **rank_out** (*int*)
- **name** (*str*)

factory: *OperatorFactory*

Alias for field number 0

name: **str**

Alias for field number 3

rank_in: **int**

Alias for field number 1

rank_out: **int**

Alias for field number 2

class OperatorType(**args*, ***kwargs*)

Bases: Protocol, Generic[TNativeArray]

An operator that acts on an array.

class PostStepHook(**args*, ***kwargs*)

Bases: Protocol, Generic[TNativeArray]

class StepperHook(**args*, ***kwargs*)

Bases: Protocol, Generic[TNativeArray]

class StepperType(**args*, ***kwargs*)

Bases: Protocol

General stepping-function type working with py-pde fields.

Instances of this protocol are typically created by solver objects.

class VirtualPointEvaluator(**args*, ***kwargs*)

Bases: Protocol

4.8 pde.trackers package

Classes for tracking simulation results in controlled interrupts.

Trackers are classes that periodically receive the state of the simulation to analyze, store, or output it. The trackers defined in this module are:

CallbackTracker	Tracker calling a function periodically.
ConsistencyTracker	Tracker interrupting the simulation when the state is not finite.
DataTracker	Tracker storing custom data obtained by calling a function.
LivePlotTracker	PlotTracker with defaults for live plotting.
MaterialConservationTracker	Tracking interrupting the simulation when material conservation is broken.
MaxRuntimeTracker	Tracker interrupting the simulation once a duration has passed.
PlotTracker	Tracker plotting data on screen, to files, or writes a movie.
PrintTracker	Tracker printing data to a stream (default: stdout)
ProgressTracker	Tracker showing the progress of the simulation.
SteadyStateTracker	Tracker aborting the simulation once steady state is reached.
WalltimeTracker	Special tracker that records runtime of the simulation at regular intervals.
InteractivePlotTracker	Tracker showing the state interactively in napari.

Some trackers can be referenced by name for convenience when using them in simulations. The list of supported names is returned by `registered_trackers()`.

Multiple trackers can be collected in a *TrackerCollection*, which provides methods for handling them efficiently. Moreover, custom trackers can be implemented by deriving from *TrackerBase*. Note that trackers generally receive a view into the current state, implying that they can adjust the state by modifying it in-place. Moreover, trackers can abort the simulation by raising the special exception StopIteration.

For each tracker, the time at which the simulation is interrupted can be decided using one of the following classes:

FixedInterrupts	Interrupts at fixed, predetermined times.
ConstantInterrupts	Interrupts equidistantly spaced in time.
LogarithmicInterrupts	Interrupts with successively increased spacing.
GeometricInterrupts	Interrupts from the geometric sequence $t_i = \Delta t f^i$

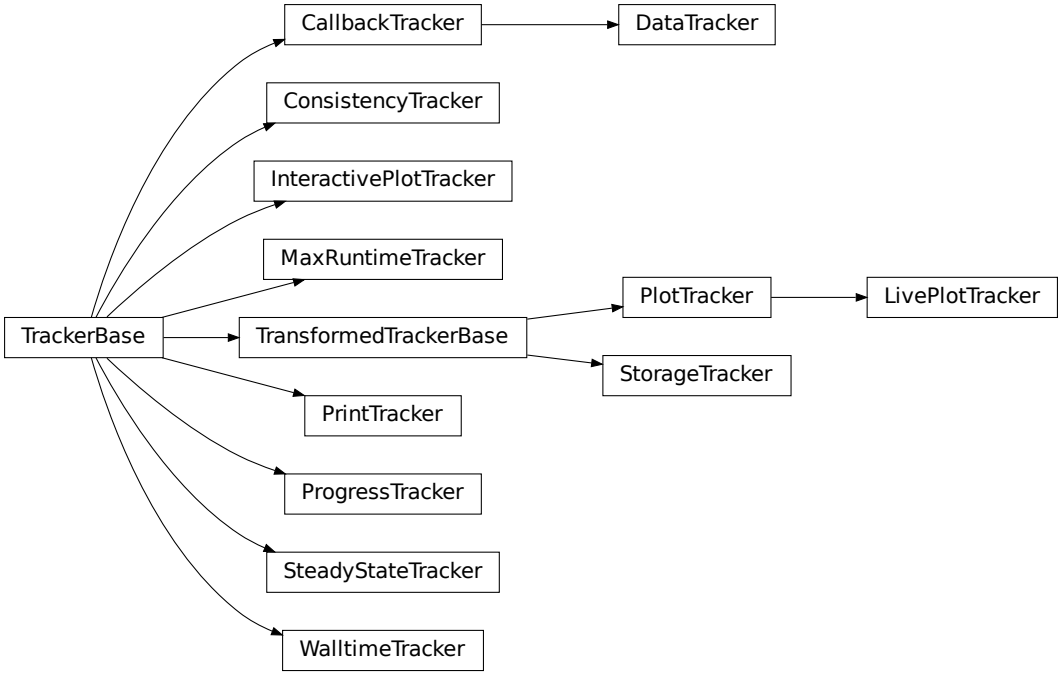
continues on next page

Table 72 – continued from previous page

RealtimeInterrupts	Interrupts spaced equidistantly in real time.

In particular, interrupts can be specified conveniently using *parse_interrupt()*.

Inheritance structure of the tracker classes:



4.8.1 pde.trackers.base module

Base classes for trackers.

TrackerBase	Base class for implementing trackers.
TrackerCollection	List of trackers providing methods to handle them efficiently.
TransformedTrackerBase	Tracker that allows modifying incoming data.
FinishedSimulation	Exception for signaling that simulation finished successfully.

exception FinishedSimulation

Bases: StopIteration

Exception for signaling that simulation finished successfully.

class TrackerBase(*interrupts=1*)

Bases: object

Base class for implementing trackers.

Parameters

interrupts (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters

info (*dict*) – Extra information from the simulation

Return type

None

classmethod from_data(*data*, ***kwargs*)

Create tracker class from given data.

Parameters

- **data** (*str or TrackerBase*) – Data describing the tracker
- ****kwargs** – Additional keyword arguments passed to the tracker constructor

Returns

An instance representing the tracker

Return type

TrackerBase

abstractmethod handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type

None

initialize(*field*, *info=None*)

Initialize the tracker with information about the simulation.

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns

The first time the tracker needs to handle data

Return type

float

class TrackerCollection(*trackers=None*)

Bases: object

List of trackers providing methods to handle them efficiently.

trackers

List of the trackers in the collection

Type
list

Parameters
trackers (*list[TrackerBase]*) – List of trackers that are to be handled.

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters
info (*dict*) – Extra information from the simulation

Return type
None

classmethod from_data(*data*, ***kwargs*)

Create tracker collection from given data.

Parameters

- **data** (*Sequence[TrackerBase | str] | TrackerBase | str | None*) – Data describing the tracker collection
- ****kwargs** – Additional keyword arguments passed to tracker constructors

Returns
An instance representing the tracker collection

Return type
TrackerCollection

handle(*state*, *t*, *atol=1e-08*)

Handle all trackers.

Parameters

- **state** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time
- **atol** (*float*) – An absolute tolerance that is used to determine whether a tracker should be called now or whether the simulation should be carried on more timesteps. This is basically used to predict the next time to decided which one is closer.

Returns
The next time the simulation needs to be interrupted to handle a tracker.

Return type
float

initialize(*field*, *info=None*)

Initialize the tracker with information about the simulation.

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

time_next_action: float
The time of the next interrupt of the simulation

Type
float

tracker_action_times: list[float]
Times at which the trackers need to be handled next

Type
list

class TransformedTrackerBase(*interrupts=1*, *, *transformation=None*)

Bases: *TrackerBase*

Tracker that allows modifying incoming data.

To support the transformations sub-classes need to call `self._transform(field, t)` to obtain the transformed field.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then used in the tracker. This allows to process derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

registered_trackers()

Returns all trackers that are currently registered.

Returns
a dictionary with the names of the trackers and the associated class

Return type
dict

4.8.2 pde.trackers.interactive module

Special module for defining an interactive tracker that uses napari to display fields.

class InteractivePlotTracker(*interrupts='0:01'*, *, *close=True*, *show_time=False*)

Bases: *TrackerBase*

Tracker showing the state interactively in napari.

Note

The interactive tracker uses the python `multiprocessing` module to run napari externally. The multiprocessing module has limitations on some platforms, which requires some care when writing your own programs. In particular, the main method needs to be safe-guarded so that the main module can be imported again after spawning a new process. An established pattern that works is to introduce a function *main* in your code, which you call using the following pattern

```
def main():
    # here goes your main code

if __name__ == "__main__":
    main()
```

The last two lines ensure that the *main* function is only called when the module is run initially and not again when it is re-imported.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **close** (*bool*) – Flag indicating whether the napari window is closed automatically at the end of the simulation. If *False*, the tracker blocks when *finalize* is called until the user closes napari manually.
- **show_time** (*bool*) – Whether to indicate the time

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters
info (*dict*) – Extra information from the simulation

Return type
None

handle(*state*, *t*)

Handle data supplied to this tracker.

Parameters

- **state** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

initialize(*state*, *info=None*)

Initialize the tracker with information about the simulation.

Parameters

- **state** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

name = 'interactive'

class NapariViewer(*state*, *t_initial=None*)

Bases: object

Allows viewing and updating data in a separate napari process.

Parameters

- **state** (*pde.fields.base.FieldBase*) – The initial state to be shown
- **t_initial** (*float*) – The initial time. If *None*, no time will be shown.

close(*force=True*)

Closes the napari process.

Parameters
force (*bool*) – Whether to force closing of the napari program. If this is *False*, this method blocks until the user closes napari manually.

update(*state*, *t*)

Update the state in the napari viewer.

Parameters

- **state** (*pde.fields.base.FieldBase*) – The new state
- **t** (*float*) – Current time

napari_process(*data_channel*, *initial_data*, *t_initial=None*, *viewer_args=None*)

multiprocessing.Process running napari120

Parameters

- **data_channel** (multiprocessing.Queue) – queue instance to receive data to view
- **initial_data** (*dict*) – Initial data to be shown by napari. The layers are named according to the keys in the dictionary. The associated value needs to be a tuple, where the first item is a string indicating the type of the layer and the second carries the associated data
- **t_initial** (*float*) – Initial time
- **viewer_args** (*dict*) – Additional arguments passed to the napari viewer

4.8.3 pde.trackers.interrupts module

Module defining classes for time interrupts for trackers.

The provided interrupt classes are:

ConstantInterrupts	Interrupts equidistantly spaced in time.
FixedInterrupts	Interrupts at fixed, predetermined times.
LogarithmicInterrupts	Interrupts with successively increased spacing.
GeometricInterrupts	Interrupts from the geometric sequence $t_i = \Delta t f^i$
RealtimeInterrupts	Interrupts spaced equidistantly in real time.

continues on next page

Table 74 – continued from previous page

parse_interrupt	Create interrupt class from various data formats.

class ConstantInterrupts(*dt=1*, *t_start=None*)

Bases: InterruptsBase

Interrupts equidistantly spaced in time.

Parameters

- **dt** (*float*) – The duration between subsequent interrupts. This is measured in simulation time units.
- **t_start** (*float, optional*) – The time after which the tracker becomes active. If omitted, the tracker starts recording right away. This argument can be used for an initial equilibration period during which no data is recorded.

initialize(*t*)

Initialize the interrupt class.

Parameters
t (*float*) – The starting time of the simulation

Returns
The first time the simulation needs to be interrupted

Return type
float

next(*t*)

Computes the next time point.

Parameters
t (*float*) – The current time point of the simulation. The returned next time point lies later than this time, so interrupts might be skipped.

Returns
The next time point

Return type
float

class FixedInterrupts(*interrupts*)

Bases: InterruptsBase

Interrupts at fixed, predetermined times.

Parameters
interrupts (*sequence of float*) – A sequence of time points at which interrupts occur

copy()

initialize(*t*)

Initialize the interrupt class.

Parameters
t (*float*) – The starting time of the simulation

Returns
The first time the simulation needs to be interrupted

Return type
float

next(*t*)

Computes the next time point.

Parameters
t (*float*) – The current time point of the simulation. The returned next time point lies later than this time, so interrupts might be skipped.

Returns
The next time point

Return type
float

class GeometricInterrupts(*scale*, *factor*)

Bases: InterruptsBase

Interrupts from the geometric sequence $t_i = \Delta t f^i$

In contrast to *LogarithmicInterrupts*, this class ensures that time points lie on the geometric sequence given above. However, data points might be skipped if the simulations progress too quickly.

Parameters

- **scale** (*float*) – Time scale Δt.
- **factor** (*float*) – Scale factor f.

initialize(*t*)

Initialize the interrupt class.

Parameters
t (*float*) – The starting time of the simulation

Returns
The first time the simulation needs to be interrupted

Return type
float

next(*t*)

Computes the next time point.

Parameters
t (*float*) – The current time point of the simulation. The returned next time point lies later than this time, so interrupts might be skipped.

Returns
The next time point

Return type
float

value(*iteration*)

Calculate value of i-th interrupt.

Parameters
iteration (*int*) – The iteration of the interrupt

Returns
time of the i-th interrupt

Return type
float

class LogarithmicInterrupts(*dt_initial=1*, *factor=1*, *t_start=None*)

Bases: *ConstantInterrupts*

Interrupts with successively increased spacing.

The durations between interrupts increases by a constant factor f:

$$t_{i+1} = t_i + \Delta t_i \qquad \text{with} \qquad \Delta t_{i+1} = f \Delta t_i$$

starting with initial values t_0 and Δt_0. This results in exponentially spaced interrupts $t_i = a + bf^i$, where $a = t_0 - \Delta t_0 (f-1)^{-1}$ and $b = \Delta t_0 (f-1)^{-1}$.

Note that the geometric sequence described above can be disrupted if other interrupts interfere. This class ensures ever increasing durations between its interrupts, at the cost of potentially oddly spaced times. If a geometric sequence is required, use *GeometricInterrupts* instead.

Parameters

- **dt_initial** (*float*) – The initial duration Δt_0 between subsequent interrupts. This is measured in simulation time units.
- **factor** (*float*) – The factor f by which the time between interrupts is increased after every interrupt. Values larger than one lead to time interrupts that are increasingly further apart.
- **t_start** (*float, optional*) – The time t_0 after which the tracker becomes active. If omitted, the tracker starts recording right away. This argument can be used for an initial equilibration period during which no data is recorded.

next(*t*)

Computes the next time point.

Parameters
t (*float*) – The current time point of the simulation. The returned next time point lies later than this time, so interrupts might be skipped.

Returns
The next time point

Return type
float

value(*iteration*)

Calculate value of i-th interrupt.

Parameters
iteration (*int*) – The iteration of the interrupt

Returns
time of the i-th interrupt

Return type
float

class RealtimeInterrupts(*duration*, *dt_initial=0.01*)

Bases: *ConstantInterrupts*

Interrupts spaced equidistantly in real time.

This spacing is only achieved approximately and depends on the initial value set by *dt_initial* and the actual variation in computation speed.

Parameters

- **duration** (*float or str*) – The duration (in real seconds) that the interrupts should be spaced apart. The duration can also be given as a string, which is then parsed using the function *parse_duration()*.
- **dt_initial** (*float*) – The initial duration between subsequent interrupts. This is measured in simulation time units.

initialize(*t*)

Initialize the interrupt class.

Parameters
t (*float*) – The starting time of the simulation

Returns
The first time the simulation needs to be interrupted

Return type
float

next(*t*)

Computes the next time point.

Parameters
t (*float*) – The current time point of the simulation. The returned next time point lies later than this time, so interrupts might be skipped.

Returns
The next time point

Return type
float

parse_interrupt(*data*)

Create interrupt class from various data formats.

Parameters
data (str or number or `InterruptsBase`) – Data determining the interrupt class. If this is a `InterruptsBase`, it is simply returned, numbers imply *ConstantInterrupts*, a string is generally parsed as a time for *RealtimeInterrupts*, and lists are interpreted as *FixedInterrupts*. Instance of *GeometricInterrupts* can be constructed with the special string `"geometric(SCALE, FACTOR)"`, specifying the *scale* and *factor* values directly as numbers.

Returns
An instance that represents the interrupt

Return type
`InterruptsBase`

4.8.4 pde.trackers.trackers module

Module defining classes for tracking results from simulations.

The trackers defined in this module are:

CallbackTracker	Tracker calling a function periodically.
ConsistencyTracker	Tracker interrupting the simulation when the state is not finite.

continues on next page

Table 75 – continued from previous page

DataTracker	Tracker storing custom data obtained by calling a function.
LivePlotTracker	PlotTracker with defaults for live plotting.
MaterialConservationTracker	Tracking interrupting the simulation when material conservation is broken.
MaxRuntimeTracker	Tracker interrupting the simulation once a duration has passed.
PlotTracker	Tracker plotting data on screen, to files, or writes a movie.
PrintTracker	Tracker printing data to a stream (default: stdout)
ProgressTracker	Tracker showing the progress of the simulation.
SteadyStateTracker	Tracker aborting the simulation once steady state is reached.
WalltimeTracker	Special tracker that records runtime of the simulation at regular intervals.

class CallbackTracker(*func*, *interrupts=1*)

Bases: *TrackerBase*

Tracker calling a function periodically.

Example

The callback tracker can be used to check for conditions during the simulation:

```
def check_simulation(state, time):
    if state.integral < 0:
        raise StopIteration


tracker = CallbackTracker(check_simulation, interrupts="0:10")
```

Adding `tracker` to the simulation will perform a check every 10 real time seconds. If the integral of the entire state falls below zero, the simulation will be aborted.

Parameters

- **func** (*Callable*) – The function to call periodically. The function signature should be *(state)* or *(state, time)*, where *state* contains the current state as an instance of *FieldBase* and *time* is a float value indicating the current time. Note that only a view of the state is supplied, implying that a copy needs to be made if the data should be stored. The function can thus adjust the state by modifying it in-place and it can even interrupt the simulation by raising the special exception `StopIteration`.
- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

class ConsistencyTracker(*interrupts=None*)

Bases: *TrackerBase*

Tracker interrupting the simulation when the state is not finite.

Parameters
interrupts (*InterruptData | None*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*. The default value *None* checks for consistency approximately every (real) second.

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

name = 'consistency'

class DataTracker(*func*, *interrupts=1*, *, *filename=None*)

Bases: *CallbackTracker*

Tracker storing custom data obtained by calling a function.

Example

The data tracker can be used to gather statistics during the run

```
def get_statistics(state, time):
    return {"mean": state.data.mean(), "variance": state.data.var()}


data_tracker = DataTracker(get_statistics, interrupts=10)
```

Adding `data_tracker` to the simulation will gather the statistics every 10 time units. After the simulation, the final result will be accessible via the *data* attribute or conveniently as a pandas from the *dataframe* attribute.

times

The time points at which the data is stored

Type
list

data

The actually stored data, which is a list of the objects returned by the callback function.

Type
list

Parameters

- **func** (*Callable*) – The function to call periodically. The function signature should be *(state)* or *(state, time)*, where *state* contains the current state as an instance of `FieldBase` and *time* is a float value indicating the current time. Note that only a view of the state is supplied, implying that a copy needs to be made if the data should be stored. Typical return values of the function are either a single number, a numpy array, a list of number, or a dictionary to return multiple numbers with assigned labels.
- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **filename** (*str*) – A path to a file to which the data is written at the end of the tracking. The data format will be determined by the extension of the filename. '.pickle' indicates a python pickle file storing a tuple *(self.times, self.data)*, whereas any other data format requires `pandas`.

data: list[Any]

property dataframe: pandas.DataFrame

the data in a dataframe.

If *func* returns a dictionary, the keys are used as column names. Otherwise, the returned data is enumerated starting with '0'. In any case the time point at which the data was recorded is stored in the column 'time'.

Type
`pandas.DataFrame`

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters
info (*dict*) – Extra information from the simulation

Return type
None

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (`FieldBase`) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

times: list[float]

to_file(*filename*, ***kwargs*)

Store data in a file.

The extension of the filename determines what format is being used. For instance, '.pickle' indicates a python pickle file storing a tuple *(self.times, self.data)*, whereas any other data format requires `pandas`. Supported formats include 'csv', 'json'.

Parameters

- **filename** (*str*) – Path where the data is stored
- ****kwargs** – Additional parameters may be supported for some formats

class LivePlotTracker(*interrupts='0:03'*, *, *show=True*, *max_fps=2*, ***kwargs*)

Bases: *PlotTracker*

PlotTracker with defaults for live plotting.

The only difference to *PlotTracker* are the changed default values, where output is by default shown on screen and the *interrupts* is set something more suitable for interactive plotting. In particular, this tracker can be enabled by simply listing 'plot' as a tracker.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **title** (*str or callable*) – Title text of the figure. If this is a string, it is shown with a potential placeholder named *time* being replaced by the current simulation time. Conversely, if *title* is a function, it is called with the current (potentially transformed) state and the time as arguments. This function is expected to return a string.
- **output_file** (*str, optional*) – Specifies a single image file, which is updated periodically, so that the progress can be monitored (e.g. on a compute cluster)
- **output_folder** (*str, optional*) – Specifies a folder to which all images are written. The files will have names with increasing numbers.
- **movie_file** (*str, optional*) – Specifies a filename to which a movie of all the frames is written after the simulation.
- **show** (*bool, optional*) – Determines whether the plot is shown while the simulation is running. If *False*, the files are created in the background. This option can slow down a simulation severely.
- **max_fps** (*float*) – Determines the maximal rate (frames per second) at which the plots are updated. Some plots are skipped if the tracker receives data at a higher rate. A larger value (e.g., *math.inf*) can be used to ensure every frame is drawn, which might penalizes the overall performance.
- **plot_args** (*dict*) – Extra arguments supplied to the plot call. For example, this can be used to specify axes ranges when a single panel is shown. For instance, the value *{'ax_style': {'ylim': (0, 1)}}* enforces the y-axis to lie between 0 and 1.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then plotted. This allows to show derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

name = 'plot'

class MaterialConservationTracker(*interrupts=1*, *atol=0.0001*, *rtol=0.0001*)

Bases: *TrackerBase*

Tracking interrupting the simulation when material conservation is broken.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **atol** (*float*) – Absolute tolerance for amount deviations
- **rtol** (*float*) – Relative tolerance for amount deviations

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

initialize(*field*, *info=None*)

Parameters

- **field** (*FieldBase*) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

name = 'material_conservation'

class MaxRuntimeTracker(*max_runtime*, *interrupts=1*)

Bases: *TrackerBase*

Tracker interrupting the simulation once a duration has passed.

Parameters

- **max_runtime** (*float or str*) – The maximal runtime of the simulation. If the runtime is exceeded, the simulation is interrupted. Values can be either given as a number (interpreted as seconds) or as a string, which is then parsed using the function *parse_duration()*.
- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

initialize(*field*, *info=None*)

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

class PlotTracker(*interrupts=1*, ***, *title='Time: {time:g}'*, *output_file=None*, *movie=None*, *show=None*, *tight_layout=False*, *max_fps=inf*, *plot_args=None*, *transformation=None*)

Bases: *TransformedTrackerBase*

Tracker plotting data on screen, to files, or writes a movie.

This tracker can be used to create movies from simulations or to simply update a single image file on the fly (i.e. to monitor simulations running on a cluster). The default values of this tracker are chosen with regular output to a file in mind.

Example

To create a movie while running the simulation, you can use

```
movie_tracker = PlotTracker(interrupts=10, movie="my_movie.mp4")
eq.solve(..., tracker=movie_tracker)
```

This will create the file *my_movie.mp4* during the simulation. Note that you can display the frames interactively by setting `show=True`.

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **title** (*str or callable*) – Title text of the figure. If this is a string, it is shown with a potential placeholder named *time* being replaced by the current simulation time. Conversely, if *title* is a function, it is called with the current (potentially transformed) state and the time as arguments. This function is expected to return a string.
- **output_file** (*str, optional*) – Specifies a single image file, which is updated periodically, so that the progress can be monitored (e.g. on a compute cluster)

- **movie** (str or *Movie*) – Create a movie. If a filename is given, all frames are written to this file in the format deduced from the extension after the simulation ran. If a *Movie* is supplied, frames are appended to the instance.
- **show** (*bool, optional*) – Determines whether the plot is shown while the simulation is running. If set to *None*, the images are only shown if neither *output_file* nor *movie* is set, otherwise they are kept hidden. Note that showing the plot can slow down a simulation severely.
- **tight_layout** (*bool*) – Determines whether `tight_layout()` is used.
- **max_fps** (*float*) – Determines the maximal rate (frames per second) at which the plots are updated in real time during the simulation. Some plots are skipped if the tracker receives data at a higher rate. A larger value (e.g., *math.inf*) can be used to ensure every frame is drawn, which might penalizes the overall performance.
- **plot_args** (*dict*) – Extra arguments supplied to the plot call. For example, this can be used to specify axes ranges when a single panel is shown. For instance, the value `{'ax_style': {'ylim': (0, 1)}}` enforces the y-axis to lie between 0 and 1.
- **transformation** (*callable, optional*) – A function that transforms the current state into a new field or field collection, which is then plotted. This allows to show derived quantities of the field during calculations. The argument needs to be a callable function taking 1 or 2 arguments. The first argument always is the current field, while the optional second argument is the associated time.

Note

If an instance of *Movie* is given as the *movie* argument, it can happen that the movie is not written to the file when the simulation ends. This is because, the movie could still be extended by appending frames. To write the movie to a file call its *save()* method. Beside adding frames before and after the simulation, an explicit movie object can also be used to adjust the output. For instance, the following example code creates a movie with a framerate of 15, a resolution of 200 dpi, and a bitrate of 6000 kilobits per second:

```
movie = Movie("movie.mp4", framerate=15, dpi=200, bitrate=6000)
eq.solve(..., tracker=PlotTracker(1, movie=movie))
movie.save()
```

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters

info (*dict*) – Extra information from the simulation

Return type

None

handle(*state*, *t*)

Handle data supplied to this tracker.

Parameters

- **state** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type

None

initialize(*state*, *info=None*)

Initialize the tracker with information about the simulation.

Parameters

- **state** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns

The first time the tracker needs to handle data

Return type

float

movie: *Movie* | None

class PrintTracker(*interrupts=1*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *TrackerBase*

Tracker printing data to a stream (default: stdout)

Parameters

- **interrupts** (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.
- **stream** (*IO[str]*) – The stream used for printing

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type

None

name = 'print'

class ProgressTracker(*interrupts=None*, ***, *fancy=True*, *ndigits=5*, *leave=True*)

Bases: *TrackerBase*

Tracker showing the progress of the simulation.

Parameters

- **interrupts** (*InterruptData | None*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*. The default value *None* updates the progress bar approximately every (real) second.
- **fancy** (*bool*) – Flag determining whether a fancy progress bar should be used in jupyter notebooks (if ipywidgets is installed)
- **ndigits** (*int*) – The number of digits after the decimal point that are shown maximally.

- **leave** (*bool*) – Whether to leave the progress bar after the simulation has finished (default: True)

atol: float = 1e-06

finalize(*info=None*)

Finalize the tracker, supplying additional information.

Parameters
info (*dict*) – Extra information from the simulation

Return type
None

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

initialize(*field*, *info=None*)

Initialize the tracker with information about the simulation.

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns
The first time the tracker needs to handle data

Return type
float

name = 'progress'

class SteadyStateTracker(*interrupts=None*, *atol=1e-08*, *rtol=1e-05*, ***, *progress=False*, *evolution_rate=None*)

Bases: *TrackerBase*

Tracker aborting the simulation once steady state is reached.

Steady state is obtained when the state does not change anymore, i.e., when the evolution rate is close to zero. If the argument *evolution_rate* is specified, it is used to calculate the evolution rate directly. If it is omitted, the evolution rate is estimated by comparing the current state *cur* to the state *prev* at the previous time step. In both cases, convergence is assumed when the absolute value of the evolution rate falls below `atol + rtol * cur` for all points. Here, *atol* and *rtol* denote absolute and relative tolerances, respectively.

Parameters

- **interrupts** (*InterruptData | None*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*. The default value *None* checks for the steady state approximately every (real) second.
- **atol** (*float*) – Absolute tolerance that must be reached to abort the simulation

- **rtol** (*float*) – Relative tolerance that must be reached to abort the simulation
- **progress** (*bool*) – Flag indicating whether the progress towards convergence is shown graphically during the simulation
- **evolution_rate** (*callable*) – Function to evaluate the current evolution rate. If omitted, the evolution rate is estimate from the change in the state variable, which can be less accurate. A suitable form of the function is returned by *eq.make_pde_rhs(state, backend="numpy")* when *eq* is the PDE class.

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

name = 'steady_state'

progress_bar_format = 'Convergence: {percentage:3.0f}%|{bar}|
[{elapsed}<{remaining}]'

Determines the format of the progress bar shown when *progress = True*

class WalltimeTracker(*interrupts=1*)

Bases: *TrackerBase*

Special tracker that records runtime of the simulation at regular intervals.

data

A list of tuples (simulation_time, wall_time)

Type
list

Parameters
interrupts (*InterruptData*) – Determines when the tracker interrupts the simulation. A single numbers determines an interval (measured in the simulation time unit) of regular interruption. A string is interpreted as a duration in real time assuming the format 'hh:mm:ss'. A list of values is taken as explicit simulation time points. More fine- grained control is possible by passing an instance of classes defined in *interrupts*.

data: list[tuple[float, float]]

handle(*field*, *t*)

Handle data supplied to this tracker.

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

Return type
None

initialize(*field*, *info*)

Initialize the tracker with information about the simulation.

Parameters

- **field** (FieldBase) – An example of the data that will be analyzed by the tracker
- **info** (*dict*) – Extra information from the simulation

Returns

The first time the tracker needs to handle data

Return type

float

4.9 pde.visualization package

Functions and classes for visualizing simulations.

movie	Produce a movie by simply plotting each frame.
movie_scalar	Produce a movie for a simulation of a scalar field.
movie_multiple	Produce a movie for a simulation with n components.
plot_interactive	plots stored data interactively using the napari viewer
plot_kymograph	Plots a single kymograph from stored data.
plot_kymographs	Plots kymographs for all fields stored in *storage*
plot_magnitudes	Plot spatially averaged quantities as a function of time.

4.9.1 pde.visualization.movies module

Functions for creating movies of simulation results.

Movie	Class for creating movies from matplotlib figures using ffmpeg.
movie_scalar	Produce a movie for a simulation of a scalar field.
movie_multiple	Produce a movie for a simulation with n components.
movie	Produce a movie by simply plotting each frame.

class Movie(*filename*, *framerate=30*, *dpi=None*, ***kwargs*)

Bases: object

Class for creating movies from matplotlib figures using ffmpeg.

Note

Internally, this class uses matplotlib.animation.FFMpegWriter. Note that the *ffmpeg* program needs to be installed in a system path, so that *matplotlib* can find it.

Warning

The movie is only fully written after the *save()* method has been called. To aid with this, it is best practice to use a contextmanager:

```
with Movie("output.mp4") as movie:
    movie.add_figure()
```

Parameters

- **filename** (*str*) – The filename where the movie is stored. The suffix of this path also determines the default movie codec.
- **framerate** (*float*) – The number of frames per second, which determines how fast the movie will appear to run.
- **dpi** (*float*) – The resolution of the resulting movie. The default value is controlled by matplotlib and is usually set to 100.
- ****kwargs** – Additional parameters are used to initialize matplotlib.animation.FFMpegWriter. Here, we can for instance set the bit rate of the resulting video using the *bitrate* parameter.

add_figure(*fig=None*)

Adds the figure *fig* as a frame to the current movie.

Parameters

fig (Figure) – The plot figure that is added to the movie

classmethod is_available()

Check whether the movie infrastructure is available.

Returns

True if movies can be created

Return type

bool

save()

Convert the recorded images to a movie using ffmpeg.

movie(*storage*, *filename*, ***, *progress=True*, *show_time=True*, *plot_args=None*, *movie_args=None*)

Produce a movie by simply plotting each frame.

Parameters

- **storage** (*StorageBase*) – The storage instance that contains all the data for the movie
- **filename** (*str*) – The filename to which the movie is written. The extension determines the format used.
- **progress** (*bool*) – Flag determining whether the progress of making the movie is shown.
- **show_time** (*bool*) – Whether to show the simulation time in the movie
- **plot_args** (*dict*) – Additional arguments for the function plotting the state
- **movie_args** (*dict*) – Additional arguments for *Movie*. For example, this can be used to set the resolution (*dpi*), the framerate (*framerate*), and the bitrate (*bitrate*).

Return type

None

movie_multiple(*storage*, *filename*, *quantities=None*, *scale='automatic'*, *progress=True*)

Produce a movie for a simulation with n components.

Parameters

- **storage** (*StorageBase*) – The storage instance that contains all the data for the movie
- **filename** (*str*) – The filename to which the movie is written. The extension determines the format used.
- **quantities** – A 2d list of quantities that are shown in a rectangular arrangement. If *quantities* is a simple list, the panels will be rendered as a single row. Each panel is defined by a dictionary, where the mandatory item 'source' defines what is being shown. Here, an integer specifies the component that is extracted from the field while a function is evaluate with the full state as an input and the result is shown. Additional items in the dictionary can be 'title' (setting the title of the panel), 'scale' (defining the color range shown; these are typically two numbers defining the lower and upper bound, but if only one is given the range [0, scale] is assumed), and 'cmap' (defining the colormap being used).
- **scale** (*str, float, tuple of float*) – Flag determining how the range of the color scale is determined. In the simplest case a tuple of numbers marks the lower and upper end of the scalar values that will be shown. If only a single number is supplied, the range starts at zero and ends at the given number. Additionally, the special value 'automatic' determines the range from the range of scalar values.
- **progress** (*bool*) – Flag determining whether the progress of making the movie is shown.

Return type

None

movie_scalar(*storage*, *filename*, *scale='automatic'*, *extras=None*, *progress=True*, *tight=False*, *show=True*)

Produce a movie for a simulation of a scalar field.

Parameters

- **storage** (*StorageBase*) – The storage instance that contains all the data for the movie
- **filename** (*str*) – The filename to which the movie is written. The extension determines the format used.
- **scale** (*str, float, tuple of float*) – Flag determining how the range of the color scale is determined. In the simplest case a tuple of numbers marks the lower and upper end of the scalar values that will be shown. If only a single number is supplied, the range starts at zero and ends at the given number. Additionally, the special value 'automatic' determines the range from the range of scalar values.
- **extras** (*dict, optional*) – Additional functions that are evaluated and shown for each time step. The key of the dictionary is used as a panel title.
- **progress** (*bool*) – Flag determining whether the progress of making the movie is shown.
- **tight** (*bool*) – Whether to call `matplotlib.pyplot.tight_layout()`. This affects the layout of all plot elements.
- **show** (*bool*) – Flag determining whether images are shown during making the movie

Return type

None

4.9.2 pde.visualization.plotting module

Functions and classes for plotting simulation data.

ScalarFieldPlot	Class managing compound plots of scalar fields.
plot_magnitudes	Plot spatially averaged quantities as a function of time.
plot_kymograph	Plots a single kymograph from stored data.
plot_kymographs	Plots kymographs for all fields stored in *storage*
plot_interactive	plots stored data interactively using the napari viewer

class ScalarFieldPlot(*fields*, *quantities=None*, *scale='automatic'*, *fig=None*, *title=None*, *tight=False*, *show=True*)

Bases: object

Class managing compound plots of scalar fields.

Parameters

- **fields** (*FieldBase*) – Collection of fields
- **quantities** – A 2d list of quantities that are shown in a rectangular arrangement. If *quantities* is a simple list, the panels will be rendered as a single row. Each panel is defined by a dictionary, where the mandatory item 'source' defines what is being shown. Here, an integer specifies the component that is extracted from the field while a function is evaluate with the full state as an input and the result is shown. Additional items in the dictionary can be 'title' (setting the title of the panel), 'scale' (defining the color range shown; these are typically two numbers defining the lower and upper bound, but if only one is given the range [0, scale] is assumed), and 'cmap' (defining the colormap being used).
- **scale** (*str, float, tuple of float*) – Flag determining how the range of the color scale is determined. In the simplest case a tuple of numbers marks the lower and upper end of the scalar values that will be shown. If only a single number is supplied, the range starts at zero and ends at the given number. Additionally, the special value 'automatic' determines the range from the range of scalar values.
- **(** (*fig*) – class:*matplotlib.figure.Figure): Figure to be used for plotting. If `None*, a new figure is created.
- **title** (*str, optional*) – Title of the plot.
- **tight** (*bool*) – Whether to call `matplotlib.pyplot.tight_layout()`. This affects the layout of all plot elements.
- **show** (*bool*) – Flag determining whether to show a plot. If *False*, the plot is kept in the background, which can be useful if it only needs to be written to a file.

classmethod from_storage(*storage*, *quantities=None*, *scale='automatic'*, *tight=False*, *show=True*)

Create ScalarFieldPlot from storage.

Parameters

- **storage** (*StorageBase*) – Instance of the storage class that contains the data
- **quantities** – A 2d list of quantities that are shown in a rectangular arrangement. If *quantities* is a simple list, the panels will be rendered as a single row. Each panel is defined by a dictionary, where the mandatory item 'source' defines what is being shown. Here, an integer specifies the component that is extracted from the field while a function is evaluate with the full state as an input and the result is shown. Additional items in the dictionary can be 'title' (setting the title of the panel), 'scale' (defining the color range shown; these are typically two

numbers defining the lower and upper bound, but if only one is given the range [0, scale] is assumed), and 'cmap' (defining the colormap being used).

- **scale** (*str, float, tuple of float*) – Flag determining how the range of the color scale is determined. In the simplest case a tuple of numbers marks the lower and upper end of the scalar values that will be shown. If only a single number is supplied, the range starts at zero and ends at the given number. Additionally, the special value 'automatic' determines the range from the range of scalar values.
- **tight** (*bool*) – Whether to call `matplotlib.pyplot.tight_layout()`. This affects the layout of all plot elements.
- **show** (*bool*) – Flag determining whether to show a plot. If *False*, the plot is kept in the background.

Returns
ScalarFieldPlot

Return type
ScalarFieldPlot

make_movie(*storage*, *filename*, *progress=True*)

Make a movie from the data stored in storage.

Parameters

- **storage** (*StorageBase*) – The storage instance that contains all the data for the movie
- **filename** (*str*) – The filename to which the movie is written. The extension determines the format used.
- **progress** (*bool*) – Flag determining whether the progress of making the movie is shown.

Return type
None

savefig(*path*, ***kwargs*)

Save plot to file.

Parameters

- **path** (*str*) – The path to the file where the image is written. The file extension determines the image format
- ****kwargs** – Additional arguments are forwarded to `matplotlib.figure.Figure.savefig()`.

update(*fields*, *title=None*)

Update the plot with the given fields.

Parameters

- **fields** (FieldBase) – The field or field collection of which the defined quantities are shown.
- **title** (*str, optional*) – The title of this view. If *None*, the current title is not changed.

Return type
None

extract_field(*fields*, *source=None*, *check_rank=None*)

Extracts a single field from a possible collection.

Parameters

- **fields** (FieldBase) – The field from which data is extracted
- **source** (*int, callable, or None, optional*) – Determines how a field is extracted from *fields*. If *None*, *fields* is passed as is, assuming it is already a scalar field. This works for the simple, standard case where only a single *ScalarField* is treated. Alternatively, *source* can be an integer, indicating which field is extracted from an instance of *FieldCollection*. Lastly, *source* can be a function that takes *fields* as an argument and returns the desired field.
- **check_rank** (*int, optional*) – Can be given to check whether the extracted field has the correct rank (0 = pde.fields.scalar.ScalarField, 1 = pde.fields.vectorial.VectorField, …).

Returns
The extracted field

Return type
DataFieldBase

plot_interactive(*storage*, *time_scaling='exact'*, *viewer_args=None*, ***kwargs*)

plots stored data interactively using the napari viewer

Parameters

- **storage** (StorageBase) – The storage instance that contains all the data
- **time_scaling** (*str*) – Defines how the time axis is scaled. Possible options are "exact" (the actual time points are used), or "scaled" (the axis is scaled so that it has similar dimension to the spatial axes). Note that the spatial axes will never be scaled.
- **viewer_args** (*dict, optional*) – Arguments passed to napari.viewer.Viewer to affect the viewer.
- ****kwargs** – Extra arguments passed to the plotting function

plot_kymograph(*storage*, *field_index=None*, *scalar='auto'*, *extract='auto'*, *colorbar=True*, *transpose=False*, **args*, *title=None*, *filename=None*, *action='auto'*, *ax_style=None*, *fig_style=None*, *ax=None*, ***kwargs*)

Plots a single kymograph from stored data.

The kymograph shows line data stacked along time. Consequently, the resulting image shows space along the horizontal axis and time along the vertical axis.

Parameters

- **storage** (StorageBase) – The storage instance that contains all the data
- **field_index** (*int, optional*) – An index to choose a single field out of many in a collection stored in *storage*. This option should not be used if only a single field is stored in a collection.
- **scalar** (*str*) – The method for extracting scalars as described in DataFieldBase.to_scalar().
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.
- **colorbar** (*bool*) – Whether to show a colorbar or not
- **transpose** (*bool*) – Determines whether the transpose of the data should is plotted
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically.
- **filename** (*str, optional*) – If given, the plot is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, matplotlib.pyplot.show() will be called to show the plot. If the value is *auto* or *none*,

the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given.

- **ax_style** (*dict*) – Dictionary with properties that will be changed on the axis after the plot has been drawn by calling `matplotlib.pyplot.setp()`. A special item i this dictionary is *use_offset*, which is flag that can be used to control whether offset are shown along the axes of the plot.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **ax** (`matplotlib.axes.Axes`) – Figure axes to be used for plotting. The special value "create" creates a new figure, while "reuse" attempts to reuse an existing figure, which is the default.
- ****kwargs** – Additional keyword arguments are passed to `matplotlib.pyplot.imshow()`.

Returns

The reference to the plot

Return type

PlotReference

plot_kymographs(*storage*, *scalar='auto'*, *extract='auto'*, *colorbar=True*, *transpose=False*, *resize_fig=True*, **args*, *title=None*, *constrained_layout=True*, *filename=None*, *action='auto'*, *fig_style=None*, *fig=None*, ***kwargs*)

Plots kymographs for all fields stored in *storage*

The kymograph shows line data stacked along time. Consequently, the resulting image shows space along the horizontal axis and time along the vertical axis.

Parameters

- **storage** (`StorageBase`) – The storage instance that contains all the data
- **scalar** (*str*) – The method for extracting scalars as described in `DataFieldBase.to_scalar()`.
- **extract** (*str*) – The method used for extracting the line data. See the docstring of the grid method *get_line_data* to find supported values.
- **colorbar** (*bool*) – Whether to show a colorbar or not
- **transpose** (*bool*) – Determines whether the transpose of the data should is plotted
- **resize_fig** (*bool*) – Whether to resize the figure to adjust to the number of panels
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically. This is shown above all panels.
- **constrained_layout** (*bool*) – Whether to use *constrained_layout* in `matplotlib.pyplot.figure()` call to create a figure. This affects the layout of all plot elements. Generally, spacing might be better with this flag enabled, but it can also lead to problems when plotting multiple plots successively, e.g., when creating a movie.
- **filename** (*str, optional*) – If given, the figure is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given. The default value *auto* implies that the plot is shown if it is not a nested plot call.

- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **fig** (`matplotlib.figures.Figure`) – Figure that is used for plotting. If omitted, a new figure is created.
- ****kwargs** – Additional keyword arguments are passed to the calls to `matplotlib.pyplot.imshow()`.

Returns
The references to all plots

Return type
list of *PlotReference*

plot_magnitudes(*storage*, *quantities=None*, **args*, *title=None*, *filename=None*, *action='auto'*, *ax_style=None*, *fig_style=None*, *ax=None*, ***kwargs*)

Plot spatially averaged quantities as a function of time.

For scalar fields, the default is to plot the average value while the averaged norm is plotted for vector fields.

Parameters

- **storage** (`StorageBase`) – Instance of *StorageBase* that contains the simulation data that will be plotted
- **quantities** – A 2d list of quantities that are shown in a rectangular arrangement. If *quantities* is a simple list, the panels will be rendered as a single row. Each panel is defined by a dictionary, where the mandatory item 'source' defines what is being shown. Here, an integer specifies the component that is extracted from the field while a function is evaluate with the full state as an input and the result is shown. Additional items in the dictionary can be 'title' (setting the title of the panel), 'scale' (defining the color range shown; these are typically two numbers defining the lower and upper bound, but if only one is given the range [0, scale] is assumed), and 'cmap' (defining the colormap being used).
- **title** (*str*) – Title of the plot. If omitted, the title might be chosen automatically.
- **filename** (*str, optional*) – If given, the plot is written to the specified file.
- **action** (*str*) – Decides what to do with the final figure. If the argument is set to *show*, `matplotlib.pyplot.show()` will be called to show the plot. If the value is *auto* or *none*, the figure will be created, but not necessarily shown. The value *close* closes the figure, after saving it to a file when *filename* is given.
- **ax_style** (*dict*) – Dictionary with properties that will be changed on the axis after the plot has been drawn by calling `matplotlib.pyplot.setp()`. A special item i this dictionary is *use_offset*, which is flag that can be used to control whether offset are shown along the axes of the plot.
- **fig_style** (*dict*) – Dictionary with properties that will be changed on the figure after the plot has been drawn by calling `matplotlib.pyplot.setp()`. For instance, using fig_style={'dpi': 200} increases the resolution of the figure.
- **ax** (`matplotlib.axes.Axes`) – Figure axes to be used for plotting. The special value "create" creates a new figure, while "reuse" attempts to reuse an existing figure, which is the default.
- ****kwargs** – All remaining parameters are forwarded to the *ax.plot* method

Returns
The reference to the plot

Return type
PlotReference

Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

pde.backends, 81
pde.backends.base, 158
pde.backends.jax, 90
pde.backends.jax.backend, 98
pde.backends.jax.config, 103
pde.backends.jax.operators, 90
pde.backends.jax.operators.cartesian, 90
pde.backends.jax.operators.cylindrical_sym, 92
pde.backends.jax.operators.polar_sym, 94
pde.backends.jax.operators.spherical_sym, 95
pde.backends.jax.typing, 103
pde.backends.numba, 104
pde.backends.numba.backend, 115
pde.backends.numba.config, 120
pde.backends.numba.grids, 120
pde.backends.numba.operators, 104
pde.backends.numba.operators.cartesian, 104
pde.backends.numba.operators.common, 106
pde.backends.numba.operators.cylindrical_sym, 107
pde.backends.numba.operators.polar_sym, 109
pde.backends.numba.operators.spherical_sym, 111
pde.backends.numba.overloads, 121
pde.backends.numba.utils, 122
pde.backends.numba_mpi, 124
pde.backends.numba_mpi.backend, 125
pde.backends.numba_mpi.overloads, 126
pde.backends.numpy, 126
pde.backends.numpy.backend, 126
pde.backends.registry, 166
pde.backends.scipy, 130
pde.backends.scipy.backend, 134
pde.backends.scipy.operators, 130
pde.backends.scipy.operators.cartesian, 130
pde.backends.scipy.operators.common, 132
pde.backends.scipy.operators.cylindrical_sym, 133
pde.backends.scipy.operators.polar_sym, 133
pde.backends.scipy.operators.spherical_sym, 134
pde.backends.torch, 135
pde.backends.torch.backend, 152
pde.backends.torch.config, 157
pde.backends.torch.operators, 135
pde.backends.torch.operators.cartesian, 135
pde.backends.torch.operators.common, 139
pde.backends.torch.operators.cylindrical_sym, 140
pde.backends.torch.operators.polar_sym, 144
pde.backends.torch.operators.spherical_sym, 148
pde.backends.torch.typing, 157
pde.backends.torch.utils, 157

f

pde.fields, 168
pde.fields.base, 190
pde.fields.collection, 195
pde.fields.datafield_base, 201
pde.fields.scalar, 212
pde.fields.tensorial, 217
pde.fields.vectorial, 221

g

pde.grids, 227
pde.grids.base, 268
pde.grids.boundaries, 228
pde.grids.boundaries.axes, 231
pde.grids.boundaries.axis, 235
pde.grids.boundaries.local, 238
pde.grids.cartesian, 279
pde.grids.coordinates, 260
pde.grids.coordinates.base, 262
pde.grids.coordinates.bipolar, 265
pde.grids.coordinates.bispherical, 266
pde.grids.coordinates.cartesian, 266
pde.grids.coordinates.cylindrical, 266
pde.grids.coordinates.polar, 267
pde.grids.coordinates.spherical, 267
pde.grids.cylindrical, 284

pde.grids.spherical, 288

p

pde, 81
pde.pdes, 293
pde.pdes.allen_cahn, 294
pde.pdes.base, 295
pde.pdes.cahn_hilliard, 302
pde.pdes.diffusion, 303
pde.pdes.klein_gordon, 304
pde.pdes.kpz_interface, 306
pde.pdes.kuramoto_sivashinsky, 307
pde.pdes.laplace, 309
pde.pdes.pde, 311
pde.pdes.reaction_diffusion, 313
pde.pdes.swift_hohenberg, 314
pde.pdes.wave, 316

s

pde.solvers, 317
pde.solvers.adams_bashforth, 318
pde.solvers.base, 318
pde.solvers.controller, 321
pde.solvers.crank_nicolson, 322
pde.solvers.euler, 322
pde.solvers.explicit_mpi, 323
pde.solvers.implicit, 324
pde.solvers.milstein, 325
pde.solvers.runge_kutta, 325
pde.solvers.scipy, 326
pde.storage, 326
pde.storage.base, 333
pde.storage.file, 338
pde.storage.memory, 340
pde.storage.modelrunner, 341
pde.storage.movie, 343

t

pde.tools, 345
pde.tools.cache, 346
pde.tools.config, 351
pde.tools.cuboid, 356
pde.tools.docstrings, 358
pde.tools.expressions, 359
pde.tools.ffmpeg, 365
pde.tools.math, 367
pde.tools.misc, 368
pde.tools.modelrunner, 372
pde.tools.mpi, 372
pde.tools.nested_dict, 374
pde.tools.numba, 377
pde.tools.output, 377
pde.tools.parse_duration, 378
pde.tools.plotting, 379
pde.tools.spectral, 384
pde.tools.typing, 385
pde.trackers, 387
pde.trackers.base, 388
pde.trackers.interactive, 391
pde.trackers.interrupts, 393
pde.trackers.trackers, 397

v

pde.visualization, 408
pde.visualization.movies, 408
pde.visualization.plotting, 411