py-pde Documentation

Release 0.20.0

David Zwicker

Jul 06, 2022

CONTENTS

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) differential operator that defines the time evolution of a (set of) physical fields u with possibly tensorial character, which depend on spatial coordinates $\boldsymbol{x}$ and time t. The framework also supports stochastic differential equations in the Itô representation, 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.

Contents

CHAPTER

ONE

GETTING STARTED

This *py-pde* package is developed for python 3.7+ and should run on all common platforms. The code is tested under Linux, Windows, and macOS.

1.1 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 and for creating movies.

1.2 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 all required dependencies to have all features of *py-pde*.

1.3 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.

1.3.1 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	0.50	Just-in-time compilation to accelerate numerics
numpy	1.18	Handling numerical data
scipy	1.4	Miscellaneous scientific functions
sympy	1.5	Dealing with user-defined mathematical expressions

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.

1.3.2 Optional packages

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

Package	Usage
h5py	Storing data in the hierarchical file format
ipywidgets	Jupyter notebook support
napari	Displaying images interactively
pandas	Handling tabular data
pyfftw	Faster Fourier transforms
tqdm	Display progress bars during calculations

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.

1.3.3 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`**.

1.4 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 *ExplicitSolver*, but more advanced implementations can be done. To obtain more details during the simulation, trackers can be attached to the solver instance, 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 the intermediate state to memory and to a file, respectively.

CHAPTER

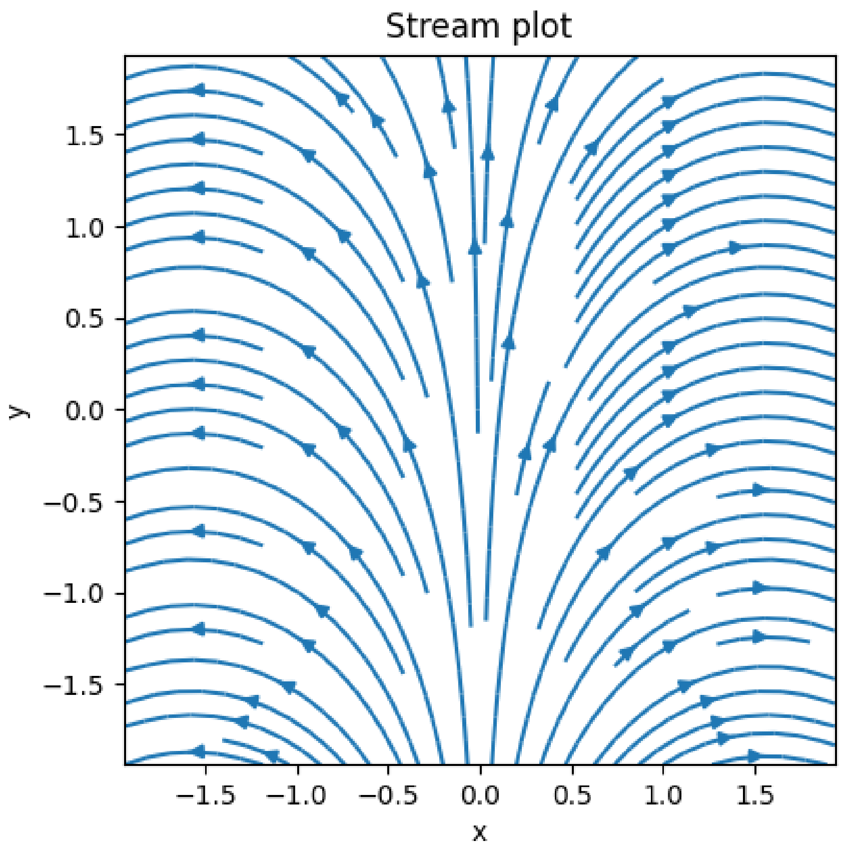
TWO

EXAMPLES

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

2.1 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)
```

(continues on next page)

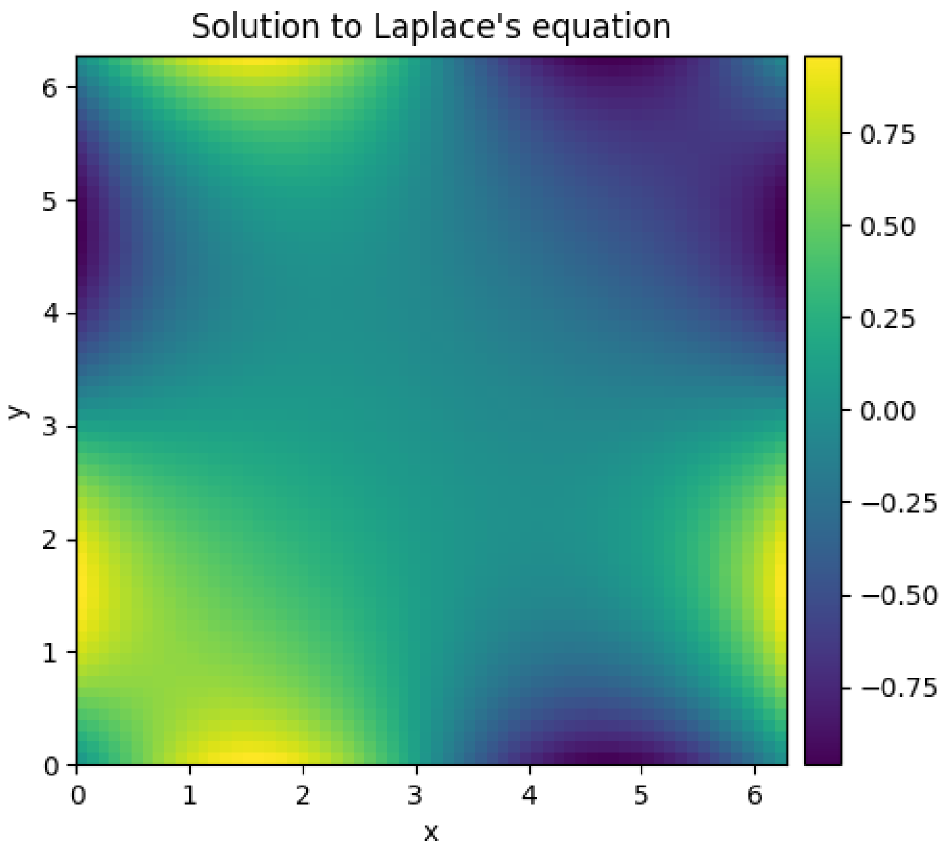
(continued from previous page)

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

Total running time of the script: (0 minutes 0.814 seconds)

2.2 Solving Laplace's equation in 2d

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

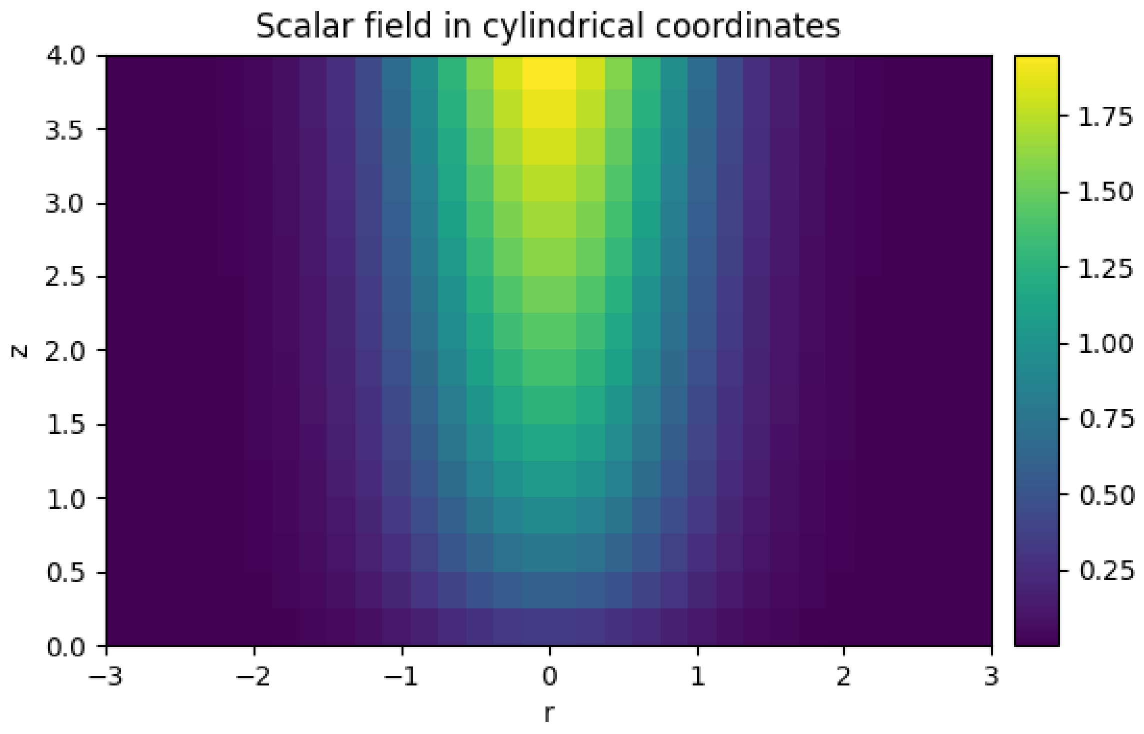


```
import numpy as np

from pde import CartesianGrid, solve_laplace_equation

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

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

Total running time of the script: (0 minutes 1.185 seconds)

2.3 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.

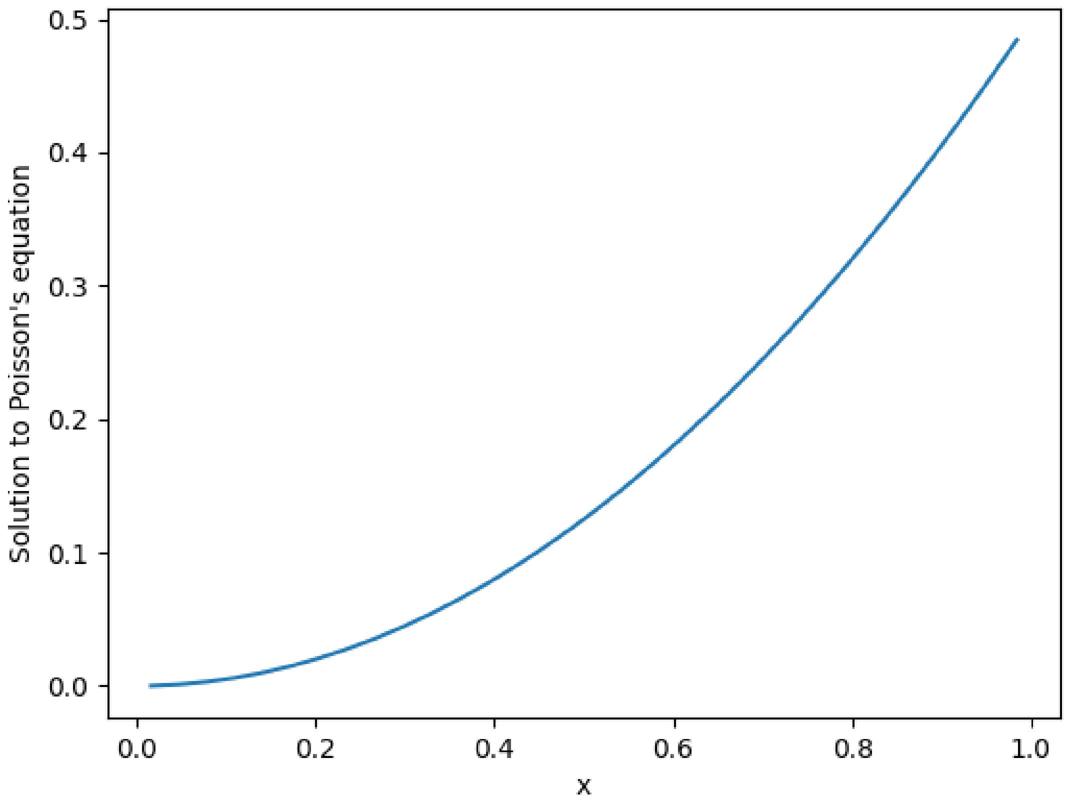


```
from pde import CylindricalSymGrid, ScalarField

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 0.434 seconds)

2.4 Solving Poisson's equation in 1d

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

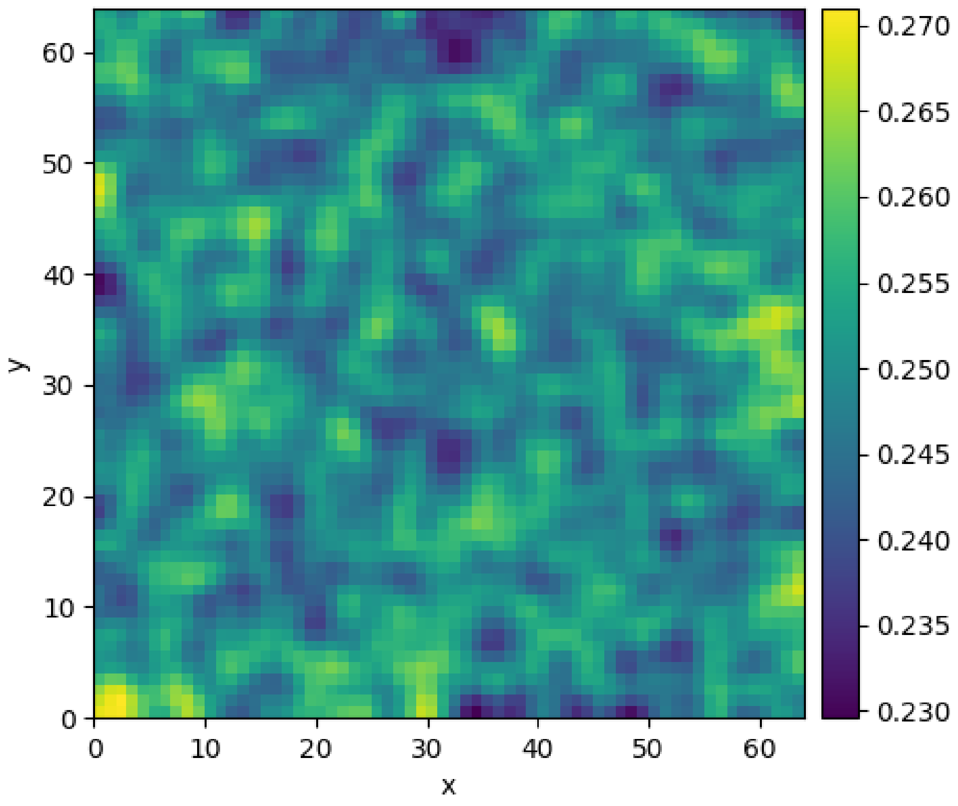


```
from pde import CartesianGrid, ScalarField, solve_poisson_equation

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

result.plot()
```

Total running time of the script: (0 minutes 0.095 seconds)

2.5 Simple diffusion equation

This example solves a simple diffusion equation in two dimensions.



Out:

```
  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<10:22:50, 3738.28s/it]
  1%|          | 0.0546/10.0 [00:11<35:52, 216.45s/it]
  8%|7         | 0.76946/10.0 [00:11<02:21, 15.36s/it]
  8%|7         | 0.76946/10.0 [00:11<02:21, 15.38s/it]
100%|##########| 10.0/10.0 [00:11<00:00,  1.18s/it]
100%|##########| 10.0/10.0 [00:11<00:00,  1.18s/it]
```

```
from pde import DiffusionPDE, ScalarField, UnitGrid

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

(continues on next page)

(continued from previous page)

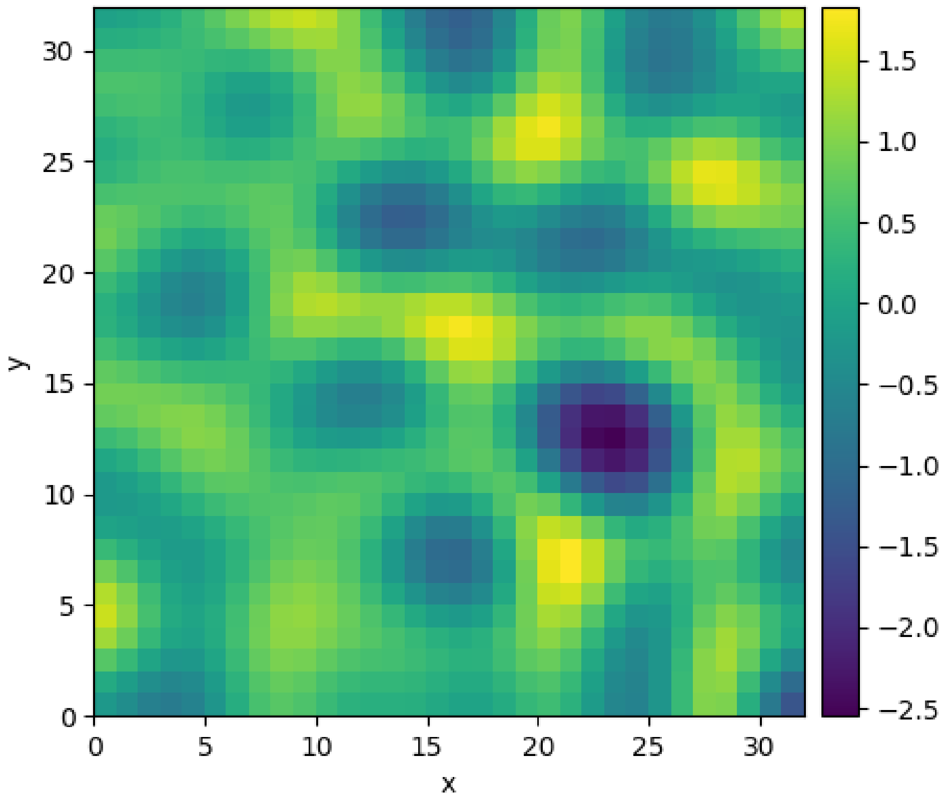
```
eq = DiffusionPDE(diffusivity=0.1)  # define the pde
result = eq.solve(state, t_range=10)
result.plot()
```

Total running time of the script: (0 minutes 12.051 seconds)

2.6 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:

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



Out:

```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:12<?, ?it/s]
  0%|          | 0.01/10.0 [00:25<7:06:03, 2558.92s/it]
  0%|          | 0.02/10.0 [00:28<3:55:57, 1418.62s/it]
```

(continues on next page)

(continued from previous page)

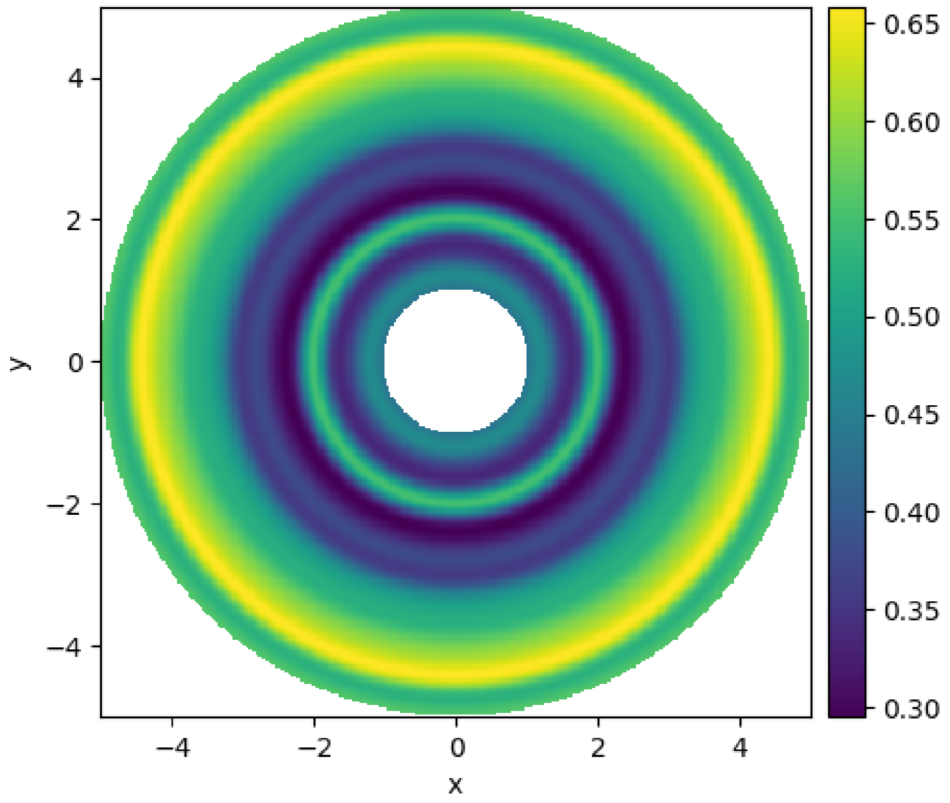
```
  0%|          | 0.03/10.0 [00:28<2:37:09, 945.76s/it]
  1%|1         | 0.1/10.0 [00:28<46:48, 283.73s/it]
 45%|####5     | 4.55/10.0 [00:28<00:33,  6.24s/it]
 45%|####5     | 4.55/10.0 [00:28<00:34,  6.24s/it]
100%|##########| 10.0/10.0 [00:28<00:00,  2.84s/it]
100%|##########| 10.0/10.0 [00:28<00:00,  2.84s/it]
```

```python
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 28.571 seconds)

2.7 Spherically symmetric PDE

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



Out:

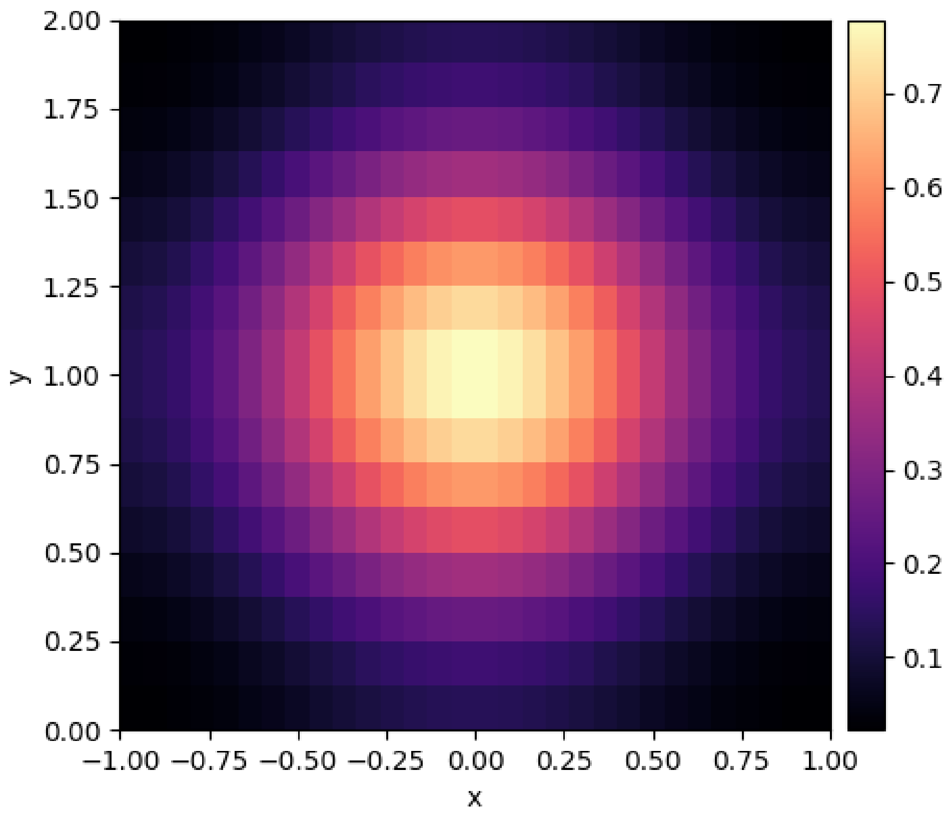
```
  0%|          | 0/0.1 [00:00<?, ?it/s]
Initializing:   0%|          | 0/0.1 [00:00<?, ?it/s]
  0%|          | 0/0.1 [00:02<?, ?it/s]
  6%|6         | 0.006/0.1 [00:03<00:52, 557.87s/it]
 15%|#5        | 0.015/0.1 [00:03<00:21, 251.30s/it]
 28%|##8       | 0.028/0.1 [00:03<00:09, 134.64s/it]
 28%|##8       | 0.028/0.1 [00:03<00:09, 134.66s/it]
100%|##########| 0.1/0.1 [00:03<00:00, 37.71s/it]
100%|##########| 0.1/0.1 [00:03<00:00, 37.71s/it]
```

```python
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 4.420 seconds)

2.8 Diffusion on a Cartesian grid

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



Out:

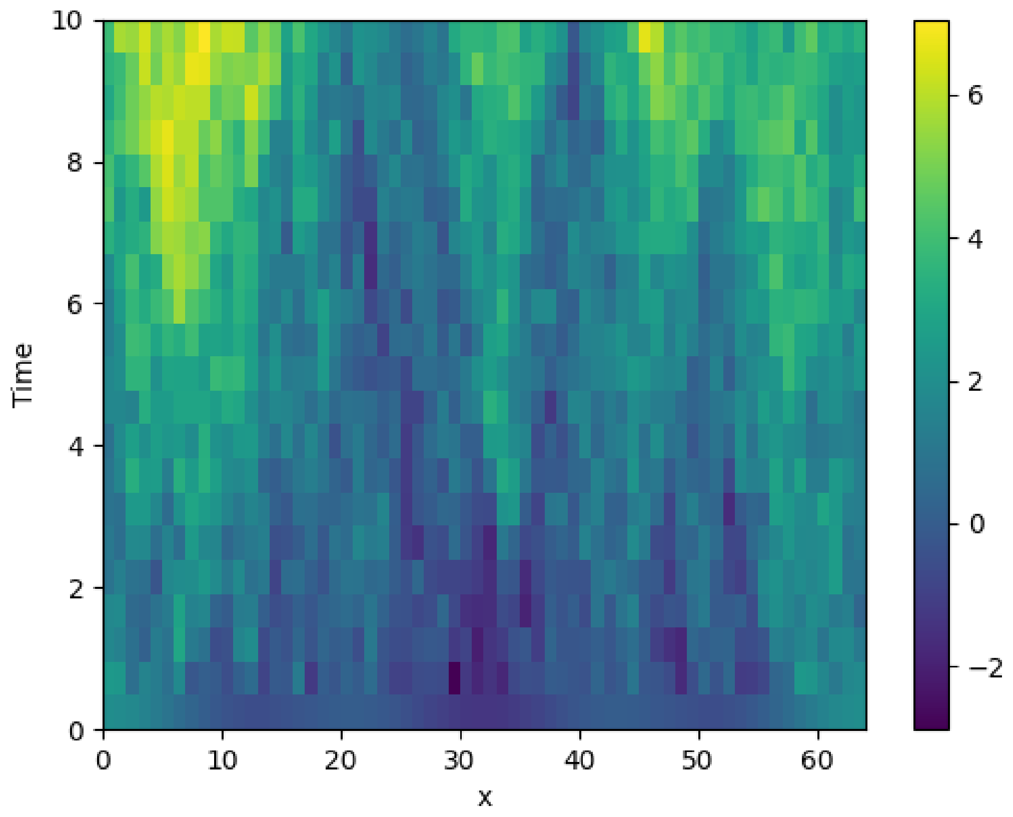
```
  0%|          | 0/1.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/1.0 [00:00<?, ?it/s]
  0%|          | 0/1.0 [00:06<?, ?it/s]
  1%|1         | 0.01/1.0 [00:07<12:53, 780.94s/it]
  2%|2         | 0.02/1.0 [00:08<07:17, 445.93s/it]
  3%|3         | 0.03/1.0 [00:08<04:48, 297.30s/it]
 27%|##7       | 0.27/1.0 [00:08<00:24, 33.03s/it]
 27%|##7       | 0.27/1.0 [00:08<00:24, 33.04s/it]
100%|##########| 1.0/1.0 [00:08<00:00,  8.92s/it]
100%|##########| 1.0/1.0 [00:08<00:00,  8.92s/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 9.071 seconds)

2.9 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()
```

(continues on next page)

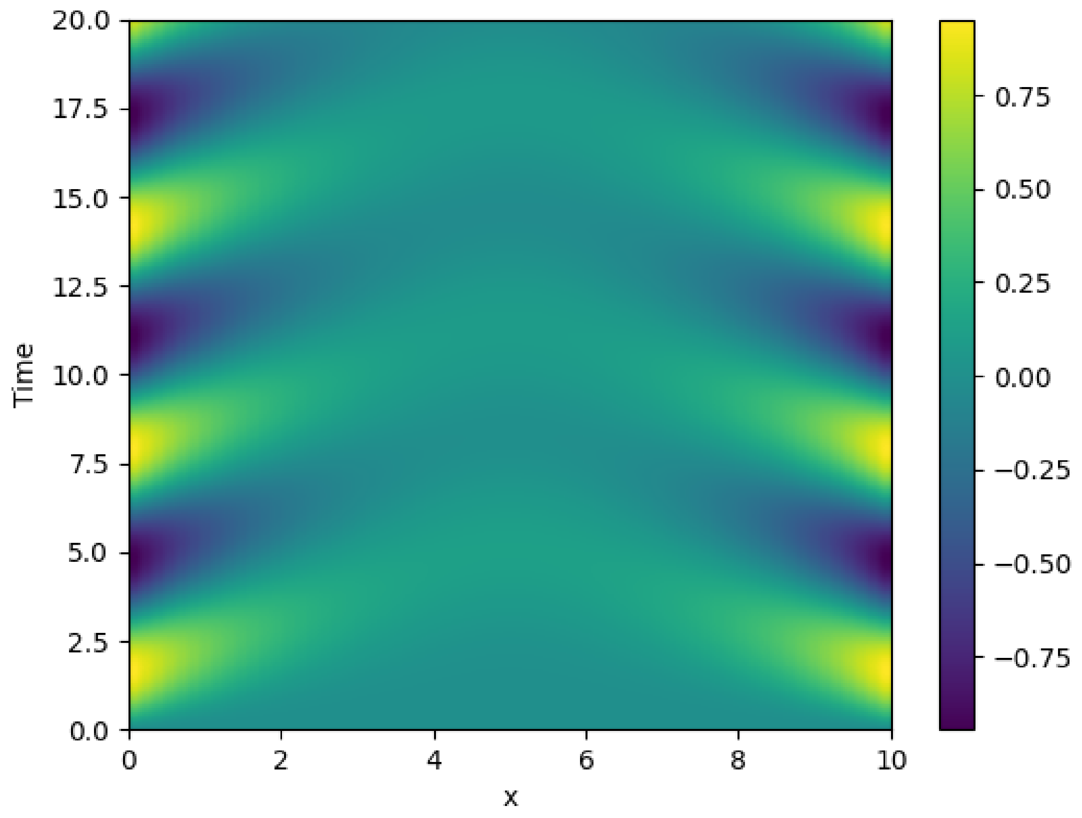
(continued from previous page)

```
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 8.290 seconds)

2.10 Time-dependent boundary conditions

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

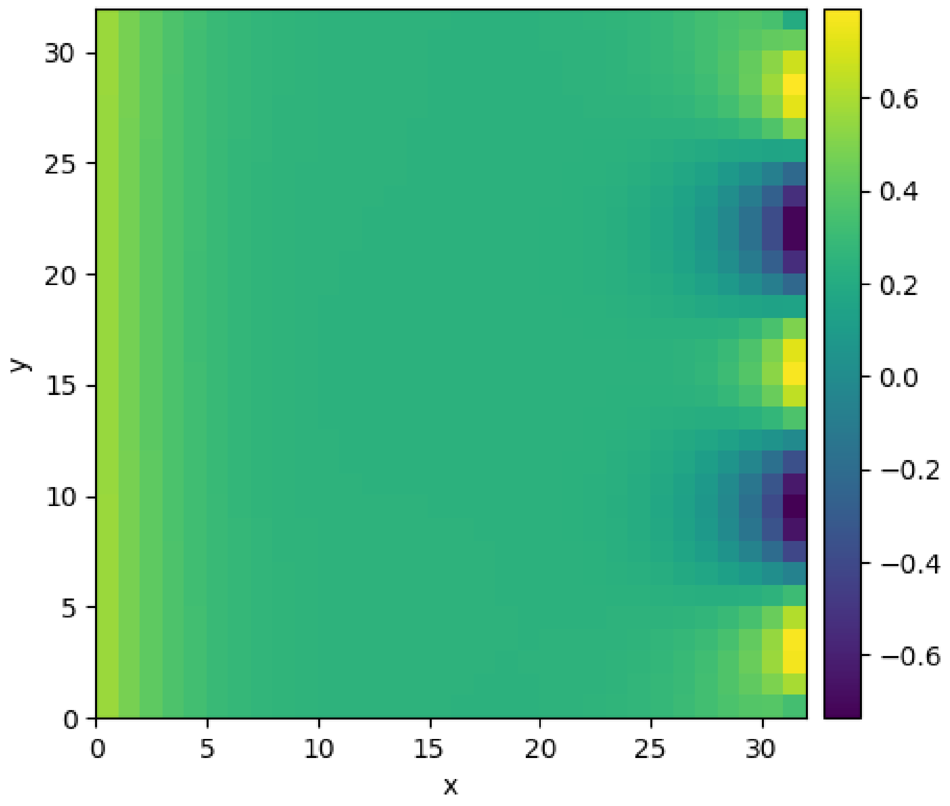


```
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

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 5.978 seconds)

2.11 Setting boundary conditions

This example shows how different boundary conditions can be specified.



Out:

```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:07<?, ?it/s]
  0%|          | 0.005/10.0 [00:08<4:33:37, 1642.61s/it]
  0%|          | 0.01/10.0 [00:09<2:32:54, 918.36s/it]
  0%|          | 0.015/10.0 [00:09<1:41:53, 612.27s/it]
  2%|2         | 0.245/10.0 [00:09<06:05, 37.49s/it]
  2%|2         | 0.245/10.0 [00:09<06:06, 37.54s/it]
100%|##########| 10.0/10.0 [00:09<00:00,  1.09it/s]
100%|##########| 10.0/10.0 [00:09<00:00,  1.09it/s]
```

```
from pde import DiffusionPDE, ScalarField, UnitGrid

grid = UnitGrid([32, 32], periodic=[False, True])  # generate grid
```

(continues on next page)

(continued from previous page)

```
state = ScalarField.random_uniform(grid, 0.2, 0.3)  # generate initial condition

# set boundary conditions `bc` for all axes
bc_x_left = {"derivative": 0.1}
bc_x_right = {"value": "sin(y / 2)"}
bc_x = [bc_x_left, bc_x_right]
bc_y = "periodic"
eq = DiffusionPDE(bc=[bc_x, bc_y])

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

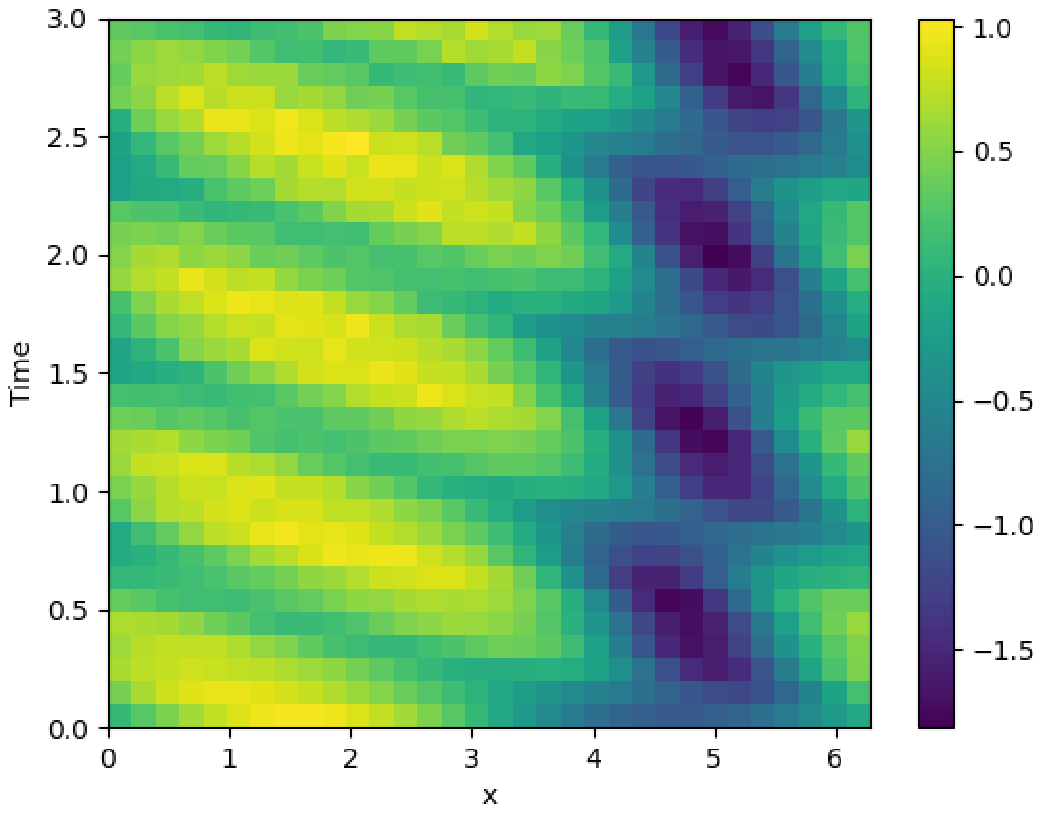
Total running time of the script: (0 minutes 9.335 seconds)

2.12 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, tracker=storage.tracker(0.1))

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

Total running time of the script: (0 minutes 7.105 seconds)

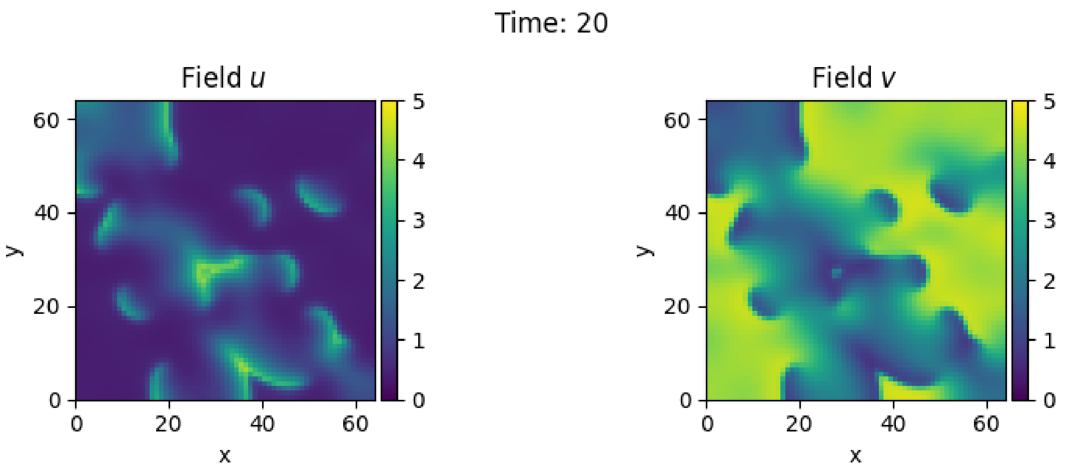
2.13 Brusselator - Using the *PDE* class

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",
```

(continues on next page)

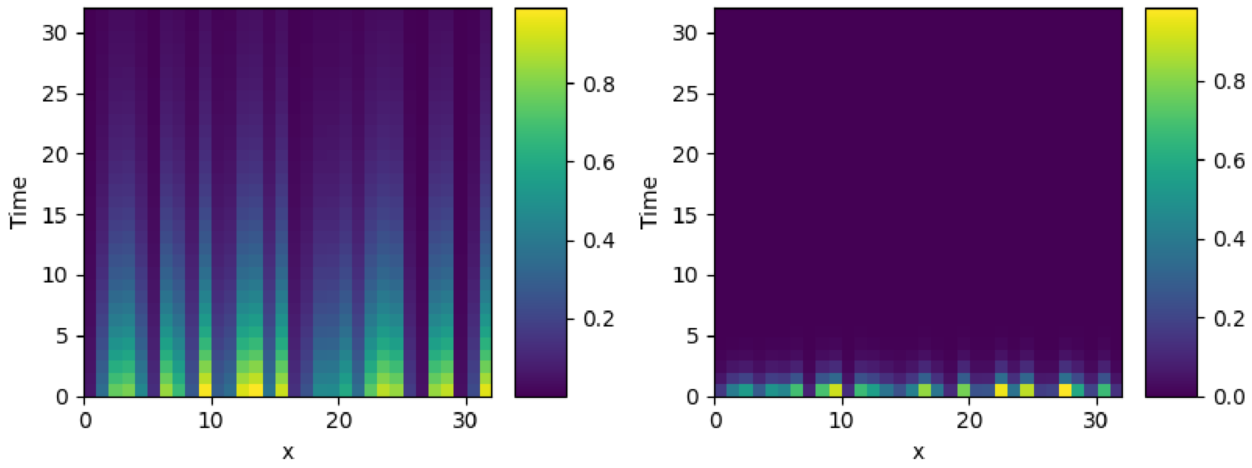
(continued from previous page)

```
        "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(interval=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 27.014 seconds)

2.14 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
path = NamedTemporaryFile(suffix=".hdf5")

# run a simulation and write the results
writer = pde.FileStorage(path.name, write_mode="truncate")
```

(continues on next page)

(continued from previous page)

```
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 5.718 seconds)

2.15 Diffusion equation with spatial dependence

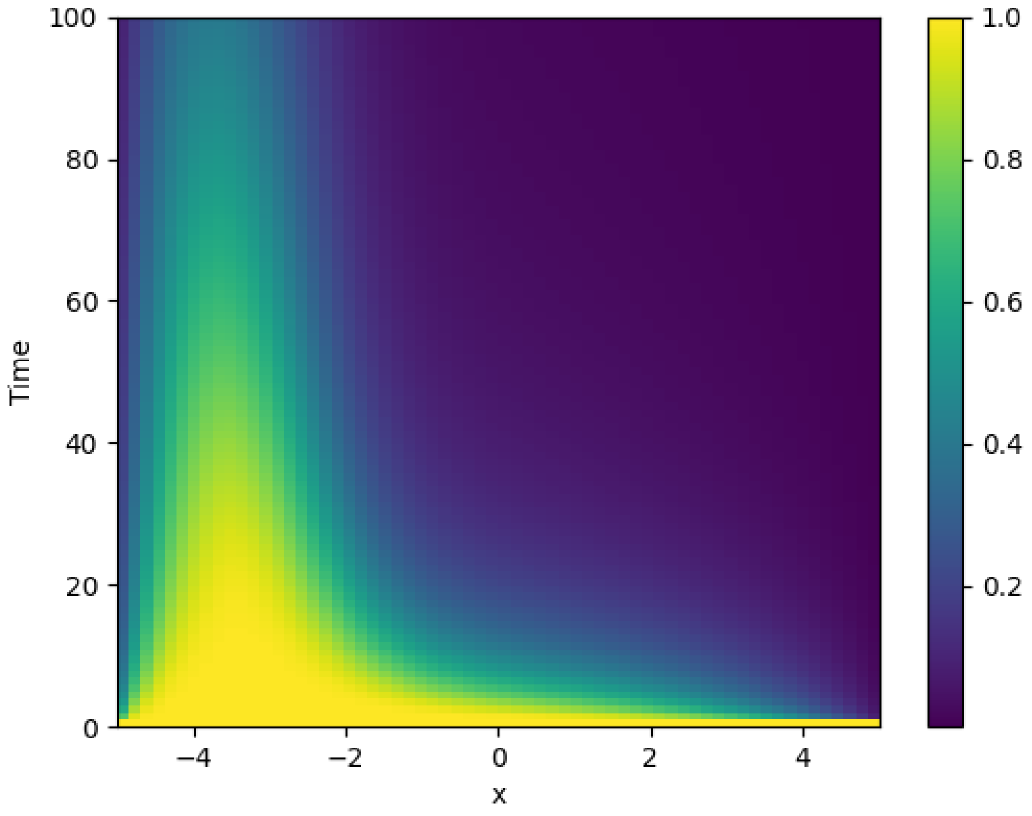
This example solve the Diffusion equation with a heterogeneous diffusivity:

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

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,

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

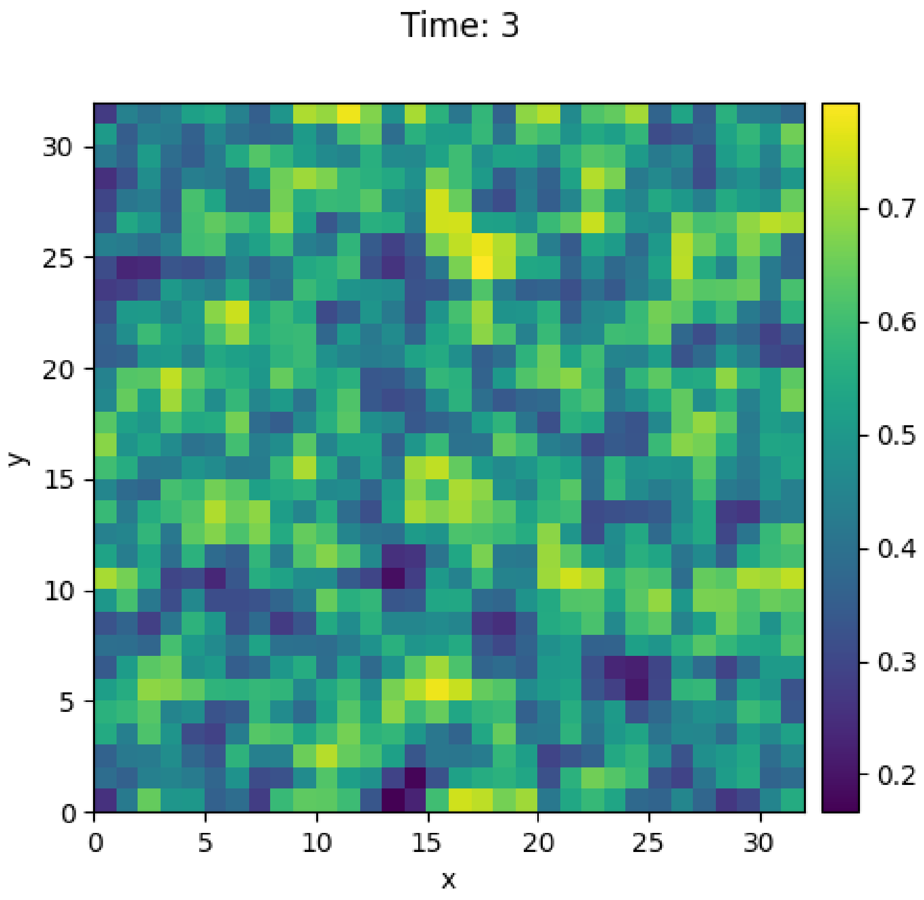


```
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

# 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
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 11.488 seconds)

2.16 Using simulation trackers

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



Out:

```
  0%|          | 0/3.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/3.0 [00:00<?, ?it/s]
  0%|          | 0/3.0 [00:07<?, ?it/s]
  3%|3         | 0.1/3.0 [00:08<04:03, 83.92s/it]
  7%|6         | 0.2/3.0 [00:09<02:10, 46.46s/it]
 10%|9         | 0.3/3.0 [00:09<01:23, 30.98s/it]
 13%|#3        | 0.4/3.0 [00:09<01:00, 23.23s/it]
 13%|#3        | 0.4/3.0 [00:09<01:01, 23.69s/it]
100%|##########| 3.0/3.0 [00:09<00:00,  3.16s/it]
100%|##########| 3.0/3.0 [00:09<00:00,  3.16s/it]
509.7241540373601
509.7241540373601
509.72415403736017
509.7241540373601
```

```python
import pde

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

storage = pde.MemoryStorage()

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

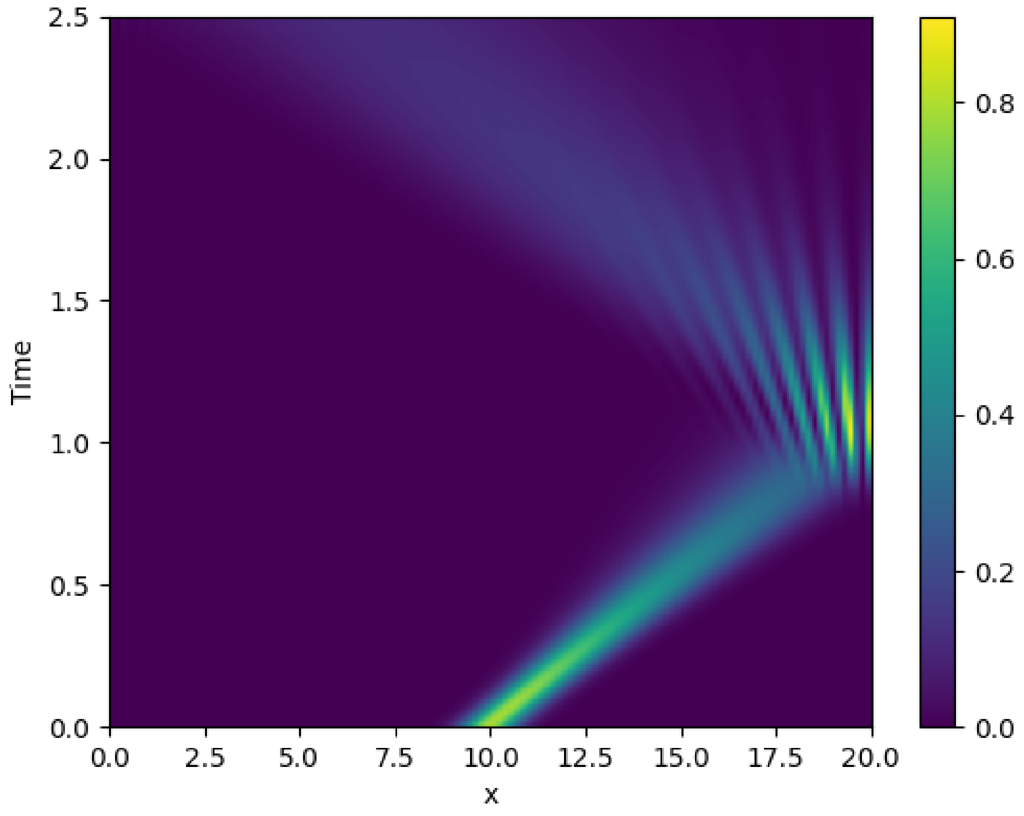
Total running time of the script: (0 minutes 9.539 seconds)

2.17 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.



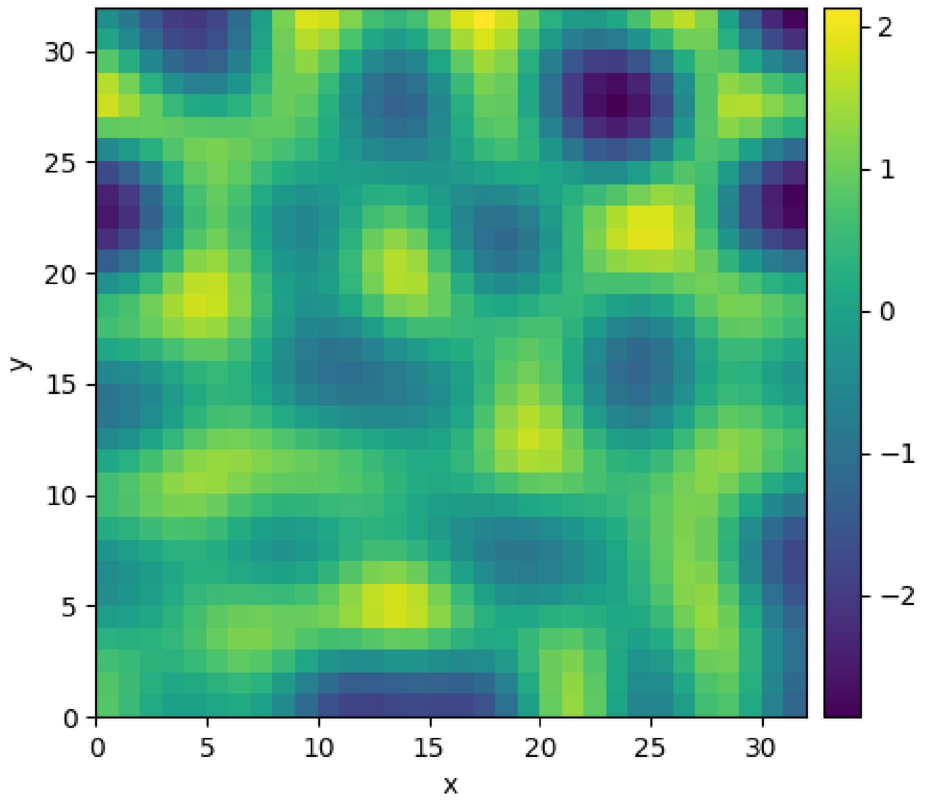
```python
from math import sqrt

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

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

# 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": f"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 5.276 seconds)

2.18 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:

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



Out:

```
  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]
  2%|2         | 0.24/10.0 [00:06<04:23, 27.00s/it]
  3%|3         | 0.33/10.0 [00:06<03:10, 19.65s/it]
 17%|#6        | 1.7/10.0 [00:06<00:31,  3.85s/it]
 72%|#######1  | 7.15/10.0 [00:06<00:02,  1.05it/s]
 72%|#######1  | 7.15/10.0 [00:06<00:02,  1.03it/s]
100%|##########| 10.0/10.0 [00:06<00:00,  1.45it/s]
100%|##########| 10.0/10.0 [00:06<00:00,  1.45it/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"""
        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()
```

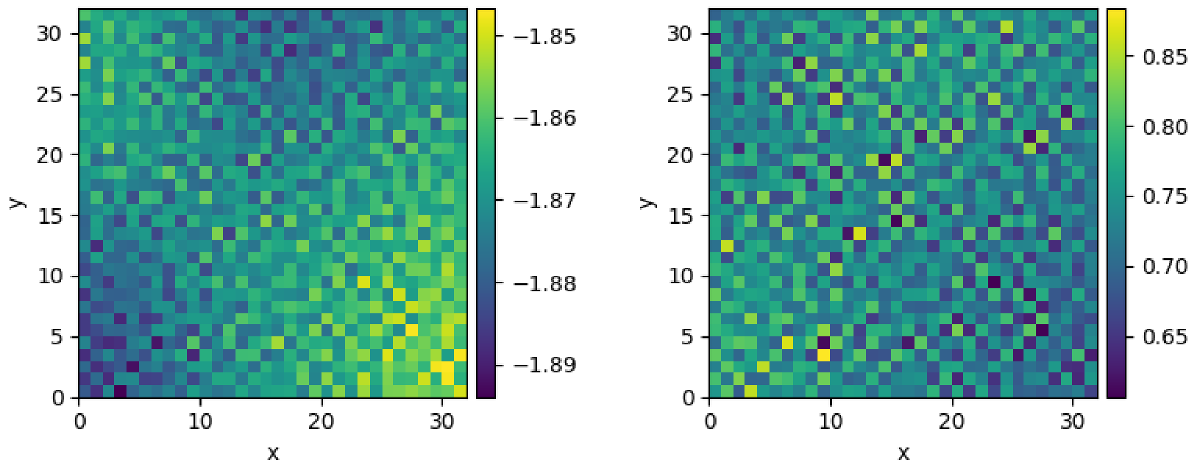
Total running time of the script: (0 minutes 7.046 seconds)

2.19 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.



Out:

```
  0%|          | 0/100.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/100.0 [00:00<?, ?it/s]
```

(continues on next page)

(continued from previous page)

```
  0%|          | 0/100.0 [00:00<?, ?it/s]
  0%|          | 0.24/100.0 [00:03<21:33, 12.96s/it]
  0%|          | 0.38/100.0 [00:03<13:37,  8.21s/it]
  2%|1         | 1.84/100.0 [00:03<02:50,  1.74s/it]
  7%|7         | 7.01/100.0 [00:03<00:46,  2.02it/s]
 17%|#6        | 16.83/100.0 [00:04<00:19,  4.20it/s]
 30%|###       | 30.32/100.0 [00:04<00:10,  6.41it/s]
 46%|####6     | 46.18/100.0 [00:05<00:06,  8.28it/s]
 63%|######3   | 63.4/100.0 [00:06<00:03,  9.76it/s]
 81%|########1 | 81.39/100.0 [00:07<00:01, 10.90it/s]
100%|#########9| 99.61/100.0 [00:08<00:00, 11.81it/s]
100%|#########9| 99.61/100.0 [00:08<00:00, 11.78it/s]
100%|##########| 100.0/100.0 [00:08<00:00, 11.83it/s]
100%|##########| 100.0/100.0 [00:08<00:00, 11.83it/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"):
        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()
```

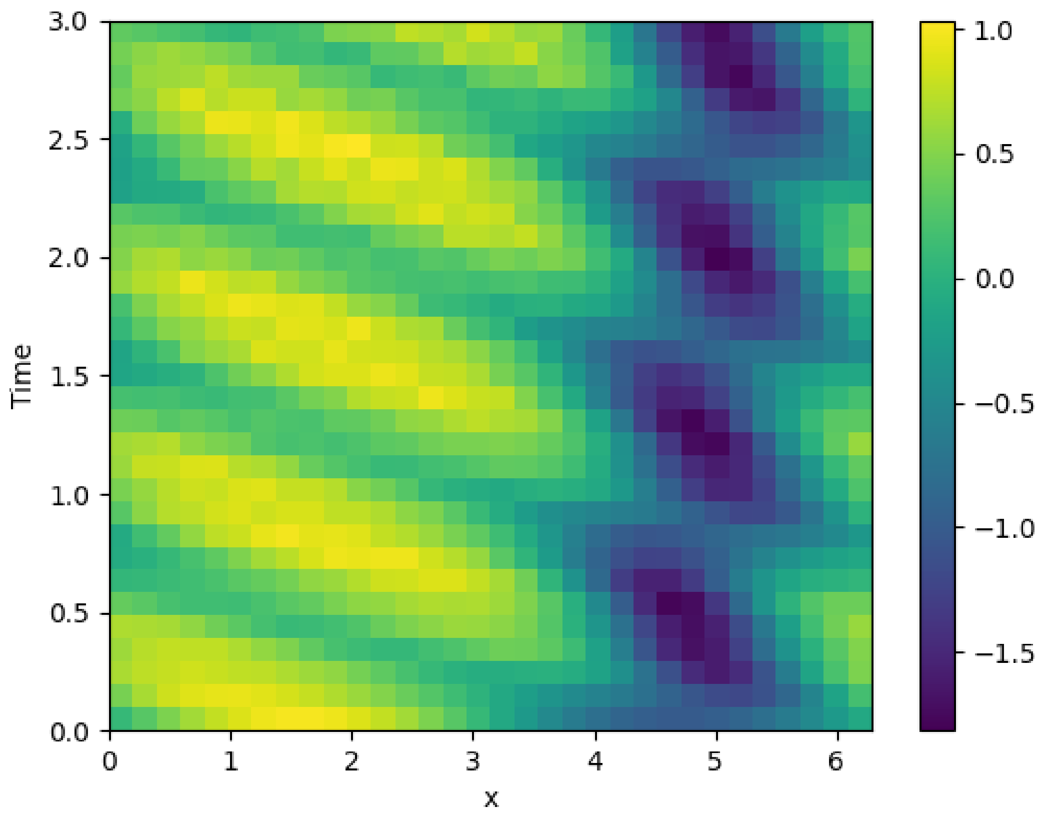
Total running time of the script: (0 minutes 8.745 seconds)

2.20 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
```

(continues on next page)

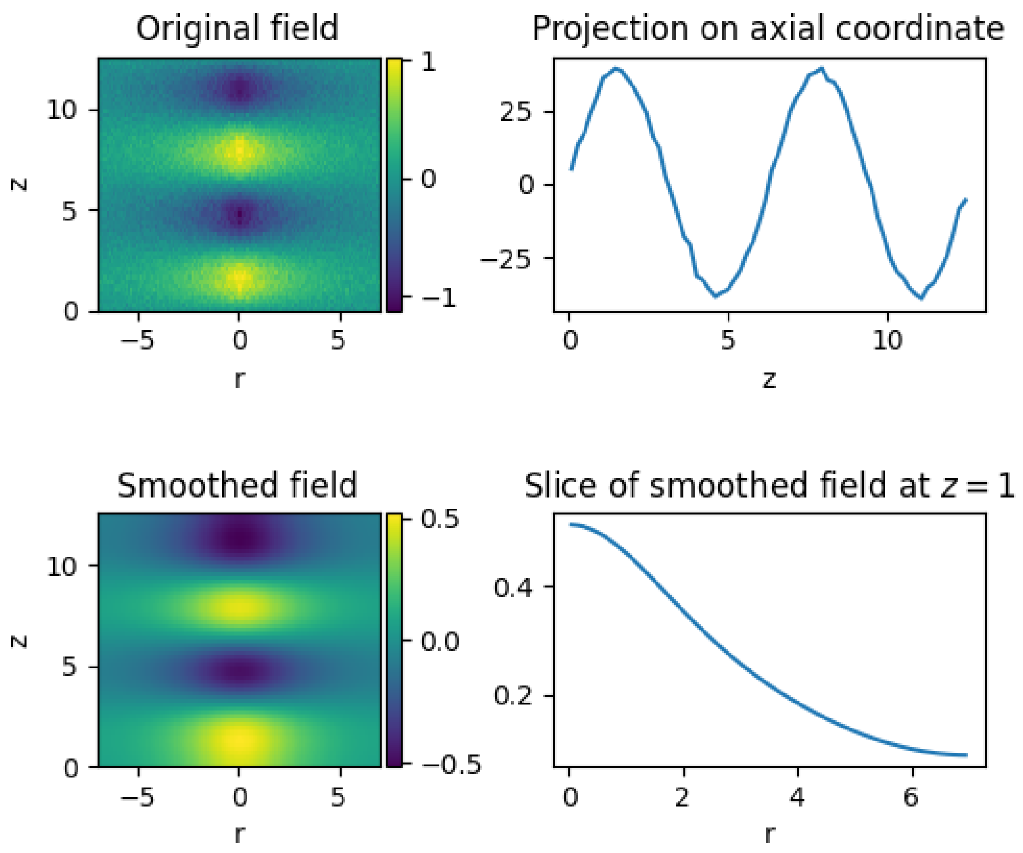
(continued from previous page)

```
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, tracker=storage.tracker(0.1))

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

Total running time of the script: (0 minutes 4.231 seconds)

2.21 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
```

(continues on next page)

(continued from previous page)

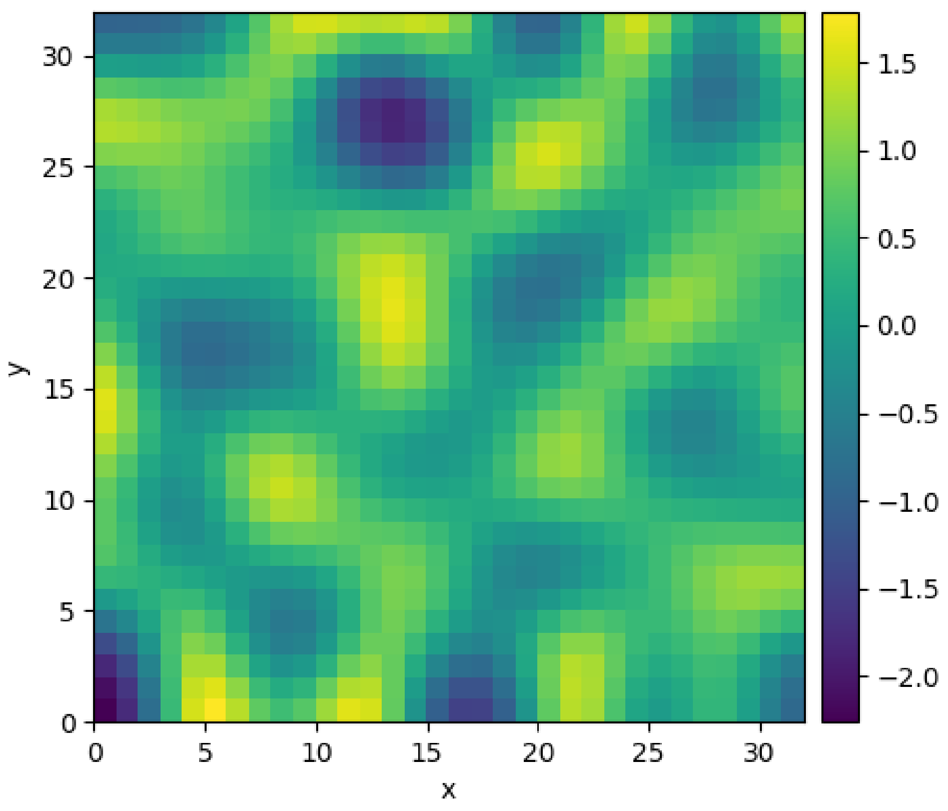
```
# create a scalar field with some noise
grid = CylindricalSymGrid(7, [0, 4 * np.pi], 64)
data = ScalarField.from_expression(grid, "sin(z) * exp(-r / 3)")
data += 0.05 * ScalarField.random_normal(grid)

# 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")
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 0.515 seconds)

2.22 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:

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



Out:

```
  0%|          | 0/10.0 [00:00<?, ?it/s]
Initializing:   0%|          | 0/10.0 [00:00<?, ?it/s]
  0%|          | 0/10.0 [00:15<?, ?it/s]
  0%|          | 0.01/10.0 [00:19<5:20:33, 1925.28s/it]
  0%|          | 0.02/10.0 [00:21<2:57:17, 1065.90s/it]
  0%|          | 0.03/10.0 [00:21<1:58:04, 710.61s/it]
  1%|1         | 0.1/10.0 [00:21<35:10, 213.19s/it]
 46%|####6     | 4.6/10.0 [00:21<00:25,  4.64s/it]
 46%|####6     | 4.6/10.0 [00:21<00:25,  4.64s/it]
100%|##########| 10.0/10.0 [00:21<00:00,  2.13s/it]
100%|##########| 10.0/10.0 [00:21<00:00,  2.13s/it]
```

```python
import numba as nb

from pde import PDEBase, ScalarField, UnitGrid


class KuramotoSivashinskyPDE(PDEBase):
```

(continues on next page)

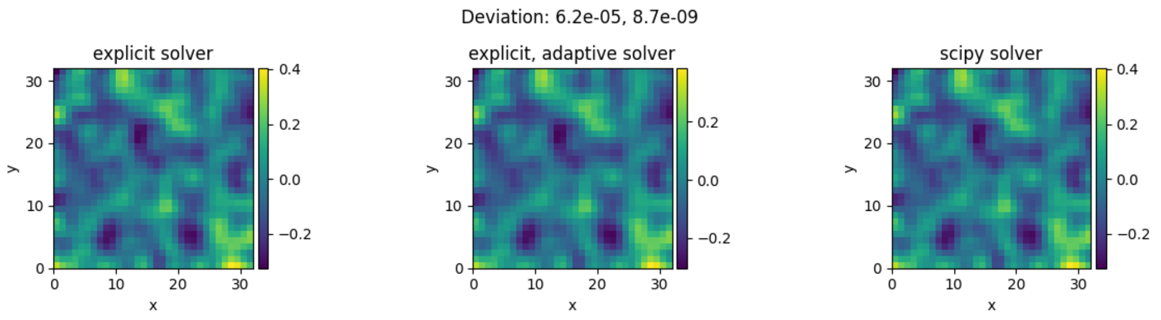
(continued from previous page)

```
    """Implementation of the normalized Kuramoto-Sivashinsky equation"""

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

    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_pde_rhs_numba(self, state):
        """nunmba-compiled implementation of the PDE"""
        gradient_squared = state.grid.make_operator("gradient_squared", bc=self.bc)
        laplace = state.grid.make_operator("laplace", bc=self.bc)

        @nb.jit
        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 21.491 seconds)

2.23 Solver comparison

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



Out:

```
Diagnostic information from first run:
{'controller': {'t_start': 0, 't_end': 1.0, 'solver_class': 'ExplicitSolver', 'solver_
↪start': '2022-07-06 10:12:36.197643', 'profiler': {'solver': 1.4591466370000035,
↪'tracker': 3.57669999857535e-05}, 'successful': True, 'stop_reason': 'Reached final␣
↪time', 'solver_duration': '0:00:01.459303', 't_final': 1.0}, 'package_version': '0.
↪20.0', 'solver': {'class': 'ExplicitSolver', 'pde_class': 'DiffusionPDE', 'dt': 0.
↪001, 'steps': 1000, 'scheme': 'euler', 'state_modifications': 0.0, 'dt_adaptive':␣
↪False, 'backend': 'numba', 'stochastic': False, 'adaptive': False}, 'jit_count': {
↪'make_stepper': 9, 'simulation': 0}}
```

(continues on next page)

(continued from previous page)

```
Diagnostic information from second run:
{'controller': {'t_start': 0, 't_end': 1.0, 'solver_class': 'ExplicitSolver', 'solver_
↪start': '2022-07-06 10:12:38.402686', 'profiler': {'solver': 2.665837649999986,
↪'tracker': 3.635100003407388e-05}, 'successful': True, 'stop_reason': 'Reached␣
↪final time', 'solver_duration': '0:00:02.666077', 't_final': 1.08972255772386},
↪'package_version': '0.20.0', 'solver': {'class': 'ExplicitSolver', 'pde_class':
↪'DiffusionPDE', 'dt': 0.001, 'steps': 16, 'scheme': 'runge-kutta', 'state_
↪modifications': 0.0, 'dt_adaptive': True, 'dt_last': 0.1635705137352299, 'backend':
↪'numba', 'stochastic': False, 'adaptive': True}, 'jit_count': {'make_stepper': 3,
↪'simulation': 0}}

Diagnostic information from third run:
{'controller': {'t_start': 0, 't_end': 1.0, 'solver_class': 'ScipySolver', 'solver_
↪start': '2022-07-06 10:12:41.824565', 'profiler': {'solver': 0.003875641999997015,
↪'tracker': 3.7002000027541726e-05}, 'successful': True, 'stop_reason': 'Reached␣
↪final time', 'solver_duration': '0:00:00.003939', 't_final': 1.0}, 'package_version
↪': '0.20.0', 'solver': {'class': 'ScipySolver', 'pde_class': 'DiffusionPDE', 'dt':␣
↪None, 'steps': 50, 'stochastic': False, 'backend': 'numba'}, 'jit_count': {'make_
↪stepper': 1, 'simulation': 0}}
```

```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()

# try the explicit solver
solver1 = pde.ExplicitSolver(eq)
controller1 = pde.Controller(solver1, t_range=1, tracker=None)
sol1 = controller1.run(field, dt=1e-3)
sol1.label = "explicit solver"
print("Diagnostic information from first run:")
print(controller1.diagnostics)
print()

# try an explicit solver with adaptive time steps
solver2 = pde.ExplicitSolver(eq, scheme="runge-kutta", adaptive=True)
controller2 = pde.Controller(solver2, t_range=1, tracker=None)
sol2 = controller2.run(field, dt=1e-3)
sol2.label = "explicit, adaptive solver"
print("Diagnostic information from second run:")
print(controller2.diagnostics)
print()

# try the standard scipy solver
solver3 = pde.ScipySolver(eq)
controller3 = pde.Controller(solver3, t_range=1, tracker=None)
sol3 = controller3.run(field)
sol3.label = "scipy solver"
```

(continues on next page)

(continued from previous page)

```
print("Diagnostic information from third run:")
print(controller3.diagnostics)
print()

# plot both fields and give the deviation as the title
title = f"Deviation: {((sol1 - sol2)**2).average:.2g}, {((sol1 - sol3)**2).average:.
↪2g}"
pde.FieldCollection([sol1, sol2, sol3]).plot(title=title)
```

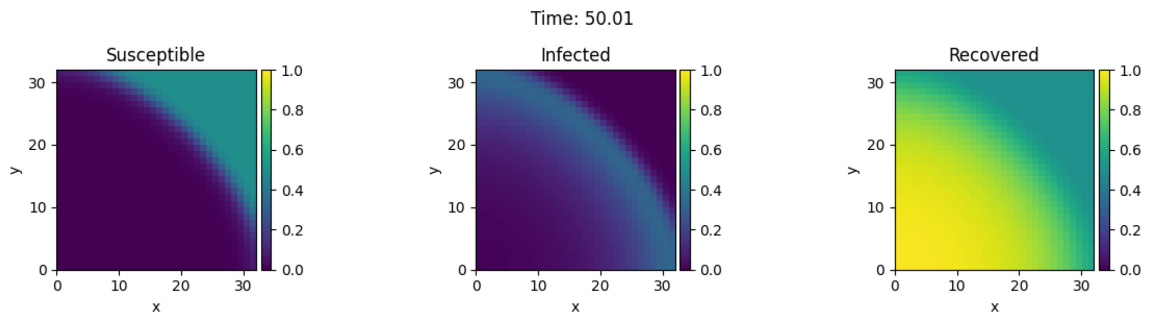
Total running time of the script: (0 minutes 13.042 seconds)

2.24 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:

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

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



Out:

```
  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 [00:03<2:33:05, 183.79s/it]
  0%|          | 0.03/50.0 [00:03<1:42:04, 122.57s/it]
  1%|          | 0.29/50.0 [00:03<10:33, 12.75s/it]
  4%|4         | 2.16/50.0 [00:03<01:24,  1.77s/it]
 15%|#4        | 7.27/50.0 [00:04<00:24,  1.74it/s]
 32%|###1      | 15.8/50.0 [00:04<00:10,  3.19it/s]
 51%|#####1    | 25.57/50.0 [00:05<00:05,  4.41it/s]
 72%|#######2  | 36.18/50.0 [00:06<00:02,  5.40it/s]
 95%|#########4| 47.33/50.0 [00:07<00:00,  6.19it/s]
 95%|#########4| 47.33/50.0 [00:08<00:00,  5.91it/s]
100%|##########| 50.0/50.0 [00:08<00:00,  6.25it/s]
100%|##########| 50.0/50.0 [00:08<00:00,  6.25it/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"
    ):
        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])


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(interval=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 8.161 seconds)

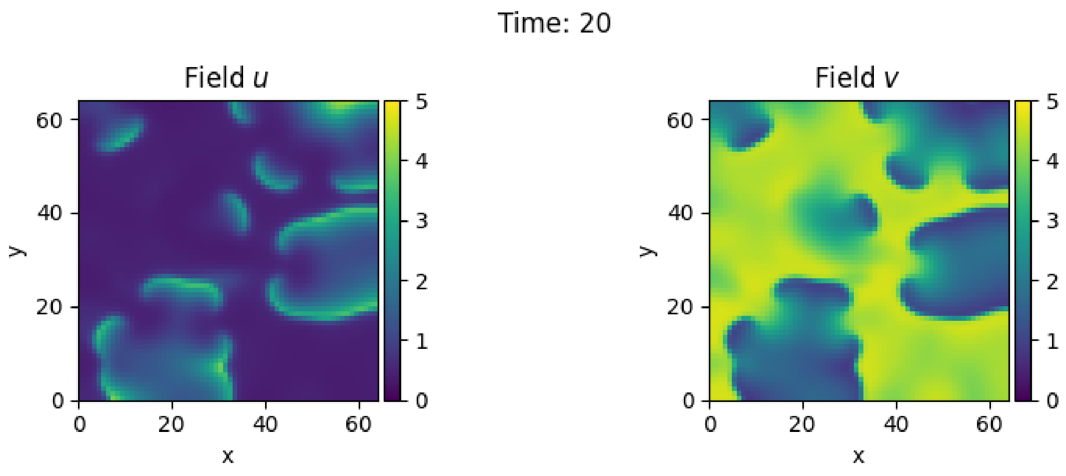
2.25 Brusselator - Using custom class

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.



```python
import numba as nb
import numpy as np

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


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

    def __init__(self, a=1, b=3, diffusivity=[1, 0.1], bc="auto_periodic_neumann"):
        self.a = a
        self.b = b
        self.diffusivity = diffusivity  # spatial mobility
        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
```

(continues on next page)

(continued from previous page)

```
        return rhs

    def _make_pde_rhs_numba(self, state):
        """nunmba-compiled implementation of the PDE"""
        d0, d1 = self.diffusivity
        a, b = self.a, self.b
        laplace = state.grid.make_operator("laplace", bc=self.bc)

        @nb.jit
        def pde_rhs(state_data, t):
            u = state_data[0]
            v = state_data[1]

            rate = np.empty_like(state_data)
            rate[0] = d0 * laplace(u) + a - (1 + b) * u + v * u**2
            rate[1] = d1 * laplace(v) + b * u - v * u**2
            return rate

        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(interval=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 16.167 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. They allow to exploit 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 are 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 here specify the used coordinate systems explictely:

Polar coordinates

Polar coordinates describe points by a radius r and an angle ϕ in a two-dimensional coordinates 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 only depend on the radial coordinate r. Note that vector and tensor fields can still have components in the polar 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 only depend on the radial coordinate r. Note that vector and tensor fields can still have components in the two angular direction.

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

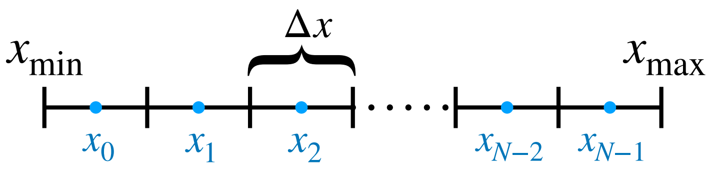
Cylindrical coordinates describe points by a radius r, an axial coordinate z, and a polar angle ϕ. The conversion to ordinary Cartesian coordinates reads

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

The associated symmetric grid `CylindricalSymGrid` assumes that fields only depend 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 though 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 difference. This requires the introducing of 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.

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 and a more advanced Runge-Kutta scheme in the `ExplicitSolver` class. For the simple implementations of these explicit methods, the user needs to specify a time step, which will be kept fixed. One problem with explicit solvers is that they require small time steps for some PDEs, which are then often called 'stiff PDEs'. 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 here 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 instance 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 coordinate 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 specifying the values at the discrete points of the grid defined in the previous section. Most PDEs discussed in the package describe a scalar variable, which can be encoded th 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 values allow to set the value of the grid, 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 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.

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 first need to generate an initial condition, i.e., the initial values of the fields that are evolved forward in time by the PDE. This field also defined 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 `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 differnt 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_lower = {"value": 4}
bc_upper = {"derivative": 2}
bc = [bc_lower, bc_upper]

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 it suffices to give one condition if both sides of the axis require the same condition. For instance, to enforce a value of *3* on both side, one could simply use `bc = {'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_x = [{"derivative": 0.1}, {"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=[bc_x, 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.

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.

There exist also special boundary conditions that impose a time-dependent value (`bc='value_expression'`) of the field or its derivative (`bc='derivative_expression'`). Beyond the spatial coordinates that are already supported for the constant conditiosn above, the expressions of these boundary conditions can depend on the time variable `t`. Note that PDEs need to supply the current time 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.

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)
bc = ["periodic", {"derivative": 0}]
field.laplace(bc)
```

For convenience, this typical situation can be described with the special boundary condition *natural*, e.g., calling the Laplace operator using `field.laplace("natural")` is identical to the example above. Alternatively, this condition can be called *auto_periodic_neumann* to stress that this chooses between periodic and Neumann boundary conditions automatically. 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.

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 conditons 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*. Finally, expressions for equations in `PDE` can explicitely 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):
        """ numpy implementation of the evolution equation """
```

(continues on next page)

(continued from previous page)

```
        state_lapacian = state.laplace(bc="auto_periodic_neumann")
        state_gradient = state.gradient(bc="auto_periodic_neumann")
        return (- state_lapacian.laplace(bc="auto_periodic_neumann")
                - state_lapacian
                - 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
        for the actual field and its second derivative """
        self.diffusivity = diffusivity
        self.bc = bc
        self.bc_laplace = bc_laplace

    def evolution_rate(self, state, t=0):
        """ numpy implementation of the evolution equation """
        state_lapacian = state.laplace(bc=self.bc)
        state_gradient = state.gradient(bc=self.bc)
        return (- state_lapacian.laplace(bc=self.bc_laplace)
                - state_lapacian
                - 0.5 * self.diffusivity * (state_gradient @ state_gradient))
```

We here replaced the call to `to_scalar('squared_sum')` by 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.

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 numba-accelerated 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'`. However, 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_interpolator_compiled(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 with `numba`.

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 Numba-accelerated 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:

```
from pde.tools.numba import jit


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
        for the actual field and its second derivative """
        self.diffusivity = diffusivity
        self.bc = bc
        self.bc_laplace = bc_laplace


    def evolution_rate(self, state, t=0):
        """ numpy implementation of the evolution equation """
        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_pde_rhs_numba(self, state):
        """ the numba-accelerated evolution equation """
        # make attributes locally available
        diffusivity = self.diffusivity

        # create operators
        laplace_u = state.grid.make_operator("laplace", bc=self.bc)
        gradient_u = state.grid.make_operator("gradient", bc=self.bc)
        laplace2_u = state.grid.make_operator("laplace", bc=self.bc_laplace)
        dot = VectorField(state.grid).make_dot_operator()
```

(continues on next page)

(continued from previous page)

```
        @jit
        def pde_rhs(state_data, t=0):
            """ compiled helper function evaluating right hand side """
            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_pde_rhs_numba()` method. This method expects an example of the state class (e.g., an instance of `ScalarField`) and 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 `numba` freezes 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 that is compiled using the `jit` decorator. Here, we use the implementation shipped with *py-pde*, which sets some default values. However, we could have also used the usual numba implementation. It is important that the implementation of the evolution rate only uses python constructs that numba can compile.

One advantage of the numba compiled implementation is that we can now use loops, which will be much faster than their python equivalents. For instance, we could have written the dot product in the last line as an explicit loop:

```
[...]

    def _make_pde_rhs_numba(self, state):
        """ the numba-accelerated evolution equation """
        # make attributes locally available
        diffusivity = self.diffusivity

        # create operators
        laplace_u = state.grid.make_operator("laplace", bc=self.bc)
        gradient_u = state.grid.make_operator("gradient", bc=self.bc)
        laplace2_u = state.grid.make_operator("laplace", bc=self.bc_laplace)
        dot = VectorField(state.grid).make_dot_operator()
        dim = state.grid.dim

        @jit
        def pde_rhs(state_data, t=0):
            """ compiled helper function evaluating right hand side """
            state_lapacian = laplace_u(state_data)
            state_grad = gradient_u(state_data)
            result = - laplace2_u(state_lapacian) - state_lapacian

            for i in range(state_data.size):
                for j in range(dim):
                    result.flat[i] -= diffusivity / 2 * state_grad[j].flat[i]**2

            return result

        return pde_rhs
```

Here, we extract the total number of elements in the state using its `size` attribute and we obtain the dimensionality of the space from the grid attribute `dim`. Note that we access numpy arrays using their `flat` attribute to provide an implementation that works for all dimensions.

3.3.6 Configuration parameters

Configuration parameters affect how the package behaves. They can be set using a dictionary-like interface of the configuration `config`, which can be imported from the base package. Here is a list of all configuration options that can be adjusted in the package:

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

numba.fastmath
Determines whether the fastmath flag is set during compilation. This affects the precision of the mathematical calculations. **(Default value: True)**

numba.parallel
Determines whether multiple cores are used in numba-compiled code. **(Default value: True)**

numba.parallel_threshold
Minimal number of support points before multithreading or multiprocessing is enabled in the numba compilations. **(Default value: 65536)**

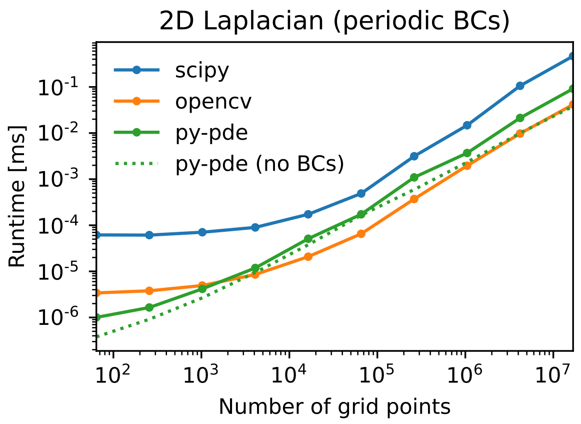
Tip: To disable parallel computing in the package, the following code could be added to the start of the script:

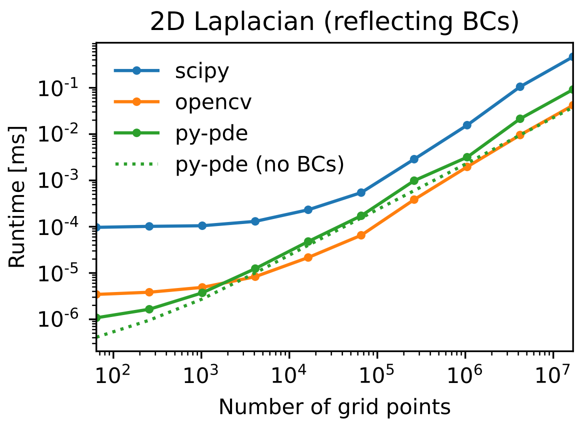
```
from pde import config
config['numba.parallel'] = False

# actual code using py-pde
```

3.4 Performance

3.4.1 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 are just-in-time compiled using `numba`, many operations of the package proceed at performances close to most compiled languages. For instance, a simple Laplace operator applied to fields defined on a Cartesian grid has performance that is similar to the operators supplied by the popular 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 lower in the *py-pde* package, so that the performance on small grids is particularly good. 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.

3.4.2 Improving performance

Factors influencing the performance of the package include the compiler used for `numpy`, `scipy`, and of course `numba`. 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 py37-numpy +gcc8+openblas
port install py37-scipy +gcc8+openblas
port install py37-numba +tbb
```

3.5 Contributing code

3.5.1 Structure of the package

The functionality of the `pde` package is split into multiple sub-package. 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`.

3.5.2 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`. It is also possible to only use parts of the package, e.g., the discretized differential operators from `operators`.

New operators can be associated with grids by registering them using `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 `Boundaries` 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 `Boundaries`. This operator would be registered with the grid by calling `CylindricalSymGrid.register_operator("operator", make_operator)`, where the first argument is the name of the operator and the second argument is the factory function.

3.5.3 Design choices

The data layout of field classes (subclasses of `FieldBase`) was chosen to allow for a simple decomposition of different fields and tensor components. Consequently, the 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.

3.5.4 Coding style

The coding style is enforced using 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, which are typically contained in separate modules in sub-packages called `tests`. These tests can be ran using scripts in the `tests` 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.

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 a lot of `numba` just-in-time compilation 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.

3.5.5 Running unit tests

The *pde* package contains several unit tests, typically contained in sub-module `tests` in the folder of a given module. 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 `tests`. The most basic script is `tests_run.sh`, which uses `pytest` to run the tests in the sub-modules of the *pde* package. Clearly, the python package `pytest` needs to be installed. There are also additional scripts that for instance run tests in parallel (need the python package `pytest-xdist` installed), measure test coverage (need 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.

3.6 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
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
```

3.7 Code of Conduct

3.7.1 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.

3.7.2 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

3.7.3 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.

3.7.4 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.

3.7.5 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.

3.7.6 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 classes and methods for solving partial differential equations.

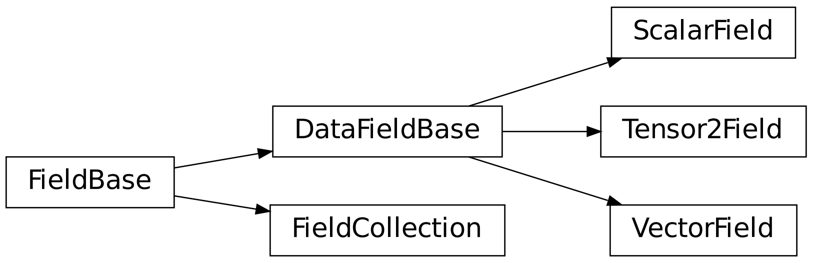
Subpackages:

4.1 pde.fields package

Defines fields, which contain the actual data stored on a discrete grid.

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:

4.1.1 pde.fields.base module

Defines base classes of fields, which are discretized on grids

class DataFieldBase(*grid: GridBase, data: Optional[Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]], str]] = 'zeros', *, label: str = None, dtype=None, with_ghost_cells: bool = 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

add_interpolated(*point: ndarray, amount: Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]]]*) → None

deprecated alias of method *insert*

property average: Union[int, float, complex, ndarray]

determine the average of data

This is calculated by integrating each component of the field over space and dividing by the grid volume

copy(**, label: str = None, dtype=None*) → TDataField

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 or the dtype of the current field is used.

property data_shape: Tuple[int, ...]

the shape of the data at each grid point

Type

tuple

property fluctuations: Union[int, float, complex, ndarray]

fluctuations over the entire space.

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
`ndarray`

classmethod from_state(*attributes: Dict[str, Any]*, *data: Optional[ndarray] = None*) → TDataField

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

get_boundary_values(*axis: int*, *upper: bool*, *bc: Optional[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]]] = None*) → Union[int, float, complex, ndarray]

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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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: int*) → Type[*DataFieldBase*]

return a `DataFieldBase` subclass describing a field with a given rank

Parameters
rank (*int*) – The rank of the tensor field

get_image_data(*scalar: str = 'auto'*, *transpose: bool = False*, ***kwargs*) → Dict[str, Any]

return data for plotting an image of the field

Parameters

- **scalar** (*str or int*) – The method for extracting scalars as described in `DataFieldBase.to_scalar()`.
- **transpose** (*bool*) – Determines whether the transpose of the data should is plotted

- `**kwargs` – Additional parameters are forwarded to *grid.get_image_data*

Returns
Information useful for plotting an image of the field

Return type
dict

`get_line_data`(*scalar: str = 'auto', extract: str = 'auto'*) → Dict[str, Any]

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`(***kwargs*) → Dict[str, Any]

return data for a vector plot of the field

Parameters
`**kwargs` – Additional parameters are forwarded to *grid.get_image_data*

Returns
Information useful for plotting an vector field

Return type
dict

`insert`(*point: ndarray, amount: Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]]]*) → None

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.

`abstract property integral: Union[int, float, complex, ndarray]`

`interpolate`(*point: ndarray, *, backend: str = 'numba', method: str = 'linear', fill: Optional[Number] = None, **kwargs*) → Union[int, float, complex, ndarray]

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.

- **backend** (*str*) – The accepted values "scipy" and "numba" determine the backend that is used for the interpolation.
- **method** (*str*) – Determines the method being used for interpolation. Typical values that are "nearest" and "linear", but the supported values depend on the chosen *backend*.
- **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.
- ****kwargs** – Additional keyword arguments are forwarded to the method `DataFieldBase.make_interpolator()`.

Returns
the values of the field

Return type
ndarray

interpolate_to_grid(*grid: GridBase, *, backend: str = 'numba', method: str = 'linear', fill: Optional[Number] = None, label: str = None*) → TDataField

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.
- **backend** (*str*) – The accepted values "scipy" and "numba" determine the backend that is used for the interpolation.
- **method** (*str*) – Determines the method being used for interpolation. Typical values that are "nearest" and "linear", but the supported values depend on the chosen *backend*.
- **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.

property magnitude: float

determine the 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_interpolator(*method: str = 'linear', *, fill: Number = None, backend: str = 'numba', **kwargs*) → Callable[[ndarray, ndarray], Union[int, float, complex, ndarray]]

returns a function that can be used to interpolate values.

Parameters

- **backend** (*str*) – The accepted values *scipy* and *numba* determine the backend that is used for the interpolation.
- **method** (*str*) – Determines the method being used for interpolation. Typical values that are "nearest" and "linear", but the supported values depend on the chosen *backend*.

- **`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.
- **`**kwargs`** – Additional keyword arguments are passed to the individual interpolator methods and can be used to further affect the behavior.

The scipy implementations use scipy.interpolate.RegularGridInterpolator and thus do not respect boundary conditions. Additional keyword arguments are directly forwarded to the constructor of *RegularGridInterpolator*.

The numba implementation respect boundary conditions, which can be set using the *bc* keywords argument. Supported values are the same as for the operators, e.g., the Laplacian. If no boundary conditions are specified, natural boundary conditions are assumed, which are periodic conditions for periodic axes and Neumann conditions otherwise.

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

`plot`(*kind: str = 'auto', *args, title: str = None, filename: str = None, action: str = 'auto', ax_style: Optional[Dict[str, Any]] = None, fig_style: Optional[Dict[str, Any]] = None, ax=None, **kwargs*) → *PlotReference*

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 *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.
- **`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 in 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.

Returns
Instance that contains information to update the plot with new data later.

Return type
`PlotReference`

classmethod random_colored(*grid: GridBase*, *exponent: float = 0*, *scale: float = 1*, ***, *label: str = None*, *dtype=None*, *rng: Optional[Generator] = None*) → TDataField

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 components of vector or 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 field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double*.
- **rng** (Generator) – Random number generator (default: `default_rng()`)

classmethod random_harmonic(*grid: ~GridBase*, *modes: int = 3*, *harmonic=<ufunc 'cos'>*, *axis_combination=<ufunc 'multiply'>*, ***, *label: ~str = None*, *dtype=None*, *rng: ~Optional[~numpy.random._generator.Generator] = None*) → TDataField

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

$$c_i(\mathbf{x}) = \prod_{\alpha=1}^N \sum_{j=1}^M a_{ij\alpha} \cos\left(\frac{2\pi x_\alpha}{jL_\alpha}\right) \;,$$

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 field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double*.
- **rng** (Generator) – Random number generator (default: `default_rng()`)

classmethod random_normal(*grid: GridBase, mean: float = 0, std: float = 1, *, scaling: str = 'none', label: str = None, dtype=None, rng: Optional[Generator] = None*) → TDataField

create field with normal distributed random values

These values are uncorrelated in space. 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, `ScalarField.random_normal(grid, 0, 1 + 1j)` creates a complex field where the real and imaginary parts are chosen from a standard normal distribution.

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 volumes).
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. If omitted, it defaults to *double* if both *mean* and *std* are real, otherwise it is *complex*.
- **rng** (`Generator`) – Random number generator (default: `default_rng()`)

classmethod random_uniform(*grid: GridBase, vmin: float = 0, vmax: float = 1, *, label: str = None, dtype=None, rng: Optional[Generator] = None*) → TDataField

create field with uniform distributed random values

These values are uncorrelated in space.

Parameters

- **grid** (*GridBase*) – Grid defining the space on which this field is defined
- **vmin** (*float*) – Smallest possible random value
- **vmax** (*float*) – Largest random value
- **label** (*str, optional*) – Name of the 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()`)

rank: int

set_ghost_cells(*bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]], *, normal: bool = False, args=None*) → None

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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **args** – Additional arguments that might be supported by special boundary conditions.

smooth (*sigma: float = 1*, **, *out: Optional[TDataField] = None*, *label: str = None*) → TDataField

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.

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.

abstract to_scalar (*scalar: str = 'auto'*, **, *label: str = None*) → *ScalarField*

classmethod unserialize_attributes (*attributes: Dict[str, str]*) → Dict[str, Any]

unserializes the given attributes

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

class FieldBase (*grid: GridBase*, *data: ndarray*, **, *label: str = 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: Callable*, *out: Optional[TField] = None*, *label: str = None*) → TField

applies a function to the data and returns it as a field

Parameters

- **func** (*callable or str*) – The (vectorized) function being applied to the data or the name of an operator that is defined for the grid of this field.
- **out** (FieldBase, *optional*) – Optional field into which the data is written
- **label** (*str, optional*) – Name of the returned field

Returns

Field with new data. This is stored at *out* if given.

assert_field_compatible(*other: FieldBase*, *accept_scalar: bool = 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`.

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() → TField

returns complex conjugate of the field

abstract copy(**, label: str = None, dtype=None*) → TField

property data: ndarray

discretized data at the support points

Type

`ndarray`

property dtype

returns the numpy dtype of the underlying data

classmethod from_file(*filename: str*) → *FieldBase*

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: Dict[str, Any], data: Optional[ndarray] = None*) → *FieldBase*

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

abstract get_image_data() → Dict[str, Any]

abstract get_line_data(*scalar: str = 'auto', extract: str = 'auto'*) → Dict[str, Any]

property grid: *GridBase*

The grid on which the field is defined

Type
  GridBase

property imag: TField

Imaginary part of the field

Type
  FieldBase

property is_complex: bool

whether the field contains real or complex data

Type
  bool

property label: Optional[str]

the name of the field

Type
  str

abstract plot(**args, **kwargs*)

plot_interactive(*viewer_args: Optional[Dict[str, Any]] = 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

property real: TField

Real part of the field

Type
FieldBase

to_file (*filename: str*, ***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

classmethod unserialize_attributes (*attributes: Dict[str, str]*) → Dict[str, Any]

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

4.1.2 pde.fields.collection module

Defines a collection of fields to represent multiple fields defined on a common grid.

class FieldCollection(*fields: Sequence[DataFieldBase], *, copy_fields: bool = False, label: str = None, labels: Optional[List[str]] = None, dtype=None*)

Bases: `FieldBase`

Collection of fields defined on the same grid

Note that all fields in the same collection must have the same data type. This might lead to upcasting, where for instance a combination of a real-valued and a complex-valued field will be both stored as complex fields.

Parameters

- **fields** – Sequence 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.
- **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.

assert_field_compatible(*other: FieldBase, accept_scalar: bool = 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: str = None, dtype=None*) → *FieldCollection*

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.

property fields: List[*DataFieldBase*]

the fields of this collection

Type
list

classmethod from_scalar_expressions(*grid: GridBase*, *expressions: Sequence[str]*, ***, *label: str = None*, *labels: Optional[Sequence[str]] = None*, *dtype=None*) → *FieldCollection*

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*.
- **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.

classmethod from_state(*attributes: Dict[str, Any]*, *data: Optional[ndarray] = None*) → *FieldCollection*

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

get_image_data(*index: int = 0*, ***kwargs*) → Dict[str, Any]

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: int = 0*, *scalar: str = 'auto'*, *extract: str = 'auto'*) → Dict[str, Any]

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: GridBase, *, backend: str = 'numba', method: str = 'linear', fill: Optional[Number] = None, label: str = None*) → *FieldCollection*

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.
- **backend** (*str*) – The accepted values "scipy" and "numba" determine the backend that is used for the interpolation.
- **method** (*str*) – Determines the method being used for interpolation. Typical values that are "nearest" and "linear", but the supported values depend on the chosen *backend*.
- **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: ndarray

scalar magnitudes of all fields

Type
`ndarray`

plot (*kind: Union[str, Sequence[str]] = 'auto', resize_fig=None, figsize='auto', arrangement='horizontal', subplot_args=None, *args, title: str = None, constrained_layout: bool = True, filename: str = None, action: str = 'auto', fig_style: Optional[Dict[str, Any]] = None, fig=None, **kwargs*) → List[*PlotReference*]

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*, or *interactive*. 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.
- **resize_fig** (*bool*) – Whether to resize the figure to adjust to the number of panels
- **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*) – Determines how the subpanels will be arranged. The default value *horizontal* places all subplots next to each other. The alternative value *vertical* puts them below each other.
- **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.
- **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.
- ****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`

classmethod scalar_random_uniform (*num_fields: int, grid: GridBase, vmin: float = 0, vmax: float = 1, *, label: str = None, labels: Optional[Sequence[str]] = None*) → *FieldCollection* 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

smooth(*sigma: float = 1*, ***, *out: Optional[FieldCollection] = None*, *label: str = None*) → *FieldCollection*

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.

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
Field collection with smoothed data, stored at *out* if given.

classmethod unserialize_attributes(*attributes: Dict[str, str]*) → Dict[str, Any]

unserializes the given attributes

Parameters
attributes (*dict*) – The serialized attributes

Returns
The unserialized attributes

Return type
dict

4.1.3 pde.fields.scalar module

Defines a scalar field over a grid

class ScalarField(*grid: GridBase*, *data: Optional[Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]], str]] = 'zeros'*, ***, *label: str = None*, *dtype=None*, *with_ghost_cells: bool = 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: GridBase, expression: str, *, label: str = None, dtype=None*) → *ScalarField*

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*.
- **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.

classmethod from_image(*path: Union[Path, str], bounds=None, periodic=False, *, label: str = None*) → *ScalarField*

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

gradient(*bc: Optional[BoundariesData], out: Optional['VectorField'] = None, **kwargs*) → *VectorField*

apply gradient operator and return result as a field

Parameters

- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`VectorField, optional`) – Optional vector field to which the result is written.
- **label** (`str, optional`) – Name of the returned field

Returns
result of applying the operator

Return type
`VectorField`

gradient_squared(*bc: Optional[BoundariesData]*, *out: Optional[ScalarField] = None*, ***kwargs*) → *ScalarField*

apply squared gradient operator and return result as a field

This evaluates $|\nabla\phi|^2$ for the scalar field ϕ

Parameters

- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`ScalarField, optional`) – Optional vector field to which the result is written.
- **label** (`str, optional`) – Name of the returned field
- **central** (`bool`) – Determines whether a central difference approximation is used for the gradient operator or not. If not, the squared gradient is calculated as the mean of the squared values of the forward and backward derivatives, which thus includes the value at a support point in the result at the same point.

Returns
the squared gradient of the field

Return type
`ScalarField`

property integral: Union[int, float, complex]

integral of the scalar field over space

Type
Number

laplace(*bc: Optional[BoundariesData]*, *out: Optional[ScalarField] = None*, ***kwargs*) → *ScalarField*

apply Laplace operator and return result as a field

Parameters

- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`ScalarField`, *optional*) – Optional scalar field to which the result is written.
- **label** (*str, optional*) – Name of the returned field
- **backend** (*str*) – The backend (e.g., 'numba' or 'scipy') used for this operator.

Returns
the Laplacian of the field

Return type
ScalarField

project (*axes: Union[str, Sequence[str]], method: str = 'integral', label: str = None*) → *ScalarField*

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. This can be either 'integral' to integrate over the removed axes or 'average' to perform an average instead.
- **label** (*str, optional*) – The label of the returned field

Returns
The projected data in a scalar field with a subgrid of the original grid.

Return type
ScalarField

rank: int = 0

slice (*position: Dict[str, float], *, method: str = 'nearest', label: str = None*) → *ScalarField*

slice data at a given position

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. *nearest* takes data from cells defined on the grid.
- **label** (*str, optional*) – The label of the returned field

Returns
The sliced data in a scalar field with a subgrid of the original grid.

Return type
ScalarField

to_scalar(*scalar: Union[str, Callable] = 'auto'*, ***, *label: str = None*) → *ScalarField*
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

4.1.4 pde.fields.tensorial module

Defines a tensorial field of rank 2 over a grid

class Tensor2Field(*grid: GridBase*, *data: Optional[Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]], str]] = 'zeros'*, ***, *label: str = None*, *dtype=None*, *with_ghost_cells: bool = False*)

Bases: *DataFieldBase*

Tensor field of rank 2 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

divergence(*bc: Optional[BoundariesData]*, *out: Optional[VectorField] = None*, ***kwargs*) → *VectorField*
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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`VectorField, optional`) – Optional scalar field to which the result is written.
- **label** (`str, optional`) – Name of the returned field
- ****kwargs** – Additional arguments affecting how the operator behaves. For instance, the argument *normal_bcs* can be used to control whether boundary conditions only specify normal components or not.

Returns
result of applying the operator

Return type
`VectorField`

dot (*other: Union[*VectorField, Tensor2Field*], out: Optional[Union[*VectorField, Tensor2Field*]] = None*, *, *conjugate:* bool *= True*, *label:* str *= 'dot product'*) → Union[*VectorField*, *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** (`VectorField or Tensor2Field`) – the second field
- **out** (`VectorField or 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 the dot operator

classmethod from_expression (*grid:* GridBase, *expressions: Sequence[Sequence[*str*]]*, *, *label:* str *= None*, *dtype=None*) → *Tensor2Field*

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*.
- **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.

property integral: ndarray

integral of each component over space

Type
ndarray

make_dot_operator(*backend: str = 'numba'*, ***, *conjugate: bool = True*) → Callable[[ndarray, ndarray, Optional[ndarray]], ndarray]

return operator calculating the dot product involving vector fields

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

Warning: This function does not check types or dimensions.

Parameters
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. Note that the returned function is jitted with numba for speed.

plot_components(*kind: str = 'auto'*, **args*, *title: str = None*, *constrained_layout: bool = True*, *filename: str = None*, *action: str = 'auto'*, *fig_style: Optional[Dict[str, Any]] = None*, *fig=None*, ***kwargs*) → List[List[PlotReference]]

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: int = 2`

`symmetrize`(*make_traceless: bool = False*, *inplace: bool = False*) → *Tensor2Field*

symmetrize the tensor field in place

Parameters

- **`make_traceless`** (*`bool`*) – Determines whether the result is also traceless
- **`inplace`** (*`bool`*) – Flag determining whether to symmetrize the current field or return a new one

Returns
result of the operation

Return type
`Tensor2Field`

`to_scalar`(*scalar: str = 'auto'*, ***, *label: str = 'scalar `{scalar}`'*) → *ScalarField*

return a scalar field by applying *method*

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: str = 'trace'*) → *ScalarField*

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 (*label: str = 'transpose'*) → *Tensor2Field*

return the transpose of the tensor field

Parameters
label (*str, optional*) – Name of the returned field

Returns
transpose of the tensor field

Return type
Tensor2Field

4.1.5 pde.fields.vectorial module

Defines a vectorial field over a grid

class VectorField (*grid:* GridBase, *data: Optional[Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]], str]] = 'zeros', *, label: str = None, dtype=None, with_ghost_cells: bool = False*)

Bases: *DataFieldBase*

Vector 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

divergence (*bc: Optional[BoundariesData], out: Optional[*ScalarField*] = None, **kwargs*) → *ScalarField*

apply divergence operator and return result as a field

Parameters

- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`ScalarField`, *optional*) – Optional scalar field to which the result is written.
- **label** (*str*, *optional*) – Name of the returned field
- ****kwargs** – Additional arguments affecting how the operator behaves. For instance, the argument *normal_bcs* can be used to control whether boundary conditions only specify normal components or not.

Returns
result of applying the operator

Return type
ScalarField

dot(*other: Union[*VectorField*,* Tensor2Field*]*, *out: Optional[Union[*ScalarField*,* VectorField*]] = None*, ***, *conjugate:* bool *= True*, *label:* str *= 'dot product'*) → Union[*ScalarField*, *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** (`ScalarField` *or* `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

classmethod from_expression(*grid:* GridBase, *expressions: Sequence[*str*]*, ***, *label:* str *= None*, *dtype=None*) → *VectorField*

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*.

- **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.

classmethod from_scalars (*fields: List[ScalarField], *, label: str = None, dtype=None*) → *VectorField*

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
VectorField

get_vector_data (*transpose: bool = False, max_points: int = None, **kwargs*) → Dict[str, Any]

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 are forwarded to *grid.get_image_data*

Returns
Information useful for plotting an vector field

Return type
dict

gradient (*bc: Optional[BoundariesData], out: Optional['Tensor2Field'] = None, **kwargs*) → *Tensor2Field*

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** – 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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`VectorField, optional`) – Optional vector field to which the result is written.
- **label** (`str, optional`) – Name of the returned field
- ****kwargs** – Additional arguments affecting how the operator behaves. For instance, the argument *normal_bcs* can be used to control whether boundary conditions only specify normal components or not.

Returns
result of applying the operator

Return type
`Tensor2Field`

property integral: ndarray
integral of each component over space

Type
`ndarray`

laplace (*bc: Optional[BoundariesData], out: Optional[*VectorField*] = None, **kwargs*) → *VectorField*
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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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** (`VectorField, optional`) – Optional vector field to which the result is written.
- **label** (`str, optional`) – Name of the returned field
- ****kwargs** – Additional arguments affecting how the operator behaves. For instance, the argument *normal_bcs* can be used to control whether boundary conditions only specify normal components or not.

Returns
result of applying the operator

Return type
VectorField

make_dot_operator(*backend: str = 'numba'*, ***, *conjugate: bool = True*) → Callable[[ndarray, ndarray, Optional[ndarray]], ndarray]

return operator calculating the dot product involving vector fields

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

Warning: This function does not check types or dimensions.

Parameters

- **backend** (*str*) – The backend (e.g., 'numba' or 'scipy') used for this 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. Note that the returned function is jitted with numba for speed.

make_outer_prod_operator(*backend: str = 'numba'*) → Callable[[ndarray, ndarray, Optional[ndarray]], ndarray]

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 'scipy') 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. Note that the returned function is jitted with numba for speed.

outer_product(*other: VectorField*, *out: Optional[Tensor2Field] = None*, ***, *label: str = None*) → *Tensor2Field*

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

rank: int = 1

to_scalar(*scalar: Union[str, int] = 'auto'*, ***, *label: str = 'scalar `{scalar}`'*) → *ScalarField*

return a scalar field by applying *method*

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

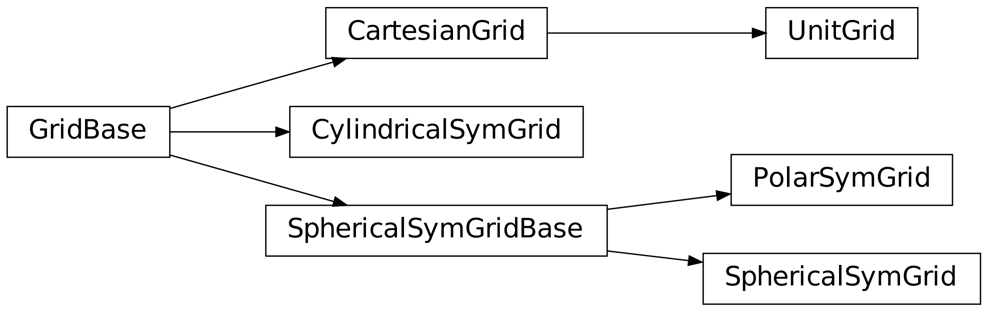
4.2 pde.grids package

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:

4.2.1 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 need to be applied at the end of each axis. Consequently, methods expecting boundary conditions typically receive a list of conditions for each axes:

```
field = ScalarField(UnitGrid([16, 16], periodic=[True, False]))
field.laplace(bc=[bc_x, bc_y])
```

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, which is done using a tuple of two conditions. 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 = ({"value": 2}, {"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 = ({"value": "y**2"}, {"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 = ({"type": "mixed", "value": 2, "const": 7},
        {"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 the 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 (instead of the tuple). For instance, `bc_y = {'value': 2}` imposes a value of *2* on both sides of the y-axis. Similarly, if all axes have the same boundary conditions, only one axis needs to be specified (instead of the list). For instance, the following example

```
field = 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 = 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.

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
- *`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
- *`MixedBC`*: Imposing the derivative of the field in the outward normal direction proportional to its value at the boundary
- *`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.

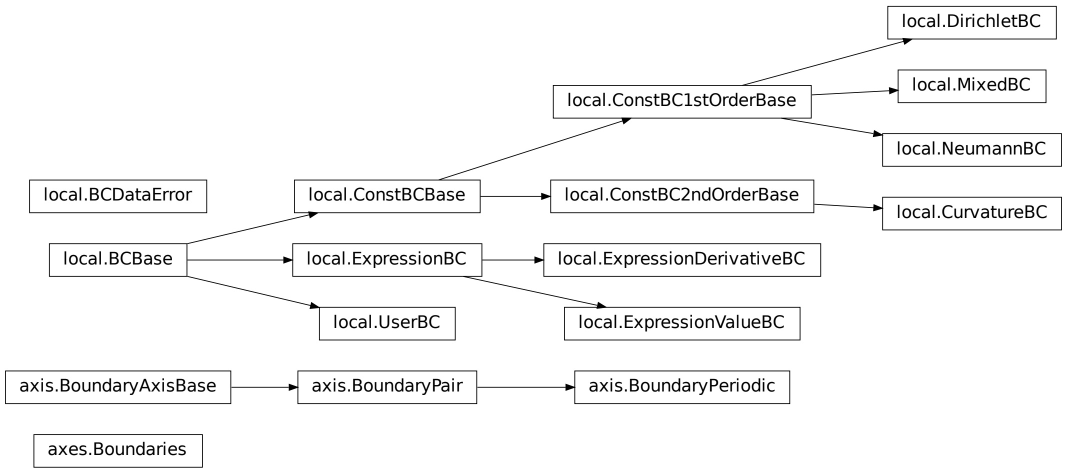
Boundaries for an axis:

- *BoundaryPair*: Uses the local boundary conditions to specify the two boundaries along an axis
- *BoundaryPeriodic*: Indicates that an axis has periodic boundary conditions

Boundaries for all axes of a grid:

- *Boundaries*: Collection of boundaries to describe conditions for all axes

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. It only defines *Boundaries*, which acts as a list of *BoundaryAxisBase*.

class Boundaries(*boundaries*)

Bases: `list`

class that bundles all boundary conditions for all axes

initialize with a list of boundaries

check_value_rank(*rank: int*) → None

check whether the values at the boundaries have the correct rank

Parameters

rank (*int*) – The tensorial rank of the field for this boundary condition

Throws:

RuntimeError: if any value does not have rank *rank*

copy() → *Boundaries*

create a copy of the current boundaries

extract_component(**indices*) → *Boundaries*

extracts the boundary conditions of the given component of the tensor.

Parameters

***indices** – One or two indices for vector or tensor fields, respectively

classmethod from_data(*grid: GridBase*, *boundaries*, *rank: int = 0*, *normal: bool = False*) → *Boundaries*

Creates all boundaries from given data

Parameters

- **grid** (*GridBase*) – The grid with which the boundary condition is associated
- **boundaries** (*str or list or tuple or dict*) – Data that describes the boundaries. This can either be a list of specifications for each dimension or a single one, which is then applied to all dimensions. The boundary for a dimensions can be specified by one of the following formats:
  - string specifying a single type for all boundaries
  - dictionary specifying the type and values for all boundaries
  - tuple pair specifying the low and high boundary individually
- **rank** (*int*) – The tensorial rank of the field for this boundary condition
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

classmethod get_help() → str

Return information on how boundary conditions can be set

get_mathematical_representation(*field_name: str = 'C'*) → str

return mathematical representation of the boundary condition

grid: ***GridBase***

GridBase: The grid for which the boundaries are defined

make_ghost_cell_setter() → Callable[[...], None]

return function that sets the ghost cells on a full array

property periodic: List[bool]

a boolean array indicating which dimensions are periodic according to the boundary conditions

Type

ndarray

set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → 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.

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`.

class BoundaryAxisBase(*low:* BCBase, *high:* BCBase)

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

get_data(*idx: Tuple[int, ...]*) → Tuple[float, Dict[int, float]]

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_mathematical_representation(*field_name: str = 'C'*) → Tuple[str, str]

return mathematical representation of the boundary condition

get_point_evaluator(*fill: Optional[ndarray] = None*) → Callable[[ndarray, Tuple[int, ...]], Union[int, float, complex, ndarray]]

return a function to evaluate values at a given point

The point can either be a point inside the domain or a virtual point right outside the domain

Parameters

fill (ndarray, optional) – Determines how values out of bounds are handled. If *None*, a *DomainError* is raised when out-of-bounds points are requested. Otherwise, the given value is returned.

Returns

A function taking a 1d array and an index as an argument,
returning the value of the array at this index.

Return type

function

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

make_derivative_evaluator(*order: int = 1*) → Callable[[ndarray, Tuple[int, ...]], Union[int, float, complex, ndarray]]

return a function to evaluate the derivative at a point

Parameters

order (*int*) – The order of the derivative

Returns

A function that can be called with the data array and a tuple indicating around what point the derivative is evaluated. The function returns the central finite difference at the point. The function takes boundary conditions into account if the point lies on the boundary.

Return type

function

make_ghost_cell_setter() → Callable[[...], None]

return function that sets the ghost cells for this axis on a full array

make_region_evaluator() → Callable[[ndarray, Tuple[int, ...]], Tuple[Union[int, float, complex, ndarray], Union[int, float, complex, ndarray], Union[int, float, complex, ndarray]]]

return a function to evaluate values in a neighborhood of a point

Returns

A function that can be called with the data array and a tuple indicating around what point the region is evaluated. The function returns the data values left of the point, at the point, and right of the point along the axis associated with this boundary condition. The function takes boundary conditions into account if the point lies on the boundary.

Return type

function

make_virtual_point_evaluators() → Tuple[Callable[[...], float], Callable[[...], float]]

returns two functions evaluating the value at virtual support points

Returns

Two functions that each take a 1d array as an argument and return the associated value at the virtual support point outside the lower and upper boundary, respectively.

Return type

tuple

property periodic: bool

whether the axis is periodic

Type

bool

set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → 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.

class BoundaryPair(*low: BCBase*, *high: BCBase*)

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: int*) → None

check whether the values at the boundaries have the correct rank

Parameters

rank (*`int`*) – The tensorial rank of the field for this boundary condition

Throws:

RuntimeError: if the value does not have rank *rank*

copy() → *BoundaryPair*

return a copy of itself, but with a reference to the same grid

extract_component(**indices*) → *BoundaryPair*

extracts the boundary pair of the given index.

Parameters

***indices** – One or two indices for vector or tensor fields, respectively

classmethod from_data(*grid: GridBase*, *axis: int*, *data*, *rank: int = 0*, *normal: bool = False*) → *BoundaryPair*

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
- **normal** (*`bool`*) – Flag indicating whether the condition is only applied in the normal direction.

Returns

the instance created from the data

Return type

`BoundaryPair`

Throws:

ValueError if *data* cannot be interpreted as a boundary pair

classmethod get_help() → str

Return information on how boundary conditions can be set

high: ***BCBase***

Boundary condition at upper end

Type
BCBase

low: ***BCBase***

Boundary condition at lower end

Type
BCBase

class BoundaryPeriodic(*grid:* GridBase, *axis: int*, *flip_sign: bool = False*)

Bases: *BoundaryPair*

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
- **flip_sign** (*bool*) – Impose different signs on the two sides of the boundary

check_value_rank(*rank: int*) → None

check whether the values at the boundaries have the correct rank

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

copy() → *BoundaryPeriodic*

return a copy of itself, but with a reference to the same grid

extract_component(**indices*) → *BoundaryPeriodic*

extracts the boundary pair of the given extract_component.

Parameters
***indices** – One or two indices for vector or tensor fields, respectively

property flip_sign

Whether different signs are imposed on the two sides of the boundary

Type
bool

high: ***BCBase***

Boundary condition at upper end

Type
BCBase

low: ***BCBase***

Boundary condition at lower end

Type
BCBase

get_boundary_axis(*grid: GridBase*, *axis: int*, *data*, *rank: int = 0*, *normal: bool = False*) → *BoundaryAxisBase*

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

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 different 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

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
- *NeumannBC*: Imposing the derivative of a field in the outward normal direction at the boundary based on a mathematical expression
- *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.

class BCBase(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*)

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

property axis_coord: float

value of the coordinate that defines this boundary condition

Type
float

check_value_rank(*rank: int*) → None

check whether the values at the boundaries have the correct rank

Parameters
rank (*int*) – The tensorial rank of the field for this boundary condition

Throws:
RuntimeError: if the value does not have rank *rank*

abstract copy(*upper: Optional[bool] = None*, *rank: int = None*) → *BCBase*

extract_component(**indices*)

extracts the boundary conditions for the given component

Parameters
***indices** – One or two indices for vector or tensor fields, respectively

classmethod from_data(*grid: GridBase*, *axis: int*, *upper: bool*, *data: Union[Dict, str, BCBase]*, ***, *rank: int = 0*, *normal: bool = False*) → *BCBase*

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

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: GridBase, axis: int, upper: bool, data: Dict[str, Any], *, rank: int = 0, normal: bool = False*) → *BCBase*

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

classmethod from_str(*grid: GridBase, axis: int, upper: bool, condition: str, *, rank: int = 0, normal: bool = False, **kwargs*) → *BCBase*

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- ****kwargs** – Additional arguments passed to the constructor

get_data(*idx: Tuple[int, ...]*) → Tuple[float, Dict[int, float]]

classmethod get_help() → str

Return information on how boundary conditions can be set

get_mathematical_representation(*field_name: str = 'C'*) → str

return mathematical representation of the boundary condition

get_virtual_point(*arr, idx: Optional[Tuple[int, ...]] = None*) → float

homogeneous: bool

determines whether the boundary condition depends on space

Type

bool

make_adjacent_evaluator() → Callable[[ndarray, int, Tuple[int, ...]], float]

make_ghost_cell_setter() → Callable[[...], None]

return function that sets the ghost cells for this boundary

abstract make_virtual_point_evaluator() → Callable[[...], float]

names: List[str]

identifiers used to specify the given boundary class

Type
list

property periodic: bool

whether the axis is periodic

Type
bool

abstract set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → None

set the ghost cell values for this boundary

exception BCDataError

Bases: `ValueError`

exception that signals that incompatible data was supplied for the BC

class ConstBC1stOrderBase(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, ndarray, str] = 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **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_data(*idx: Tuple[int, ...]*) → Tuple[float, Dict[int, float]]

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: Optional[Tuple[int, ...]] = None*) → float
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

abstract get_virtual_point_data(*compiled: bool = False*) → Tuple[Any, Any, int]

make_adjacent_evaluator() → Callable[[ndarray, int, Tuple[int, ...]], float]
returns a function evaluating the value adjacent to a given point

Returns
A function with signature (arr_1d, i_point, bc_idx), where *arr_1d* is the one-dimensional data array (the data points along the axis perpendicular to the boundary), *i_point* is the index into this array for the current point and bc_idx are the remaining indices of the current point, which indicate the location on the boundary plane. The result of the function is the data value at the adjacent point along the axis associated with this boundary condition in the upper (lower) direction when *upper* is True (False).

Return type
function

make_virtual_point_evaluator() → Callable[[...], float]
returns a function evaluating the value at the virtual support point

Returns
A function that takes the data array and an index marking the current point, which is assumed to be a virtual point. The result is the data value at this point, which is calculated using the boundary condition.

Return type
function

set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → None
set the ghost cell values for this boundary

Parameters

- **data_full** (*ndarray*) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

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 ConstBC2ndOrderBase(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, ndarray, str] = 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **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_data(*idx: Tuple[int, ...]*) → Tuple[float, Dict[int, float]]

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: Optional[Tuple[int, ...]] = None*) → float

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

abstract get_virtual_point_data() → Tuple[Any, Any, int, Any, int]

return data suitable for calculating virtual points

Returns
the data associated with this virtual point

Return type
tuple

make_adjacent_evaluator() → Callable[[ndarray, int, Tuple[int, ...]], float]

returns a function evaluating the value adjacent to a given point

Returns
A function with signature (arr_1d, i_point, bc_idx), where *arr_1d* is the one-dimensional data array (the data points along the axis perpendicular to the boundary), *i_point* is the index into this array for the current point and bc_idx are the remaining indices of the current point, which indicate the location on the boundary plane. The result of the function is the data value at the adjacent point along the axis associated with this boundary condition in the upper (lower) direction when *upper* is True (False).

Return type
function

make_virtual_point_evaluator() → Callable[[...], float]

returns a function evaluating the value at the virtual support point

Returns
A function that takes the data array and an index marking the current point, which is assumed to be a virtual point. The result is the data value at this point, which is calculated using the boundary condition.

Return type
function

set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → None

set the ghost cell values for this boundary

Parameters

- **data_full** (`ndarray`) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

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 ConstBCBase(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, ndarray, str] = 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **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: Optional[bool] = None*, *rank: int = None*, *value: Optional[Union[float, ndarray, str]] = None*) → *ConstBCBase*

return a copy of itself, but with a reference to the same grid

extract_component(**indices*)

extracts the boundary conditions for the given component

Parameters
***indices** – One or two indices for vector or tensor fields, respectively

link_value(*value: ndarray*)

link value of this boundary condition to external array

property value: ndarray

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: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, ndarray, str] = 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **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: str = 'C'*) → str

return mathematical representation of the boundary condition

get_virtual_point_data() → Tuple[ndarray, ndarray, int, ndarray, int]

return data suitable for calculating virtual points

Returns
the data structure associated with this virtual point

Return type
tuple

homogeneous: bool

determines whether the boundary condition depends on space

Type
bool

names: List[str] = ['curvature', 'second_derivative', 'extrapolate']

identifiers used to specify the given boundary class

Type
list

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 DirichletBC(*grid: GridBase, axis: int, upper: bool, *, rank: int = 0, normal: bool = False, value: Union[float, ndarray, str] = 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **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: str = 'C'*) → str

return mathematical representation of the boundary condition

get_virtual_point_data(*compiled: bool = False*) → Tuple[Any, Any, int]

return data suitable for calculating virtual points

Parameters
compiled (*bool*) – Flag indicating whether a compiled version is required, which automatically takes updated values into account when it is used in numba-compiled code.

Returns
the data structure associated with this virtual point

Return type
`BC1stOrderData`

homogeneous: bool

determines whether the boundary condition depends on space

Type
bool

names: List[str] = ['value', 'dirichlet']

identifiers used to specify the given boundary class

Type
list

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 ExpressionBC(*grid: GridBase, axis: int, upper: bool, *, rank: int = 0, normal: bool = False, value: Union[float, str] = 0, target: str = 'virtual_point'*)

Bases: `BCBase`

represents a boundary whose virtual point is calculated from an expression

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **value** (*float or str*) – An expression that determines the value of the boundary condition.
- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, and *virtual_point*.

copy(*upper: Optional[bool] = None*, *rank: int = None*) → *ExpressionBC*

return a copy of itself, but with a reference to the same grid

get_data(*idx: Tuple[int, ...]*) → Tuple[float, Dict[int, float]]

get_mathematical_representation(*field_name: str = 'C'*) → str

return mathematical representation of the boundary condition

get_virtual_point(*arr*, *idx: Optional[Tuple[int, ...]] = None*) → float

homogeneous: bool

determines whether the boundary condition depends on space

Type
bool

make_adjacent_evaluator() → Callable[[ndarray, int, Tuple[int, ...]], float]

make_virtual_point_evaluator() → Callable[[...], float]

returns a function evaluating the value at the virtual support point

Returns
A function that takes the data array and an index marking the current point, which is assumed to be a virtual point. The result is the data value at this point, which is calculated using the boundary condition.

Return type
function

names: List[str] = ['virtual_point']

identifiers used to specify the given boundary class

Type
list

set_ghost_cells(*data_full: ndarray*, ***, *args=None*) → None

set the ghost cell values for this boundary

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **args** – Additional arguments that might be supported by special boundary conditions.

class ExpressionDerivativeBC(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, str] = 0*, *target: str = 'derivative'*)

Bases: *ExpressionBC*

represents a boundary whose outward derivative is calculated from an expression

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **value** (*float or str*) – An expression that determines the value of the boundary condition.
- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, and *virtual_point*.

homogeneous: bool

determines whether the boundary condition depends on space

Type

bool

names: List[str] = ['derivative_expression', 'derivative_expr']

identifiers used to specify the given boundary class

Type

list

class ExpressionValueBC(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, str] = 0*, *target: str = 'value'*)

Bases: *ExpressionBC*

represents a boundary whose value is calculated from an expression

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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.
- **value** (*float or str*) – An expression that determines the value of the boundary condition.
- **target** (*str*) – Selects which value is actually set. Possible choices include *value*, *derivative*, and *virtual_point*.

homogeneous: bool

determines whether the boundary condition depends on space

Type

bool

names: List[str] = ['value_expression', 'value_expr']

identifiers used to specify the given boundary class

Type

list

class MixedBC(*grid: GridBase, axis: int, upper: bool, *, rank: int = 0, normal: bool = False, value: Union[float, ndarray, str] = 0, const: Union[float, ndarray, str] = 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
- **`normal`** (*`bool`*) – Flag indicating whether the condition is only applied in the normal direction.
- **`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: Optional[bool] = None*, *rank: int = None*, *value: Optional[Union[float, ndarray, str]] = None*, *const: Optional[Union[float, ndarray, str]] = None*) → *MixedBC*

return a copy of itself, but with a reference to the same grid

`get_mathematical_representation`(*field_name: str = 'C'*) → str

return mathematical representation of the boundary condition

`get_virtual_point_data`(*compiled: bool = False*) → Tuple[Any, Any, int]

return data suitable for calculating virtual points

Parameters

`compiled` (*`bool`*) – Flag indicating whether a compiled version is required, which automatically takes updated values into account when it is used in numba-compiled code.

Returns

the data structure associated with this virtual point

Return type

`BC1stOrderData`

`homogeneous: bool`

determines whether the boundary condition depends on space

Type

bool

`names: List[str] = ['mixed', 'robin']`

identifiers used to specify the given boundary class

Type

list

`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 NeumannBC(*grid: GridBase*, *axis: int*, *upper: bool*, ***, *rank: int = 0*, *normal: bool = False*, *value: Union[float, ndarray, str] = 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
- **normal** (`bool`) – Flag indicating whether the condition is only applied in the normal direction.
- **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: str = 'C'*) → str

return mathematical representation of the boundary condition

get_virtual_point_data(*compiled: bool = False*) → Tuple[Any, Any, int]

return data suitable for calculating virtual points

Parameters

compiled (`bool`) – Flag indicating whether a compiled version is required, which automatically takes updated values into account when it is used in numba-compiled code.

Returns

the data structure associated with this virtual point

Return type

`BC1stOrderData`

homogeneous: bool

determines whether the boundary condition depends on space

Type

bool

names: List[str] = ['derivative', 'neumann']

identifiers used to specify the given boundary class

Type

list

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 UserBC(*grid: GridBase, axis: int, upper: bool, *, rank: int = 0, normal: bool = False*)

Bases: *BCBase*

represents a boundary whose virtual point 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
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

copy(*upper: Optional[bool] = None, rank: int = None*) → UserBC

return a copy of itself, but with a reference to the same grid

get_mathematical_representation(*field_name: str = 'C'*) → str

return mathematical representation of the boundary condition

homogeneous: bool

determines whether the boundary condition depends on space

Type
bool

make_ghost_cell_setter() → Callable[[...], None]

return function that sets the ghost cells for this boundary

make_virtual_point_evaluator() → Callable[[...], float]

returns a function evaluating the value at the virtual support point

Returns
A function that takes the data array and an index marking the current point, which is assumed to be a virtual point. The result is the data value at this point, which is calculated using the boundary condition.

Return type
function

`names: List[str] = ['user']`

identifiers used to specify the given boundary class

Type
list

`set_ghost_cells`(*data_full: ndarray*, *, *args=None*) → None

set the ghost cell values for this boundary

Parameters

- **data_full** (ndarray) – The full field data including ghost points
- **values** (ndarray) – The values determining the values of the ghost cell. The interpretation of this values is determined by *self.target*.

`registered_boundary_condition_classes`() → Dict[str, Type[*BCBase*]]

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`() → Dict[str, Type[*BCBase*]]

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

4.2.2 pde.grids.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
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.grids.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_poisson_solver	make a operator that solves Poisson's equation

make_divergence(*grid: CartesianGrid*, *backend: str = 'auto'*) → Callable[[ndarray, ndarray], None]

make a divergence operator on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the divergence operator. If backend='auto', a suitable backend is chosen automatically.

Returns

A function that can be applied to an array of values

make_gradient(*grid: CartesianGrid*, *backend: str = 'auto'*) → Callable[[ndarray, ndarray], None]

make a gradient operator on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the gradient operator. If backend='auto', a suitable backend is chosen automatically.

Returns

A function that can be applied to an array of values

make_laplace(*grid: CartesianGrid*, *backend: str = 'auto'*) → Callable[[ndarray, ndarray], None]

make a laplace operator on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the laplace operator. If backend='auto', a suitable backend is chosen automatically.

Returns

A function that can be applied to an array of values

make_poisson_solver(*bcs: Boundaries*, *method: str = 'auto'*) → Callable[[ndarray, ndarray], None]

make a operator that solves Poisson's equation

Parameters

- **bcs** (*Boundaries*) – {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

make_tensor_divergence(*grid: CartesianGrid*, *backend: str = 'numba'*) → Callable[[ndarray, ndarray], None]

make a tensor divergence operator on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the tensor divergence operator.

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

make_vector_gradient(*grid: CartesianGrid*, *backend: str = 'numba'*) → Callable[[ndarray, ndarray], None]

make a vector gradient operator on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the vector gradient operator.

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

make_vector_laplace(*grid: CartesianGrid*, *backend: str = 'numba'*) → Callable[[ndarray, ndarray], None]

make a vector Laplacian on a Cartesian grid

Parameters

- **grid** (*CartesianGrid*) – The grid for which the operator is created
- **backend** (*str*) – Backend used for calculating the vector laplace operator.

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

pde.grids.operators.common module

Common functions that are used by many operators

make_general_poisson_solver(*matrix*, *vector*, *method: str = 'auto'*) → Callable[[ndarray, ndarray], None]

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

make_laplace_from_matrix(*matrix*, *vector*) → Callable[[ndarray, ndarray], None]

make a Laplace operator using matrix vector products

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.

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

uniform_discretization (*grid:* GridBase) → float

returns the uniform discretization or raises RuntimeError

pde.grids.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:* CylindricalSymGrid) → Callable[[ndarray, ndarray], 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

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

make_gradient (*grid:* CylindricalSymGrid) → Callable[[ndarray, ndarray], 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

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

make_gradient_squared (*grid:* CylindricalSymGrid, *central: bool = True*) → Callable[[ndarray, ndarray], None]

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
- **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

make_laplace(*grid: CylindricalSymGrid*) → Callable[[ndarray, ndarray], 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

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

make_tensor_divergence(*grid: CylindricalSymGrid*) → Callable[[ndarray, ndarray], 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

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

make_vector_gradient(*grid: CylindricalSymGrid*) → Callable[[ndarray, ndarray], 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

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

make_vector_laplace(*grid: CylindricalSymGrid*) → Callable[[ndarray, ndarray], 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

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

pde.grids.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_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:* PolarSymGrid) → Callable[[ndarray, ndarray], 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 $r_i = r_\mathrm{min} + (i + \frac12)\Delta r$ 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 $r_\mathrm{max} = r_\mathrm{min} + N_r\Delta r$.

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

`make_gradient`(*grid:* PolarSymGrid) → Callable[[ndarray, ndarray], None]

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 $r_i = r_\mathrm{min} + (i + \frac12)\Delta r$ 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 $r_\mathrm{max} = r_\mathrm{min} + N_r\Delta r$.

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

`make_gradient_squared`(*grid:* PolarSymGrid, *central: bool = True*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_\mathrm{min} + (i + \frac12)\Delta r$ 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 $r_\mathrm{max} = r_\mathrm{min} + N_r\Delta r$.

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

`make_laplace`(*grid:* PolarSymGrid) → Callable[[ndarray, ndarray], 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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

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

make_poisson_solver (*bcs: Boundaries*, *method: str = 'auto'*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

Parameters

- **bcs** (*Boundaries*) – Specifies the boundary conditions applied to the field. This must be an instance of *Boundaries*, 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

make_tensor_divergence (*grid: PolarSymGrid*) → Callable[[ndarray, ndarray], 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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

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

make_vector_gradient (*grid: PolarSymGrid*) → Callable[[ndarray, ndarray], 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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

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

pde.grids.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_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_divergence(*grid: SphericalSymGrid*, *safe: bool = True*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r \Delta r$.

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
- **safe** (*bool*) – Add extra checks for the validity of the input

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

make_gradient(*grid: SphericalSymGrid*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r \Delta r$.

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

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

make_gradient_squared(*grid: SphericalSymGrid*, *central: bool = True*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – 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

make_laplace (*grid: SphericalSymGrid*, *conservative: bool = True*) → Callable[[ndarray, ndarray], 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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The polar 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).

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

make_poisson_solver (*bcs: Boundaries*, *method: str = 'auto'*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

Parameters

- **bcs** (*Boundaries*) – Specifies the boundary conditions applied to the field. This must be an instance of *Boundaries*, 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

make_tensor_divergence (*grid: SphericalSymGrid*, *safe: bool = True*) → Callable[[ndarray, ndarray], None]

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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r\Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The polar grid for which this operator will be defined
- **safe** (*bool*) – Add extra checks for the validity of the input

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

make_vector_gradient (*grid: SphericalSymGrid*, *safe: bool = True*) → Callable[[ndarray, ndarray], 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 $r_i = r_{\mathrm{min}} + (i + \frac{1}{2})\Delta r$ 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 $r_{\mathrm{max}} = r_{\mathrm{min}} + N_r \Delta r$.

Parameters

- **grid** (*SphericalSymGrid*) – The polar grid for which this operator will be defined
- **safe** (*bool*) – Add extra checks for the validity of the input

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

4.2.3 pde.grids.base module

Bases classes

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: GridBase*) → None

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

axes: List[str]

property axes_bounds: Tuple[Tuple[float, float], ...]

lower and upper bounds of each axis

Type
tuple

property axes_coords: Tuple[ndarray, ...]

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

cell_coords

coordinate values for all axes of each cell

Type
ndarray

cell_to_point(*cells: ndarray*, *cartesian: bool = True*) → ndarray

convert cell coordinates to real coordinates

Parameters

- **cells** (ndarray) – Indices of the cells whose center coordinates are requested. This can be float values to indicate positions relative to the cell center.
- **cartesian** (*bool*) – Determines whether the point is returned in Cartesian coordinates or grid coordinates.

Returns
The center points of the respective cells

Return type
ndarray

Warning: This method is deprecated since 2022-03-14 and will be removed soon.

cell_volume_data: Sequence[Union[float, ndarray]]

cell_volumes

volume of each cell

Type
ndarray

compatible_with(*other: GridBase*) → bool

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: ndarray*, *, *coords: str = 'cartesian'*, *wrap: bool = True*) → ndarray

check whether the point is contained in the grid

Parameters

- **point** (ndarray) – Coordinates of the point

- **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] = []

copy() → *GridBase*

return a copy of the grid

difference_vector_real(*p1: ndarray*, *p2: ndarray*) → ndarray

return the vector 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)

Returns
The difference vectors between the points with periodic boundary conditions applied.

Return type
ndarray

dim: int

property discretization: ndarray

the linear size of a cell along each axis

Type
numpy.array

distance_real(*p1: ndarray*, *p2: ndarray*) → float

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

Returns
Distance between the two positions

Return type
float

classmethod from_state(*state: Union[str, Dict[str, Any]]*) → *GridBase*

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*.

get_axis_index(*key: Union[int, str]*, *allow_symmetric: bool = True*) → int

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

abstract get_boundary_conditions(*bc: BoundariesData = 'auto_periodic_neumann'*, *rank: int = 0*, *normal: bool = False*) → *Boundaries*

abstract get_image_data(*data: ndarray*) → Dict[str, Any]

abstract get_line_data(*data: ndarray*, *extract: str = 'auto'*) → Dict[str, Any]

abstract get_random_point(***, *boundary_distance: float = 0*, *coords: str = 'cartesian'*) → ndarray

get_subgrid(*indices: Sequence[int]*) → *GridBase*

return a subgrid of only the specified axes

integrate(*data: Union[int, float, complex, ndarray]*, *axes: Optional[Union[int, Sequence[int]]] = None*) → ndarray

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`

abstract iter_mirror_points(*point: ndarray*, *with_self: bool = False*, *only_periodic: bool = True*) → Generator

make_cell_volume_compiled(*flat_index: bool = False*) → Callable[[...], float]

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(*, *full_data: bool = False*) → Callable[[ndarray, ndarray, Union[int, float, complex, ndarray]], None]

return a compiled function to insert values at interpolated positions

Parameters
full_data (*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.

make_integrator() → Callable[[ndarray], ndarray]

Return function that can be used to integrates discretized data over the grid

Note that currently only scalar fields are supported.

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: bool = True*) → Callable[[ndarray], None]

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: Union[str, OperatorInfo], bc: BoundariesData*, *, *normal_bcs: bool = None*, ***kwargs*) → Callable[..., np.ndarray]

return a compiled function applying an operator with boundary conditions

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*.

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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **normal_bcs** (*bool*) – Flag indicating whether the condition is only applied in the normal direction. This value normally determined automatically from the properties of the operator. Here, normal=True is assumed if the operator reduced the rank of the field, so that the output has lower rank than the input.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

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

Return type
callable

make_operator_no_bc(*operator: Union[str, OperatorInfo], **kwargs*) → Callable[[ndarray, ndarray], None]

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.
- ****kwargs** – Specifies extra arguments influencing how the operator is created.

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

Return type
callable

normalize_point(*point: ndarray*, ***, *reflect: bool = False*) → ndarray

normalize grid coordinates by applying periodic boundary conditions

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

- **point** (`ndarray`) – Coordinates of a single point
- **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

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

periodic: List[bool]

plot()

visualize the grid

abstract point_from_cartesian(*points: ndarray*) → ndarray

abstract point_to_cartesian(*points: ndarray*, ***, *full: bool = False*) → ndarray

point_to_cell(*points: ndarray*) → ndarray

Determine cell(s) corresponding to given point(s)

Parameters

points (`ndarray`) – Real coordinates

Returns

The indices of the respective cells

Return type

`ndarray`

Warning: This method is deprecated since 2022-03-14 and will be removed soon.

abstract polar_coordinates_real(*origin: ndarray*, ***, *ret_angle: bool = False*) → Union[ndarray, Tuple[ndarray, ...]]

classmethod register_operator(*name: str*, *factory_func: Optional[Callable] = None*, *rank_in: int = 0*, *rank_out: int = 0*)

register an operator for this grid

Example

The method can either be used directly:

```
GridClass.register_operator("operator", make_operator)
```

or as a decorator for the factory function:

```
@GridClass.register_operator("operator")
def make_operator(bcs: Boundaries):
    ...
```

Parameters

- **name** (*str*) – The name of the operator to register
- **factory_func** (*callable*) – A function with signature `(bcs: Boundaries, **kwargs)`, which takes boundary conditions 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

property shape: Tuple[int, ...]

the number of support points of each axis

Type
tuple of int

abstract property state: Dict[str, Any]

property state_serialized: str

JSON-serialized version of the state of this grid

Type
str

transform(*coordinates: ndarray*, *source: str*, *target: str*) → ndarray

converts coordinates from one coordinate system to another

Supported coordinate systems include

- *cartesian*: Cartesian coordinates where each point carries *dim* values
- *cell*: Grid coordinates based on indexing the discretization cells
- *grid*: Grid coordinates where each point carries *num_axes* values

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

abstract property volume: float

class OperatorInfo(*factory: Callable[..., OperatorType]*, *rank_in: int*, *rank_out: int*, *name: str = ''*)

Bases: `tuple`

stores information about an operator

Create new instance of OperatorInfo(factory, rank_in, rank_out, name)

property factory

Alias for field number 0

property name

Alias for field number 3

property rank_in

Alias for field number 1

property rank_out

Alias for field number 2

exception PeriodicityError

Bases: `RuntimeError`

exception indicating that the grid periodicity is inconsistent

`discretize_interval`(*x_min: float*, *x_max: float*, *num: int*) → Tuple[ndarray, float]

construct a list of equidistantly placed intervals

The discretization is defined as

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

where N is the number of intervals given by *num*.

Parameters

- **`x_min`** (`float`) – Minimal value of the axis
- **`x_max`** (`float`) – Maximal value of the axis
- **`num`** (`int`) – Number of intervals

Returns

(midpoints, dx): the midpoints of the intervals and the used discretization *dx*.

Return type

tuple

`registered_operators`() → Dict[str, List[str]]

returns all operators that are currently defined

Returns

a dictionary with the names of the operators defined for each grid class

Return type

dict

4.2.4 pde.grids.cartesian module

Cartesian grids of arbitrary dimension.

class `CartesianGrid`(*bounds: Sequence[Tuple[float, float]], shape: Union[int, Sequence[int]], periodic: Union[Sequence[bool], bool] = 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

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

where $N^{(k)}$ is the number of cells along this dimension. Consequently, cells have dimension $\Delta x^{(k)}$ and cover the interval $[x^{(k)}_{\text{min}}, x^{(k)}_{\text{max}}]$.

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*

from_polar_coordinates(*distance: ndarray, angle: ndarray, origin: Optional[ndarray] = None*) → ndarray

convert polar coordinates to Cartesian coordinates

This function is currently only implemented for 1d and 2d systems.

Parameters

- **distance** (ndarray) – The radial distance
- **angle** (ndarray) – The angle with respect to the origin
- **origin** (ndarray, optional) – Sets the origin of the coordinate system. If omitted, the zero point is assumed as the origin.

Returns

The Cartesian coordinates corresponding to the given polar coordinates.

Return type

ndarray

classmethod from_state(*state: Dict[str, Any]*) → *CartesianGrid*

create a field from a stored *state*.

Parameters

state (*dict*) – The state from which the grid is reconstructed.

get_boundary_conditions(*bc: BoundariesData = 'auto_periodic_neumann', rank: int = 0, normal: bool = False*) → *Boundaries*

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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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.
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

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: ndarray*) → Dict[str, Any]

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.

get_line_data(*data: ndarray*, *extract: str = 'auto'*) → Dict[str, Any]

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.

get_random_point(**, boundary_distance: float = 0, coords: str = 'cartesian', rng: Optional[Generator] = None, cartesian: Optional[bool] = None*) → ndarray

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_subgrid(*indices: Sequence[int]*) → *CartesianGrid*

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

iter_mirror_points (*point: ndarray*, *with_self: bool = False*, *only_periodic: bool = True*) → Generator

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

plot (**args*, *title: str = None*, *filename: str = None*, *action: str = 'auto'*, *ax_style: Optional[Dict[str, Any]] = None*, *fig_style: Optional[Dict[str, Any]] = 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 *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.
- **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 in 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

point_from_cartesian (*coords: ndarray*) → ndarray

convert points given in Cartesian coordinates to this grid

Parameters
coords (ndarray) – Points in Cartesian coordinates.

Returns
Points given in the coordinates of the grid

Return type
`ndarray`

point_to_cartesian(*points: ndarray*, ***, *full: bool = False*) → ndarray

convert coordinates of a point to Cartesian coordinates

Parameters

- **points** (`ndarray`) – Points given in grid coordinates
- **full** (*bool*) – Compatibility option not used in this method

Returns
The Cartesian coordinates of the point

Return type
`ndarray`

polar_coordinates_real(*origin: ndarray*, ***, *ret_angle: bool = False*) → Union[ndarray, Tuple[ndarray, ndarray, ndarray]]

return polar coordinates associated with the grid

Parameters

- **origin** (`ndarray`) – Coordinates of the origin at which the polar coordinate system is anchored.
- **ret_angle** (*bool*) – Determines whether angles are returned alongside the distance. If *False* only the distance to the origin is returned for each support point of the grid. If *True*, the distance and angles are returned. For a 1d system system, the angle is defined as the sign of the difference between the point and the origin, so that angles can either be 1 or -1. For 2d systems and 3d systems, polar coordinates and spherical coordinates are used, respectively.

property state: Dict[str, Any]

the state of the grid

Type
dict

property volume: float

total volume of the grid

Type
float

CartesianGridBase

alias of *CartesianGrid*

class UnitGrid(*shape: Sequence[int]*, *periodic: Union[Sequence[bool], bool] = 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.

axes: List[str]

cuboid: *Cuboid*

dim: int

classmethod from_state(*state: Dict[str, Any]*) → *UnitGrid*

create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

get_subgrid(*indices: Sequence[int]*) → *UnitGrid*

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

num_axes: int

periodic: List[bool]

property state: Dict[str, Any]

the state of the grid

Type
dict

to_cartesian() → *CartesianGrid*

convert unit grid to *CartesianGrid*

4.2.5 pde.grids.cylindrical module

Cylindrical grids with azimuthal symmetry

class CylindricalSymGrid(*radius: float, bounds_z: Tuple[float, float], shape: Tuple[int, int], periodic_z: bool = 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 as

$$r_i = \left(i + \frac{1}{2}\right)\Delta r \qquad \text{for} \quad i = 0, \ldots, N_r - 1 \qquad \text{with} \quad \Delta r = \frac{R}{N_r}$$

$$z_j = z_{\text{min}} + \left(j + \frac{1}{2}\right)\Delta z \qquad \text{for} \quad j = 0, \ldots, N_z - 1 \qquad \text{with} \quad \Delta z = \frac{z_{\text{max}} - z_{\text{min}}}{N_z}$$

where R is the radius of the cylindrical grid, z_{min} and z_{max} denote the respective lower and upper bounds of the axial direction, so that $z_{\mathrm{max}} - z_{\mathrm{min}}$ 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 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, e.g., use `field['z']` instead of `field[1]`.

Parameters

- **radius** (*float*) – The radius of the cylinder
- **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.

axes: List[str] = ['r', 'z']

axes_symmetric: List[str] = ['phi']

The names of the additional axes that the fields do not depend on, e.g. along which they are constant.

Type
list

cell_volume_data: Sequence[FloatNumerical]

the volumes of all cells

Type
`ndarray`

coordinate_constraints: List[int] = [0, 1]

dim: int = 3

classmethod from_state(*state: Dict[str, Any]*) → *CylindricalSymGrid*

create a field from a stored *state*.

Parameters
state (*dict*) – The state from which the grid is reconstructed.

get_boundary_conditions(*bc: BoundariesData = 'auto_periodic_neumann', rank: int = 0, normal: bool = False*) → *Boundaries*

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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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.
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

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_cartesian_grid(*mode: str = 'valid'*) → *CartesianGrid*

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: ndarray*) → Dict[str, Any]

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.

get_line_data(*data: ndarray*, *extract: str = 'auto'*) → Dict[str, Any]

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.

get_random_point(**, boundary_distance: float = 0, avoid_center: bool = False, coords: str = 'cartesian', rng: Optional[Generator] = None, cartesian: Optional[bool] = None*) → ndarray

return a random point within the grid

Note that these points will be uniformly distributed on the radial axis, which implies that they are not uniformly distributed in the volume.

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`

get_subgrid(*indices: Sequence[int]*) → Union[*CartesianGrid*, *PolarSymGrid*]

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

iter_mirror_points(*point: ndarray*, *with_self: bool = False*, *only_periodic: bool = True*) → Generator

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

property length: float

length of the cylinder

Type
float

num_axes: int = 2

periodic: List[bool]

point_from_cartesian(*points: ndarray*) → ndarray

convert points given in Cartesian coordinates to this grid

This function returns points restricted to the x-z plane, i.e., the y-coordinate will be zero.

Parameters
points (`ndarray`) – Points given in Cartesian coordinates.

Returns
Points given in the coordinates of the grid

Return type
`ndarray`

point_to_cartesian(*points: ndarray*, *, *full: bool = False*) → ndarray

convert coordinates of a point to Cartesian coordinates

Parameters

- **points** (ndarray) – The grid coordinates of the points
- **full** (*bool*) – Flag indicating whether angular coordinates are specified

Returns

The Cartesian coordinates of the point

Return type

ndarray

polar_coordinates_real(*origin: ndarray*, *, *ret_angle: bool = False*) → Union[ndarray, Tuple[ndarray, ndarray]]

return spherical coordinates associated with the grid

Parameters

- **origin** (ndarray) – Coordinates of the origin at which the polar coordinate system is anchored. Note that this must be of the form *[0, 0, z_val]*, where only *z_val* can be chosen freely.
- **ret_angle** (*bool*) – Determines whether the azimuthal angle is returned alongside the distance. If *False* only the distance to the origin is returned for each support point of the grid. If *True*, the distance and angles are returned.

property radius: float

radius of the cylinder

Type

float

property state: Dict[str, Any]

the state of the grid

Type

state

property volume: float

total volume of the grid

Type

float

4.2.6 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.

class PolarSymGrid(*radius: Union[float, Tuple[float, float]], shape: Union[Tuple[int], int]*)

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

$$r_i = R_\mathrm{inner} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N - 1 \quad \text{with} \quad \Delta r = \frac{R_\mathrm{outer} - R_\mathrm{inner}}{N}$$

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, $(R_\mathrm{inner}, R_\mathrm{outer})$.
- **`shape`** (*`tuple or int`*) – A single number setting the number N of support points along the radial coordinate

`axes: List[str] = ['r']`

`axes_symmetric: List[str] = ['phi']`

The names of the additional axes that the fields do not depend on, e.g. along which they are constant.

Type

list

`cell_volume_data: Sequence[FloatNumerical]`

`coordinate_constraints: List[int] = [0, 1]`

`dim: int = 2`

`point_to_cartesian`(*points: ndarray*, ***, *full: bool = False*) → ndarray

convert coordinates of a point to Cartesian coordinates

This function returns points along the y-coordinate, i.e, the x coordinates will be zero.

Parameters

- **`points`** (`ndarray`) – The grid coordinates of the points
- **`full`** (*`bool`*) – Flag indicating whether angular coordinates are specified

Returns

The Cartesian coordinates of the point

Return type

`ndarray`

`class SphericalSymGrid`(*radius: Union[float, Tuple[float, float]], shape: Union[Tuple[int], int]*)

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:

$$r_i = R_\mathrm{inner} + \left(i + \frac{1}{2}\right)\Delta r \quad \text{for} \quad i = 0, \ldots, N - 1 \quad \text{with} \quad \Delta r = \frac{R_\mathrm{outer} - R_\mathrm{inner}}{N}$$

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*) – A single number setting the number N of support points along the radial coordinate

axes: List[str] = ['r']

axes_symmetric: List[str] = ['theta', 'phi']

The names of the additional axes that the fields do not depend on, e.g. along which they are constant.

Type
list

cell_volume_data: Sequence[FloatNumerical]

coordinate_constraints: List[int] = [0, 1, 2]

dim: int = 3

point_to_cartesian(*points: ndarray*, ***, *full: bool = False*) → ndarray

convert coordinates of a point to Cartesian coordinates

This function returns points along the z-coordinate, i.e, the x and y coordinates will be zero.

Parameters

- **points** (ndarray) – The grid coordinates of the points
- **full** (*bool*) – Flag indicating whether angular coordinates are specified

Returns
The Cartesian coordinates of the point

Return type
ndarray

class SphericalSymGridBase(*radius: Union[float, Tuple[float, float]]*, *shape: Union[Tuple[int], int]*)

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*) – A single number setting the number N of support points along the radial coordinate

axes: List[str]

cell_volume_data: Sequence[FloatNumerical]

the volumes of all cells

Type

tuple of ndarray

dim: int

classmethod from_state(*state: Dict[str, Any]*) → *SphericalSymGridBase*

create a field from a stored *state*.

Parameters

state (*dict*) – The state from which the grid is reconstructed.

get_boundary_conditions(*bc='auto_periodic_neumann', rank: int = 0, normal: bool = False*) → *Boundaries*

constructs boundary conditions from a flexible data format.

If the inner boundary condition for a grid without a hole is not specified, this condition is automatically set to a vanishing derivative at $r = 0$.

Parameters

- **bc** (*str or list or tuple or dict*) – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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.
- **normal** (*bool*) – Flag indicating whether the condition is only applied in the normal direction.

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_cartesian_grid(*mode: str = 'valid', num: int = None*) → *CartesianGrid*

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: ndarray, performance_goal: str = 'speed', fill_value: float = 0, masked: bool = True*) → Dict[str, Any]

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
A dictionary with information about the image, which is convenient for plotting.

get_line_data(*data: ndarray, extract: str = 'auto'*) → Dict[str, Any]

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.

get_random_point(**, boundary_distance: float = 0, avoid_center: bool = False, coords: str = 'cartesian', rng: Optional[Generator] = None, cartesian: Optional[bool] = None*) → ndarray

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

iter_mirror_points(*point: ndarray*, *with_self: bool = False*, *only_periodic: bool = True*) → Generator

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

num_axes: int = 1

periodic: List[bool] = [False]

plot(**args*, *title: str = None*, *filename: str = None*, *action: str = 'auto'*, *ax_style: Optional[Dict[str, Any]] = None*, *fig_style: Optional[Dict[str, Any]] = 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 *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.
- **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 in 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

point_from_cartesian(*points: ndarray*) → ndarray

convert points given in Cartesian coordinates to this grid

Parameters

points (`ndarray`) – Points given in Cartesian coordinates.

Returns

Points given in the coordinates of the grid

Return type
`ndarray`

polar_coordinates_real(*origin=None*, ***, *ret_angle: bool = False*, ***kwargs*) → Union[ndarray, Tuple[ndarray, ...]]

return spherical coordinates associated with the grid

Parameters

- **origin** – Place holder variable to comply with the interface
- **ret_angle** (*bool*) – Determines whether angles are returned alongside the distance. If *False* only the distance to the origin is returned for each support point of the grid. If *True*, the distance and angles are returned. Note that in the case of spherical grids, this angle is zero by convention.

property radius: Union[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: TNumArr*, *dim: int*) → TNumArr

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`

4.3 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.

PDE	PDE defined by mathematical expressions
AllenCahnPDE	A simple Allen-Cahn equation
CahnHilliardPDE	A simple Cahn-Hilliard equation
DiffusionPDE	A simple diffusion equation
KPZInterfacePDE	The Kardar–Parisi–Zhang (KPZ) equation
KuramotoSivashinskyPDE	The Kuramoto-Sivashinsky 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 Laplace's equation on a given grid

4.3.1 pde.pdes.allen_cahn module

A Allen-Cahn equation

class AllenCahnPDE(*interface_width: float = 1, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann'*)

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 interfacial width.

Parameters

- **interface_width** (*float*) – The diffusivity of the described species
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

evolution_rate(*state: ScalarField, t: float = 0*) → *ScalarField*

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: Optional[bool] = 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`

4.3.2 pde.pdes.base module

Base classes

`class PDEBase`(**, noise: Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]]] = 0, rng: Optional[Generator] = None*)

Bases: `object`

base class for solving partial differential equations

Custom PDEs can be implemented by specifying their evolution rate. In the simple case of deterministic PDEs, the methods *`PDEBase.evolution_rate()`* and `PDEBase._make_pde_rhs_numba()` need to be overwritten for the *numpy* and *numba* backend, respectively.

Parameters

- **`noise`** (float or `ndarray`) – Magnitude 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.
- **`rng`** (`Generator`) – Random number generator (default: `default_rng()`). Note that this random number generator is only used for numpy function, while compiled numba code is unaffected.

Note: If more complicated noise structures are required, the methods *`PDEBase.noise_realization()`* and `PDEBase._make_noise_realization_numba()` need to be overwritten for the *numpy* and *numba* backend, respectively.

`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 this might lead to wrong results if the parameters of the PDE were changed after the first call. This option is thus disabled by default and should be used with care.

Type
bool

`check_implementation: bool = True`
Flag determining whether (some) numba-compiled functions should be checked against their numpy counterparts. This can help with implementing a correct compiled version for a PDE class.

Type
bool

check_rhs_consistency(*state:* FieldBase, *t: float = 0*, ***, *tol: float = 1e-07*, *rhs_numba: Optional[Callable] = None*)

check the numba compiled right hand side versus the numpy variant

Parameters

- **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
- **rhs_numba** (*callable*) – The implementation of the numba variant that is to be checked. If omitted, an implementation is obtained by calling `PDEBase._make_pde_rhs_numba_cached()`.

complex_valued: bool = False

Flag indicating whether the right hand side is a complex-valued PDE, which requires all involved variables to be of complex type

Type
bool

abstract evolution_rate(*state:* FieldBase, *t: float = 0*) → *FieldBase*

explicit_time_dependence: Optional[bool] = 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

The `BasePDF` 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*.

make_modify_after_step(*state:* FieldBase) → Callable[[ndarray], float]

returns a function that can be called to modify a state

This function is applied to the state after each integration step when an explicit stepper is used. The default behavior is to not change the state.

Parameters
state (`FieldBase`) – An example for the state from which the grid and other information can be extracted

Returns
Function that can be applied to a state to modify it and which returns a measure for the corrections applied to the state

make_pde_rhs(*state:* FieldBase, *backend: str = 'auto'*) → Callable[[ndarray, float], ndarray]

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*) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.

Returns
Function determining the right hand side of the PDE

Return type
callable

make_sde_rhs (*state: FieldBase*, *backend: str = 'auto'*) → Callable[[ndarray, float], Tuple[ndarray, ndarray]]

return a function for evaluating the right hand side of the SDE

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. Accepted values are 'python` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.

Returns
Function determining the deterministic part of the right hand side of the PDE together with a noise realization.

noise_realization (*state: FieldBase*, *t: float = 0*, *label: str = 'Noise realization'*) → *FieldBase*

returns a realization for the noise

Parameters

- **state** (ScalarField) – The scalar field describing the concentration distribution
- **t** (*float*) – The current time point
- **label** (*str*) – The label for the returned field

Returns
Scalar field describing the evolution rate of the PDE

Return type
`ScalarField`

solve (*state: FieldBase*, *t_range: TRangeType*, *dt: float = None*, *tracker: Optional[Union[Sequence[Union[TrackerBase, str]], TrackerBase, str]] = 'auto'*, *method: Union[str, SolverBase] = 'auto'*, *ret_info: bool = False*, ***kwargs*) → Union[*FieldBase*, Tuple[*FieldBase*, Dict[str, Any]]]

convenience method for solving 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*. To obtain full flexibility, it is advisable to construct these classes explicitly.

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 assumed to be *(0, t_end)*.
- **dt** (*float*) – Time step of the chosen stepping scheme. If *None*, a default value based on the stepper will be chosen. In particular, if *method == 'auto'*, *ScipySolver* with an automatic, adaptive time step provided by scipy is used. This is a flexible choice, but

can also result in unstable or slow simulations. If an adaptive stepper is used (supported by `ScipySolver` and `ExplicitSolver`), the value given here sets the initial time step.

- **tracker** – Defines a tracker that process the state of the simulation at specified time intervals. A tracker is either an instance of `TrackerBase` or a string, which identifies a tracker. All possible identifiers can be obtained by calling `get_named_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 interval at which the tracker is evaluated can be chosen when creating a tracker object explicitly.
- **method** (`SolverBase` or str) – Specifies the method for solving the differential equation. This can either be an instance of `SolverBase` or a descriptive name like 'explicit' or 'scipy'. The valid names are given by `pde.solvers.base.SolverBase.registered_solvers()`. The default value 'auto' selects `ScipySolver` if *dt* is not specified and `ExplicitSolver` otherwise. Details of the solvers and additional features (like adaptive time steps) are explained in their documentation.
- **ret_info** (`bool`) – Flag determining whether diagnostic information about the solver process should be returned.
- ****kwargs** – Additional keyword arguments are forwarded to the solver class chosen with the *method* argument. In particular, `ExplicitSolver` supports several *schemes* and an adaptive stepper can be enabled using `adaptive=True`. Conversely, `ScipySolver` accepts the additional arguments of `scipy.integrate.solve_ivp()`.

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.

Return type
`FieldBase`

expr_prod(*factor: float*, *expression: str*) → str

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

4.3.3 pde.pdes.cahn_hilliard module

A Cahn-Hilliard equation

class CahnHilliardPDE(*interface_width: float = 1, bc_c: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann', bc_mu: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann'*)

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 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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_mu** – The boundary conditions applied to the chemical potential associated with the scalar field c. Supports the same options as *bc_c*.

evolution_rate(*state: ScalarField, t: float = 0*) → *ScalarField*

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: Optional[bool] = 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

4.3.4 pde.pdes.diffusion module

A simple diffusion equation

class DiffusionPDE(*diffusivity: float = 1, noise: float = 0, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann'*)

Bases: `PDEBase`

A simple diffusion equation

The mathematical definition is

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

where c is a scalar field that is distributed with diffusivity D.

Parameters

- **diffusivity** (`float`) – The diffusivity of the described species
- **noise** (`float`) – Strength of the (additive) noise term
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

evolution_rate(*state: ScalarField, t: float = 0*) → *ScalarField*

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: Optional[bool] = 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

4.3.5 pde.pdes.kpz_interface module

The Kardar–Parisi–Zhang (KPZ) equation describing the evolution of an interface

class KPZInterfacePDE(*nu: float = 0.5, lmbda: float = 1, *, noise: float = 0, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann'*)

Bases: *PDEBase*

The Kardar–Parisi–Zhang (KPZ) equation

The mathematical definition is

$$\partial_t h = \nu \nabla^2 h + \frac{\lambda}{2} \left(\nabla h\right)^2 + \eta(\boldsymbol{r}, t)$$

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 strenth of the gradient term
- **noise** (*float*) – Strength of the (additive) noise term
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

evolution_rate(*state: ScalarField, t: float = 0*) → *ScalarField*

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: Optional[bool] = 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

4.3.6 pde.pdes.kuramoto_sivashinsky module

The Kardar–Parisi–Zhang (KPZ) equation describing the evolution of an interface

class KuramotoSivashinskyPDE(*nu: float = 1, *, noise: float = 0, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann', bc_lap: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = None*)

Bases: *PDEBase*

The Kuramoto-Sivashinsky equation

The mathematical definition is

$$\partial_t u = -\nu \nabla^4 u - \nabla^2 u - \frac{1}{2} \left(\nabla h\right)^2 + \eta(\boldsymbol{r}, t)$$

where u is the height of the interface in Monge parameterization. The dynamics are governed by the parameters 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
- **noise** (*float*) – Strength of the (additive) noise term
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_lap** – 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*.

evolution_rate(*state:* ScalarField, *t: float = 0*) → *ScalarField*

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: Optional[bool] = 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

4.3.7 pde.pdes.laplace module

Solvers for Poisson's and Laplace's equation

solve_laplace_equation(*grid:* GridBase, *bc: Union[Dict[str, Union[Dict, str,* BCBase*]], Dict, str,* BCBase*, Tuple[Union[Dict, str,* BCBase*], Union[Dict, str,* BCBase*]], Sequence[Union[Dict[str, Union[Dict, str,* BCBase*]], Dict, str,* BCBase*, Tuple[Union[Dict, str,* BCBase*], Union[Dict, str,* BCBase*]]]]], label: str = "Solution to Laplace's equation"*) → *ScalarField*

Solve Laplace's equation on a given grid.

This is implemented by calling `solve_poisson_equation()` with a vanishing right hand side.

Parameters

- **grid** (*GridBase*) – The grid on which the equation is solved
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **label** (*str*) – The label of the returned field.

Returns

The field that solves the equation. This field will be defined on the given *grid*.

Return type

ScalarField

solve_poisson_equation(*rhs: ScalarField, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]], label: str = "Solution to Poisson's equation", **kwargs*) → *ScalarField*

Solve Laplace'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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **label** (*str*) – The label of the returned field.

Returns
The field u that solves the equation. This field will be defined on the same grid as *rhs*.

Return type
ScalarField

4.3.8 pde.pdes.pde module

Defines a PDE class whose right hand side is given as a string

class PDE(*rhs: Dict[str, str], *, noise: Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]]] = 0, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann', bc_ops: Optional[Dict[str, Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]]]] = None, user_funcs: Optional[Dict[str, Callable]] = None, consts: Optional[Dict[str, Union[int, float, complex, ndarray]]] = None*)

Bases: `PDEBase`

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

`diagnostics`

Additional diagnostic information that might help with analyzing problems, e.g., when `sympy` cannot parse or :mod`numba` cannot compile a function.

Type
dict

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*.
- **`noise`** (float or `ndarray`) – Magnitude 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.
- **`bc`** – 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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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 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*.

- **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.

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.

evolution_rate (*state: FieldBase, t: float = 0.0*) → *FieldBase*

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

4.3.9 pde.pdes.swift_hohenberg module

The Swift-Hohenberg equation

class SwiftHohenbergPDE (*rate: float = 0.1, kc2: float = 1.0, delta: float = 1.0, *, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann', bc_lap: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 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** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.
- **bc_lap** – 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*.

evolution_rate (*state:* ScalarField, *t: float = 0*) → *ScalarField*

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: Optional[bool] = 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

4.3.10 pde.pdes.wave module

A simple diffusion equation

class WavePDE (*speed: float = 1, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann'*)

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 u is the density field that and c sets the wave speed.

Parameters

- **speed** (*float*) – The speed c of the wave
- **bc** – The boundary conditions applied to the field. Boundary conditions are generally given as a list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. More information can be found in the *boundaries documentation*.

evolution_rate(*state: FieldCollection*, *t: float = 0*) → *FieldCollection*

evaluate the right hand side of the PDE

Parameters

- **state** (`FieldCollection`) – The fields u and v distribution
- **t** (*float*) – The current time point

Returns
Scalar field describing the evolution rate of the PDE

Return type
`FieldCollection`

explicit_time_dependence: Optional[bool] = 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: ScalarField*, *v: Optional[ScalarField] = 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`

4.4 pde.solvers package

Solvers define how a PDE is solved, i.e., how the initial state is advanced in time.

Controller	class controlling a simulation
ExplicitSolver	class for solving partial differential equations explicitly
ImplicitSolver	class for solving partial differential equations implicitly
ScipySolver	class for solving partial differential equations using scipy
registered_solvers	returns all solvers that are currently registered

class Controller(*solver: SolverBase, t_range: Union[float, Tuple[float, float]], tracker: Optional[Union[Sequence[Union[TrackerBase, str]], TrackerBase, str]] = 'auto'*)

Bases: `object`

class controlling a simulation

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** – Defines a tracker that process the state of the simulation at specified time intervals. A tracker is either an instance of *TrackerBase* or a string, which identifies a tracker. All possible identifiers can be obtained by calling *get_named_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 interval at which the tracker is evaluated can be chosen when creating a tracker object explicitly.

get_current_time()

process_time() -> float

Process time for profiling: sum of the kernel and user-space CPU time.

run(*state: TState, dt: float = None*) → TState

run the simulation

Diagnostic information about the solver procedure are available in the *diagnostics* property of the instance after this function has been called.

Parameters

- **state** – 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*) – Time step of the chosen stepping scheme. If *None*, a default value based on the stepper will be chosen.

Returns

The state at the final time point.

property t_range: Tuple[float, float]

start and end time of the simulation

Type
tuple

class ExplicitSolver(*pde: PDEBase, scheme: str = 'euler', backend: str = 'auto', adaptive: bool = False, tolerance: float = 1e-05*)

Bases: `SolverBase`

class for solving partial differential equations explicitly

Parameters

- **pde** (`PDEBase`) – The instance describing the pde that needs to be solved
- **scheme** (`str`) – Defines the explicit scheme to use. Supported values are 'euler' and 'runge-kutta' (or 'rk' for short).
- **backend** (`str`) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.
- **adaptive** (`bool`) – When enabled, the time step is adjusted during the simulation using the error tolerance set with *tolerance*.
- **tolerance** (`float`) – The error tolerance used in adaptive time stepping. This is used in adaptive time stepping to choose a time step which is small enough so the truncation error of a single step is below *tolerance*.

dt_max = 10000000000.0

dt_min = 1e-10

make_stepper(*state: FieldBase, dt=None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function using an explicit scheme

Parameters

- **state** (`FieldBase`) – An example for the state from which the grid and other information can be extracted
- **dt** (`float`) – Time step of the explicit stepping. 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)*

name = 'explicit'

class ImplicitSolver(*pde: PDEBase, maxiter: int = 100, maxerror: float = 0.0001, backend: str = 'auto'*)

Bases: `SolverBase`

class for solving partial differential equations implicitly

Parameters

- **pde** (`PDEBase`) – The instance describing the pde that needs to be solved
- **maxiter** (`int`) – The maximal number of iterations per step
- **maxerror** (`float`) – The maximal error that is permitted in each step
- **backend** (`str`) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.

make_stepper(*state:* FieldBase, *dt=None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function using an implicit scheme

Parameters

- **state** (`FieldBase`) – An example for the state from which the grid and other information can be extracted
- **dt** (*`float`*) – Time step of the explicit stepping. 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)*

name = 'implicit'

class ScipySolver(*pde:* PDEBase, *backend: str = 'auto'*, ***kwargs*)

Bases: *`SolverBase`*

class for solving partial differential equations using scipy

This class is a thin wrapper around `scipy.integrate.solve_ivp()`. In particular, it supports all the methods implemented by this function.

Parameters

- **pde** (*`PDEBase`*) – The instance describing the pde that needs to be solved
- **backend** (*`str`*) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.
- ****kwargs** – All extra arguments are forwarded to `scipy.integrate.solve_ivp()`.

make_stepper(*state:* FieldBase, *dt: float = None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function

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*.

name = 'scipy'

registered_solvers() → List[str]

returns all solvers that are currently registered

Returns

List with the names of the solvers

Return type

list of str

4.4.1 pde.solvers.base module

Package that contains base classes for solvers

class SolverBase(*pde: PDEBase*)

Bases: `object`

base class for solvers

Parameters

pde (*PDEBase*) – The partial differential equation that should be solved

classmethod from_name(*name: str, pde: PDEBase, **kwargs*) → *SolverBase*

create solver class based on its 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`

abstract make_stepper(*state, dt: float = None*) → Callable[[*FieldBase*, float, float], float]

registered_solvers = ['ExplicitSolver', 'ImplicitSolver', 'ScipySolver', 'explicit', 'implicit', 'scipy']

4.4.2 pde.solvers.controller module

Defines the `Controller` class for solving pdes.

class Controller(*solver: SolverBase, t_range: Union[float, Tuple[float, float]], tracker: Optional[Union[Sequence[Union[TrackerBase, str]], TrackerBase, str]] = 'auto'*)

Bases: `object`

class controlling a simulation

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** – Defines a tracker that process the state of the simulation at specified time intervals. A tracker is either an instance of `TrackerBase` or a string, which identifies a tracker. All possible identifiers can be obtained by calling `get_named_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 interval at which the tracker is evaluated can be chosen when creating a tracker object explicitly.

`get_current_time`()

process_time() -> float

Process time for profiling: sum of the kernel and user-space CPU time.

`run`(*state: TState*, *dt: float = None*) → TState

run the simulation

Diagnostic information about the solver procedure are available in the *diagnostics* property of the instance after this function has been called.

Parameters

- **`state`** – 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`*) – Time step of the chosen stepping scheme. If *None*, a default value based on the stepper will be chosen.

Returns

The state at the final time point.

`property t_range: Tuple[float, float]`

start and end time of the simulation

Type

tuple

4.4.3 pde.solvers.explicit module

Defines an explicit solver supporting various methods

`class ExplicitSolver`(*pde: PDEBase*, *scheme: str = 'euler'*, *backend: str = 'auto'*, *adaptive: bool = False*, *tolerance: float = 1e-05*)

Bases: *`SolverBase`*

class for solving partial differential equations explicitly

Parameters

- **`pde`** (*`PDEBase`*) – The instance describing the pde that needs to be solved
- **`scheme`** (*`str`*) – Defines the explicit scheme to use. Supported values are 'euler' and 'runge-kutta' (or 'rk' for short).
- **`backend`** (*`str`*) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.
- **`adaptive`** (*`bool`*) – When enabled, the time step is adjusted during the simulation using the error tolerance set with *tolerance*.
- **`tolerance`** (*`float`*) – The error tolerance used in adaptive time stepping. This is used in adaptive time stepping to choose a time step which is small enough so the truncation error of a single step is below *tolerance*.

`dt_max = 10000000000.0`

`dt_min = 1e-10`

`info: Dict[str, Any]`

make_stepper(*state:* FieldBase, *dt=None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function using an explicit scheme

Parameters

- **state** (*FieldBase*) – An example for the state from which the grid and other information can be extracted
- **dt** (*float*) – Time step of the explicit stepping. 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)*

name = 'explicit'

4.4.4 pde.solvers.implicit module

Defines an implicit solver

exception ConvergenceError

Bases: `RuntimeError`

class ImplicitSolver(*pde:* PDEBase, *maxiter: int = 100*, *maxerror: float = 0.0001*, *backend: str = 'auto'*)

Bases: *SolverBase*

class for solving partial differential equations implicitly

Parameters

- **pde** (*PDEBase*) – The instance describing the pde that needs to be solved
- **maxiter** (*int*) – The maximal number of iterations per step
- **maxerror** (*float*) – The maximal error that is permitted in each step
- **backend** (*str*) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.

info: Dict[str, Any]

make_stepper(*state:* FieldBase, *dt=None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function using an implicit scheme

Parameters

- **state** (`FieldBase`) – An example for the state from which the grid and other information can be extracted
- **dt** (*float*) – Time step of the explicit stepping. 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)*

name = 'implicit'

4.4.5 pde.solvers.scipy module

Defines a solver using `scipy.integrate`

class ScipySolver(*pde: PDEBase*, *backend: str = 'auto'*, ***kwargs*)

Bases: *SolverBase*

class for solving partial differential equations using scipy

This class is a thin wrapper around `scipy.integrate.solve_ivp()`. In particular, it supports all the methods implemented by this function.

Parameters

- **pde** (*PDEBase*) – The instance describing the pde that needs to be solved
- **backend** (*str*) – Determines how the function is created. Accepted values are 'numpy` and 'numba'. Alternatively, 'auto' lets the code decide for the most optimal backend.
- ****kwargs** – All extra arguments are forwarded to `scipy.integrate.solve_ivp()`.

info: Dict[str, Any]

make_stepper(*state: FieldBase*, *dt: float = None*) → Callable[[*FieldBase*, float, float], float]

return a stepper function

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*.

name = 'scipy'

4.5 pde.storage package

Module defining classes for storing simulation data.

get_memory_storage	a context manager that can be used to create a MemoryStorage
MemoryStorage	store discretized fields in memory
FileStorage	store discretized fields in a hdf5 file

4.5.1 pde.storage.base module

Base classes for storing data

class StorageBase(*info: Optional[Dict[str, Any]] = None*, *write_mode: str = '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: FieldBase*, *time: float = None*) → None

add field to the storage

Parameters

- **field** (*FieldBase*) – The field that is added to the storage
- **time** (*float, optional*) – The time point

apply(*func: Callable*, *out: Optional[StorageBase] = None*, ***, *progress: bool = False*) → *StorageBase*

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: bool = False*) → None

truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

copy(*out: Optional[StorageBase] = None*, ***, *progress: bool = False*) → *StorageBase*

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: Tuple[int, ...]
the current data type.

Raises
RuntimeError – if data_type was not set

end_writing() → None
finalize the storage after writing

extract_field(*field_id: Union[int, str], label: str = None*) → *MemoryStorage*
extract the time course of a single field from a collection

Note: This might return a view into the original data, so modifying the returned data can also change the underlying original 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: Union[float, Tuple[float, float]] = None*) → *MemoryStorage*
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: Optional[*GridBase*]
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() → Iterator[Tuple[float, *FieldBase*]]
iterate over all times and stored fields, returning pairs

property shape: Optional[Tuple[int, ...]]
the shape of the stored data

start_writing(*field: FieldBase*, *info: Optional[Dict[str, Any]] = None*) → 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

times: Sequence[float]

tracker(*interval: Union[int, float, ConstantIntervals] = 1*) → *StorageTracker*
create object that can be used as a tracker to fill this storage

Parameters
interval – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.

Returns
The tracker that fills the current storage

Return type
StorageTracker

write_mode: str

class StorageTracker(*storage*, *interval: Union[ConstantIntervals, float, int, str] = 1*)
Bases: *TrackerBase*

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
- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.

finalize(*info: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters

info (*dict*) – Extra information from the simulation

handle(*field: FieldBase, t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

initialize(*field: FieldBase, info: Optional[Dict[str, Any]] = None*) → float

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

4.5.2 pde.storage.file module

Defines a class storing data on the file system using the hierarchical data format (hdf).

class FileStorage(*filename: str, info: Optional[Dict[str, Any]] = None, write_mode: str = 'truncate_once', max_length: int = None, compression: bool = True, keep_opened: bool = 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.

clear(*clear_data_shape: bool = False*)

truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

close() → None

close the currently opened file

property data

The actual data for all time

Type
ndarray

end_writing() → None

finalize the storage after writing.

This makes sure the data is actually written to a file when self.keep_opened == False

start_writing(*field: FieldBase, info: Optional[Dict[str, Any]] = None*) → 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

property times

The times at which data is available

Type
ndarray

write_mode: str

4.5.3 pde.storage.memory module

Defines a class storing data in memory.

class MemoryStorage(*times: Optional[Sequence[float]] = None, data: Optional[List[ndarray]] = None, field_obj: Optional[FieldBase] = None, info: Optional[Dict[str, Any]] = None, write_mode: str = '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: bool = False*) → None

truncate the storage by removing all stored data.

Parameters

clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

data: Any

classmethod from_collection (*storages: Sequence[StorageBase], label: str = None, *, rtol: float = 1e-05, atol: float = 1e-08*) → *MemoryStorage*

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: Optional[Sequence[float]] = None, fields: Optional[Sequence[FieldBase]] = None, info: Optional[Dict[str, Any]] = None, write_mode: str = 'truncate_once'*) → *MemoryStorage*

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.

start_writing (*field: FieldBase, info: Optional[Dict[str, Any]] = None*) → 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

times: Sequence[float]

write_mode: str

get_memory_storage(*field: FieldBase*, *info: Optional[Dict[str, Any]] = 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 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
math	Auxiliary mathematical functions
misc	Miscellaneous python functions
numba	Helper functions for just-in-time compilation with numba
output	Python functions for handling output
parameters	Infrastructure for managing classes with parameters
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
spherical	Module collecting functions for handling spherical geometry
typing	Provides support for mypy type checking of the package

4.6.1 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 This class provides hooks for converting arbitrary keys and values to strings, which are then stored in the database.

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: str = 'pickle', value_serialization: str = 'pickle', storage_dict: Optional[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., storage_dict = 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):
```

(continues on next page)

(continued from previous page)

```
        # `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 argument list.
- **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 argument list.
- **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*) → int

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*) → str

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: str*) → Callable

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: str*) → Callable

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*) → bool

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

4.6.2 pde.tools.config module

Handles configuration variables of the package

`Config`	class handling the package configuration
`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
`environment`	obtain information about the compute environment

class Config(*items: Optional[Dict[str, Any]] = None*, *mode: str = 'update'*)

Bases: `UserDict`

class handling the package configuration

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*.

to_dict() → Dict[str, Any]

convert the configuration to a simple dictionary

Returns
A representation of the configuration in a normal `dict`.

Return type
dict

check_package_version(*package_name: str*, *min_version: str*)

checks whether a package has a sufficient version

environment() → Dict[str, Any]

obtain information about the compute environment

Returns
information about the python installation and packages

Return type
dict

get_package_versions(*packages: List[str]*, ***, *na_str='not available'*) → Dict[str, str]

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

parse_version_str(*ver_str: str*) → List[int]

helper function converting a version string into a list of integers

4.6.3 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: bool = 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: Union[float, ndarray] = 0*, *inplace=False*) → *Cuboid*

dilate the cuboid by a certain amount in all directions

property centroid

contains_point(*points: ndarray*) → ndarray

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() → *Cuboid*

property corners: Tuple[ndarray, ndarray]

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: ndarray*, ***kwargs*) → *Cuboid*

create cuboid from bounds

Parameters
bounds (*list*) – Two dimensional array of axes bounds

Returns
cuboid with positive size

Return type
Cuboid

classmethod from_centerpoint(*centerpoint: ndarray*, *size: ndarray*, ***kwargs*) → *Cuboid*

create cuboid from two points

Parameters

- **centerpoint** (*list*) – Coordinates of the center
- **size** (*list*) – Size of the cuboid

Returns
cuboid with positive size

Return type
Cuboid

classmethod from_points(*p1: ndarray*, *p2: ndarray*, ***kwargs*) → *Cuboid*

create cuboid from two points

Parameters

- **p1** (*list*) – Coordinates of first corner point

- **p2** (*list*) – Coordinates of second corner point

Returns
cuboid with positive size

Return type
Cuboid

property mutable: bool

property size: ndarray

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: ndarray*, *dtype=None*, *writeable: bool = 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** – the resulant dtype
- **writeable** (*bool*) – Flag determining whether the results is writable

Returns
array with same data as *data* but with flags adjusted.

Return type
ndarray

4.6.4 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: TFunc*) → TFunc

decorator that replaces text in the docstring of a function

get_text_block(*identifier: str*) → str

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: TFunc*, *token: str*, *value: str*, *docstring: str = None*) → TFunc

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

4.6.5 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: Basic*, *signature: Optional[Sequence[Union[str, List[str]]]] = None*, ***, *user_funcs: Optional[Dict[str, Callable]] = None*, *consts: Optional[Dict[str, Union[int, float, complex, ndarray]]] = 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.

classmethod check_reserved_symbols(*symbols: Iterable[str], strict: bool = True*) → None

throws an error if reserved symbols are found

Parameters

- **symbols** (*iterable*) – A sequence or set of strings with symbols to check.
- **strict** (*bool*) – Flag determining whether an exception is raised

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: str*) → bool

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: bool = False*) → Callable[[...], Union[int, float, complex, ndarray]]

return numba function evaluating expression

Parameters
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

Returns
the compiled function

Return type
function

property rank: int

the rank of the expression

Type
int

abstract property shape: Tuple[int, ...]

class ScalarExpression(*expression: Union[float, str, ndarray, Basic, ExpressionBase] = 0, signature: Optional[Sequence[Union[str, List[str]]]] = None, *, user_funcs: Optional[Dict[str, Callable]] = None, consts: Optional[Dict[str, Union[int, float, complex, ndarray]]] = None, allow_indexed: bool = 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, which is 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`.
- **allow_indexed** (*bool*) – Whether to allow indexing of variables. If enabled, array variables are allowed to be indexed using square bracket notation.

copy() → *ScalarExpression*

return a copy of the current expression

derivatives

differentiate the expression with respect to all variables

differentiate(*var: str*) → *ScalarExpression*

return the expression differentiated with respect to var

property is_zero: bool

returns whether the expression is zero

Type
bool

shape: Tuple[int, ...] = ()

property value: Union[int, float, complex]
the value for a constant expression

Type
float

class TensorExpression(*expression: Union[float, str, ndarray, Basic,* ExpressionBase*]*, *signature: Optional[Sequence[Union[str, List[str]]]] = None*, *, *user_funcs: Optional[Dict[str, Callable]] = None*, *consts: Optional[Dict[str, Union[int, float, complex, ndarray]]] = 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, which is 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`.

derivatives
differentiate the expression with respect to all variables

differentiate(*var: str*) → *TensorExpression*
return the expression differentiated with respect to var

get_compiled_array(*single_arg: bool = True*) → Callable[[ndarray, Optional[ndarray]], ndarray]
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.

property shape: Tuple[int, ...]
the shape of the tensor

Type
tuple

property value

the value for a constant expression

evaluate(*expression: str, fields: Dict[str, DataFieldBase], *, bc: Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]] = 'auto_periodic_neumann', bc_ops: Optional[Dict[str, Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]], Sequence[Union[Dict[str, Union[Dict, str, BCBase]], Dict, str, BCBase, Tuple[Union[Dict, str, BCBase], Union[Dict, str, BCBase]]]]]]] = None, user_funcs: Optional[Dict[str, Callable]] = None, consts: Optional[Dict[str, Union[int, float, complex, ndarray]]] = None, label: str = None*) → *DataFieldBase*

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*) – Dictionary of the fields involved in the expression.
- **bc** – 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 list with one condition for each axis. For periodic axis, 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 a tuple of two conditions). 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 'natural' 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.
- **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: Union[str, int, float, complex], variables: Optional[Mapping[str, Number]] = None*) → Number

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

4.6.6 pde.tools.math module

Auxiliary mathematical functions

class SmoothData1D(*x, y, sigma: float = 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

sigma_auto_scale: float = 10

scale for setting automatic values for sigma

Type
float

4.6.7 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
`preserve_scalars`	decorator that makes vectorized methods work with scalars
`decorator_arguments`	make a decorator usable with and without arguments:
`skipUnlessModule`	decorator that skips a test when a module is not available
`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
`number_array`	convert an array with arbitrary dtype either to np.double or np.cdouble

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 change the deleter on a property.

getter(*fget*)

Descriptor to change the getter on a property.

setter(*fset*)

Descriptor to change the setter on a property.

decorator_arguments(*decorator: Callable*) → Callable

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` – the decorator that needs to be modified

Returns
the decorated function

`ensure_directory_exists`(*folder: Union[str, Path]*)

creates a folder if it not already exists

Parameters
`folder` (`str`) – path of the new folder

`estimate_computation_speed`(*func: Callable*, **args*, ***kwargs*) → float

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_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: Optional[Dict[str, Any]] = None*, *raise_serialization_error: bool = False*) → None

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

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: str*)

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: str*) → bool

check whether a python module is available

Parameters

module_name (*str*) – The name of the module

Returns

True if the module can be imported and *False* otherwise

number(*value: Union[int, float, complex, str]*) → Number

convert a value into a float or complex number

Parameters

value (*Number or str*) – The value which needs to be converted

Result:

Number: A complex number or a float if the imaginary part vanishes

number_array(*data: Union[int, float, complex, ndarray, Sequence[Union[int, float, complex, ndarray]], Sequence[Sequence[Any]]]*, *dtype=None*, *copy: bool = True*) → ndarray

convert an array with arbitrary dtype either to np.double or np.cdouble

Parameters

- **data** (*ndarray*) – The data that needs to be converted to a float 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 determined from *data* automatically.
- **copy** (*bool*) – Whether the data must be copied (in which case the original array is left untouched). Note that data will always be copied when changing the dtype.

Returns

An array with the correct dtype

Return type

ndarray

preserve_scalars(*method: TFunc*) → TFunc

decorator that makes vectorized methods work with scalars

This decorator allows to call functions that are written to work on numpy arrays to also accept python scalars, like *int* and *float*. Essentially, this wrapper turns them into an array and unboxes the result.

Parameters

method – The method being decorated

Returns

The decorated method

`skipUnlessModule`(*module_names: Union[Sequence[str], str]*) → Callable[[TFunc], TFunc]

decorator that skips a test when a module is not available

Parameters

`module_names` (*`str`*) – The name of the required module(s)

Returns

A function, so this can be used as a decorator

4.6.8 pde.tools.numba module

Helper functions for just-in-time compilation with numba

`class Counter`(*value: int = 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.

`increment`()

`convert_scalar`(*arr*)

helper function that turns 0d-arrays into scalars

This helps to avoid the bug discussed in https://github.com/numba/numba/issues/6000

`flat_idx`(*arr*, *i*)

helper function allowing indexing of scalars as if they arrays

`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: TFunc*, *signature=None*, *parallel: bool = False*, ***kwargs*) → TFunc

apply nb.jit with predefined arguments

Parameters

- **`function`** – The function which is jitted
- **`signature`** – Signature of the function to compile
- **`parallel`** (*`bool`*) – Allow parallel compilation of the function
- **`**kwargs`** – Additional arguments to *nb.jit*

Returns

Function that will be compiled using numba

jit_allocate_out (*func: Callable*, *parallel: bool = False*, *out_shape: Optional[Tuple[int, ...]] = None*, *num_args: int = 1*, ***kwargs*) → Callable

Decorator that compiles a function with allocating an output array.

This decorator compiles a function that takes the arguments *arr* and *out*. The point of this decorator is to make the *out* array optional by supplying an empty array of the same shape as *arr* if necessary. This is implemented efficiently by using `numba.generated_jit()`.

Parameters

- **func** – The function to be compiled
- **parallel** (*bool*) – Determines whether the function is jitted with parallel=True.
- **out_shape** (*tuple*) – Determines the shape of the *out* array. If omitted, the same shape as the input array is used.
- **num_args** (*int, optional*) – Determines the number of input arguments of the function.
- ****kwargs** – Additional arguments used in `numba.jit()`

Returns
The decorated function

make_array_constructor (*arr: ndarray*) → Callable[[], ndarray]

returns an array within a jitted function using basic information

Parameters
arr (*ndarray*) – The array that should be accessible within jit

Warning: A reference to the array needs to be retained outside the numba code to prevent garbage collection from removing the array

numba_dict (*data: Optional[Dict[str, Any]] = None*) → Optional[Dict]

converts a python dictionary to a numba typed dictionary

numba_environment () → Dict[str, Any]

return information about the numba setup used

Returns
(dict) information about the numba setup

4.6.9 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: str = '', footer: str = ''*)

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 html in a jupyter cell

class OutputBase

Bases: `object`

base class for output management

abstract show()

class SimpleProgress(*iterable=None, *args, **kwargs*)

Bases: `object`

indicates progress by printing dots to stderr

close(**args, **kwargs*)

refresh(**args, **kwargs*)

set_description(*msg: str, refresh: bool = True, *args, **kwargs*)

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() → Type[*SimpleProgress*]

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.

in_jupyter_notebook() → bool

checks whether we are in a jupyter notebook

4.6.10 pde.tools.parameters module

Infrastructure for managing classes with parameters

One aim is to allow easy management of inheritance of parameters.

`Parameter`	class representing a single parameter
`DeprecatedParameter`	a parameter that can still be used normally but is deprecated
`HideParameter`	a helper class that allows hiding parameters of the parent classes
`Parameterized`	a mixin that manages the parameters of a class
`get_all_parameters`	get a dictionary with all parameters of all registered classes

class DeprecatedParameter(*name: str*, *default_value=None*, *cls=<class 'object'>*, *description: str = ''*, *hidden: bool = False*, *extra: ~Optional[~Dict[str, ~Any]] = None*)

Bases: `Parameter`

a parameter that can still be used normally but is deprecated

initialize a parameter

Parameters

- **name** (`str`) – The name of the parameter
- **default_value** – The default value
- **cls** – The type of the parameter, which is used for conversion
- **description** (`str`) – A string describing the impact of this parameter. This description appears in the parameter help
- **hidden** (`bool`) – Whether the parameter is hidden in the description summary
- **extra** (`dict`) – Extra arguments that are stored with the parameter

class HideParameter(*name: str*)

Bases: `object`

a helper class that allows hiding parameters of the parent classes

Parameters

name (`str`) – The name of the parameter

class Parameter(*name: str*, *default_value=None*, *cls=<class 'object'>*, *description: str = ''*, *hidden: bool = False*, *extra: ~Optional[~Dict[str, ~Any]] = None*)

Bases: `object`

class representing a single parameter

initialize a parameter

Parameters

- **name** (`str`) – The name of the parameter

- **default_value** – The default value
- **cls** – The type of the parameter, which is used for conversion
- **description** (*str*) – A string describing the impact of this parameter. This description appears in the parameter help
- **hidden** (*bool*) – Whether the parameter is hidden in the description summary
- **extra** (*dict*) – Extra arguments that are stored with the parameter

convert(*value=None*)

converts a *value* into the correct type for this parameter. If *value* is not given, the default 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 – The value to convert

Returns
The converted value, which is of type *self.cls*

class Parameterized(*parameters: Optional[Dict[str, Any]] = None*)

Bases: `object`

a mixin that manages the parameters of a class

initialize the parameters of the object

Parameters
parameters (*dict*) – A dictionary of parameters to change the defaults. The allowed parameters can be obtained from `get_parameters()` or displayed by calling `show_parameters()`.

get_parameter_default(*name*)

return the default value for the parameter with *name*

Parameters
name (*str*) – The parameter name

classmethod get_parameters(*include_hidden: bool = False*, *include_deprecated: bool = False*, *sort: bool = True*) → Dict[str, *Parameter*]

return a dictionary of parameters that the class supports

Parameters

- **include_hidden** (*bool*) – Include hidden parameters
- **include_deprecated** (*bool*) – Include deprecated parameters
- **sort** (*bool*) – Return ordered dictionary with sorted keys

Returns
a dictionary of instance of `Parameter` with their names as keys.

Return type
dict

parameters_default: Sequence[Union[*Parameter*, *HideParameter*]] = []

show_parameters(*description: Optional[bool] = None*, *sort: bool = False*, *show_hidden: bool = False*, *show_deprecated: bool = False*)

show all parameters in human readable format

Parameters

- **description** (*bool*) – Flag determining whether the parameter description is shown. The default is to show the description only when we are in a jupyter notebook environment.
- **sort** (*bool*) – Flag determining whether the parameters are sorted
- **show_hidden** (*bool*) – Flag determining whether hidden parameters are shown
- **show_deprecated** (*bool*) – Flag determining whether deprecated parameters are shown
- **default_value** (*bool*) – Flag determining whether the default values or the current values are shown

All flags default to *False*.

get_all_parameters(*data: str = 'name'*) → Dict[str, Any]

get a dictionary with all parameters of all registered classes

Parameters

data (*str*) – Determines what data is returned. Possible values are 'name', 'value', or 'description', to return the respective information about the parameters.

sphinx_display_parameters(*app*, *what*, *name*, *obj*, *options*, *lines*)

helper function to display parameters in sphinx documentation

Example

This function should be connected to the 'autodoc-process-docstring' event like so:

app.connect('autodoc-process-docstring', sphinx_display_parameters)

4.6.11 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.

parse_duration(*value: str*) → timedelta

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

4.6.12 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: str = None, show: bool = 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 shown in the plot
- **show** (*bool*) – Flag determining whether plots are actually shown

class JupyterPlottingContext(*title: str = None*, *show: bool = 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: bool = 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: Any*, *parameters: Optional[Dict[str, Any]] = 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: str = None*, *show: bool = 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

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: matplotlib.cm.ScalarMappable*, *ax=None*, *aspect: float = 20*, *pad_fraction: float = 0.5*, *label: str = ''*, ***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: str = None*, *show: bool = True*) → *PlottingContextBase*

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() → bool

try to detect whether we are in an ipython shell, e.g., a jupyter notebook

napari_add_layers(*viewer: napari.viewer.Viewer*, *layers_data: Dict[str, dict]*)

adds layers to a napari viewer

Parameters

- **viewer** (napar i.viewer.Viewer) – The napari application
- **layers_data** (*dict*) – Data for all layers that will be added.

napari_viewer(*grid: GridBase*, *run: bool = None*, *close: bool = False*, ***kwargs*) → Generator[napari.viewer.Viewer, None, None]

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`

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):
        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):
```

(continues on next page)

(continued from previous page)

```
        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):
        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.

4.6.13 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_colored_noise(*shape: Tuple[int, ...], dx=1.0, exponent: float = 0, scale: float = 1, rng: Optional[Generator] = None*) → Callable[[], ndarray]

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.

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** – Exponent ν of the power spectrum
- **scale** – Scaling factor Γ determining noise strength
- **rng** (Generator) – Random number generator (default: `default_rng()`)

Returns
a function returning a random realization

Return type
callable

4.6.14 pde.tools.spherical module

Module collecting functions for handling spherical geometry

The coordinate systems use the following convention for polar coordinates (r, ϕ), where r is the radial coordinate and ϕ is the polar angle:

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

Similarly, for spherical coordinates (r, θ, ϕ), where r is the radial coordinate, θ is the azimuthal angle, and ϕ is the polar angle, we use

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

The module also provides functions for handling spherical harmonics. These spherical harmonics are described by the degree l and the order m or, alternatively, by the mode k. The relation between these values is

$$k = l(l + 1) + m$$

and

$$l = \text{floor}(\sqrt{k})$$
$$m = k - l(l + 1)$$

We will use these indices interchangeably, although the mode k is preferred internally. Note that we also consider axisymmetric spherical harmonics, where the order is always zero and the degree l and the mode k are thus identical.

radius_from_volume	Return the radius of a sphere with a given volume
volume_from_radius	Return the volume of a sphere with a given radius
surface_from_radius	Return the surface area of a sphere with a given radius
spherical_index_k	returns the mode *k* from the degree *degree* and order *order*
spherical_index_lm	returns the degree *l* and the order *m* from the mode *k*
spherical_index_count	return the number of modes for all indices <= l
spherical_index_count_optimal	checks whether the modes captures all orders for maximal degree
spherical_harmonic_symmetric	axisymmetric spherical harmonics with degree *degree*, so *m=0*.
spherical_harmonic_real	real spherical harmonics of degree l and order m
spherical_harmonic_real_k	real spherical harmonics described by mode k

class PointsOnSphere(*points*)

Bases: `object`

class representing points on an n-dimensional unit sphere

Parameters
points (`ndarray`) – The list of points on the unit sphere

get_area_weights(*balance_axes: bool = True*)

return the weight of each point associated with the unit cell size

Parameters
balance_axes (*bool*) – Flag determining whether the weights should be chosen such that the weighted average of all points is the zero vector

Returns
The weight associated with each point

Return type
`ndarray`

get_distance_matrix()

calculate the (spherical) distances between each point

Returns
the distance of each point to each other

Return type
`ndarray`

get_mean_separation() → float

float: calculates the mean distance to the nearest neighbor

classmethod make_uniform(*dim: int*, *num_points: int = None*)

create uniformly distributed points on a sphere

Parameters

- **dim** (*int*) – The dimension of space
- **num_points** (*int, optional*) – The number of points to generate. Note that for one-dimensional spheres (intervals), only exactly two points can be generated

write_to_xyz(*path: str*, *comment: str = ''*, *symbol: str = 'S'*)

write the point coordinates to a xyz file

Parameters

- **path** (*str*) – location of the file where data is written
- **comment** (*str, optional*) – comment that is written to the second line
- **symbol** (*str, optional*) – denotes the symbol used for the atoms

get_spherical_polygon_area(*vertices: ndarray*, *radius: float = 1*) → float

Calculate the surface area of a polygon on the surface of a sphere. Based on equation provided here: http://mathworld.wolfram.com/LHuiliersTheorem.html Decompose into triangles, calculate excess for each

Adapted from https://github.com/tylerjereddy/spherical-SA-docker-demo Licensed under MIT License (see copy in root of this project)

Parameters

- **vertices** (`ndarray`) – List of vertices (using Cartesian coordinates) that describe the corners of the polygon. The vertices need to be oriented.
- **radius** (*float*) – Radius of the sphere

haversine_distance(*point1: ndarray*, *point2: ndarray*) → ndarray

Calculate the haversine-based distance between two points on the surface of a sphere. Should be more accurate than the arc cosine strategy. See, for example: https://en.wikipedia.org/wiki/Haversine_formula

Adapted from https://github.com/tylerjereddy/spherical-SA-docker-demo Licensed under MIT License (see copy in root of this project)

Parameters

- **point1** (`ndarray`) – First point(s) on the sphere (given in Cartesian coordinates)
- **point2** (`ndarray`) – Second point on the sphere Second point(s) on the sphere (given in Cartesian coordinates)

Returns
The distances between the points

Return type
`ndarray`

`make_radius_from_volume_compiled`(*dim: int*) → Callable[[TNumArr], TNumArr]

Return a function calculating the radius of a sphere with a given volume

Parameters
`dim` (*`int`*) – Dimension of the space

Returns
A function that takes a volume and returns the radius

Return type
function

`make_surface_from_radius_compiled`(*dim: int*) → Callable[[TNumArr], TNumArr]

Return a function calculating the surface area of a sphere

Parameters
`dim` (*`int`*) – Dimension of the space

Returns
A function that takes a radius and returns the surface area

Return type
function

`make_volume_from_radius_compiled`(*dim: int*) → Callable[[TNumArr], TNumArr]

Return a function calculating the volume of a sphere with a given radius

Parameters
`dim` (*`int`*) – Dimension of the space

Returns
A function that takes a radius and returns the volume

Return type
function

`points_cartesian_to_spherical`(*points: ndarray*) → ndarray

Convert points from Cartesian to spherical coordinates

Parameters
`points` (`ndarray`) – Points in Cartesian coordinates

Returns
Points (r, θ, φ) in spherical coordinates

Return type
`ndarray`

`points_spherical_to_cartesian`(*points: ndarray*) → ndarray

Convert points from spherical to Cartesian coordinates

Parameters
`points` (`ndarray`) – Points in spherical coordinates (r, θ, φ)

Returns
Points in Cartesian coordinates

Return type
`ndarray`

radius_from_surface(*surface: TNumArr*, *dim: int*) → TNumArr

Return the radius of a sphere with a given surface area

Parameters

- **surface** (float or `ndarray`) – Surface area of the sphere
- **dim** (*int*) – Dimension of the space

Returns
Radius of the sphere

Return type
float or `ndarray`

radius_from_volume(*volume: TNumArr*, *dim: int*) → TNumArr

Return the radius of a sphere with a given volume

Parameters

- **volume** (float or `ndarray`) – Volume of the sphere
- **dim** (*int*) – Dimension of the space

Returns
Radius of the sphere

Return type
float or `ndarray`

spherical_harmonic_real(*degree: int*, *order: int*, *θ: float*, *φ: float*) → float

real spherical harmonics of degree l and order m

Parameters

- **degree** (*int*) – Degree l of the spherical harmonics
- **order** (*int*) – Order m of the spherical harmonics
- **θ** (*float*) – Azimuthal angle (in $[0, \pi]$) at which the spherical harmonics is evaluated.
- **φ** (*float*) – Polar angle (in $[0, 2\pi]$) at which the spherical harmonics is evaluated.

Returns
The value of the spherical harmonics

Return type
float

spherical_harmonic_real_k(*k: int*, *θ: float*, *φ: float*) → float

real spherical harmonics described by mode k

Parameters

- **k** (*int*) – Combined index determining the degree and order of the spherical harmonics
- **θ** (*float*) – Azimuthal angle (in $[0, \pi]$) at which the spherical harmonics is evaluated.
- **φ** (*float*) – Polar angle (in $[0, 2\pi]$) at which the spherical harmonics is evaluated.

Returns
The value of the spherical harmonics

Return type
float

`spherical_harmonic_symmetric`(*degree: int*, *θ: float*) → float

axisymmetric spherical harmonics with degree *degree*, so *m=0*.

Parameters

- **`degree`** (*`int`*) – Degree of the spherical harmonics
- **`θ`** (*`float`*) – Azimuthal angle at which the spherical harmonics is evaluated (in $[0, \pi]$)

Returns
The value of the spherical harmonics

Return type
float

`spherical_index_count`(*l: int*) → int

return the number of modes for all indices <= l

The returned value is one less than the maximal mode *k* required.

Parameters
`l` (*`int`*) – Maximal degree of the spherical harmonics

Returns
The number of modes

Return type
int

`spherical_index_count_optimal`(*k_count: int*) → bool

checks whether the modes captures all orders for maximal degree

Parameters
`k_count` (*`int`*) – The number of modes considered

`spherical_index_k`(*degree: int*, *order: int = 0*) → int

returns the mode *k* from the degree *degree* and order *order*

Parameters

- **`degree`** (*`int`*) – Degree of the spherical harmonics
- **`order`** (*`int`*) – Order of the spherical harmonics

Raises
`ValueError` – if *order* < *-degree* or *order* > *degree*

Returns
a combined index k

Return type
int

`spherical_index_lm`(*k: int*) → Tuple[int, int]

returns the degree *l* and the order *m* from the mode *k*

Parameters
`k` (*`int`*) – The combined index for the spherical harmonics

Returns
The degree *l* and order *m* of the spherical harmonics assoicated with the combined index

Return type
tuple

surface_from_radius(*radius: TNumArr*, *dim: int*) → TNumArr

Return the surface area of a sphere with a given radius

Parameters

- **radius** (float or `ndarray`) – Radius of the sphere
- **dim** (*int*) – Dimension of the space

Returns

Surface area of the sphere

Return type

float or `ndarray`

volume_from_radius(*radius: TNumArr*, *dim: int*) → TNumArr

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`

4.6.15 pde.tools.typing module

Provides support for mypy type checking of the package

4.7 pde.trackers package

Classes for tracking simulation results in controlled intervals

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
`ProgressTracker`	Tracker showing the progress of the simulation
`PrintTracker`	Tracker printing data to a stream (default: stdout)
`PlotTracker`	Tracker plotting data on screen, to files, or writes a movie
`LivePlotTracker`	PlotTracker with defaults for live plotting
`DataTracker`	Tracker storing custom data obtained by calling a function
`SteadyStateTracker`	Tracker interrupting the simulation once steady state is reached
`RuntimeTracker`	Tracker interrupting the simulation once a duration has passed
`ConsistencyTracker`	Tracker interrupting the simulation when the state is not finite
`InteractivePlotTracker`	Tracker showing the state interactively in napari

Some trackers can also be referenced by name for convenience when using them in simulations. The lit of supported names is returned by `get_named_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 interrupt the simulation by raising the special exception `StopIteration`.

For each tracker, the interval at which it is called can be decided using one of the following classes:

`ConstantIntervals`	class representing equidistantly spaced time intervals
`LogarithmicIntervals`	class representing logarithmically spaced time intervals
`RealtimeIntervals`	class representing time intervals spaced equidistantly in real time

4.7.1 pde.trackers.base module

Base classes for trackers

exception FinishedSimulation

Bases: `StopIteration`

exception for signaling that simulation finished successfully

class TrackerBase(*interval: Union[ConstantIntervals, float, int, str] = 1*)

Bases: `object`

base class for implementing trackers

Parameters

interval – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control.

finalize(*info: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters

info (*dict*) – Extra information from the simulation

classmethod from_data(*data: Union[TrackerBase, str], **kwargs*) → *TrackerBase*

create tracker class from given data

Parameters

data (*str or TrackerBase*) – Data describing the tracker

Returns

An instance representing the tracker

Return type

`TrackerBase`

abstract handle(*field: FieldBase, t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation

- **t** (*float*) – The associated time

initialize(*field: FieldBase, info: Optional[Dict[str, Any]] = None*) → float

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: Optional[List[TrackerBase]] = 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 of trackers that are to be handled.

finalize(*info: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters
info (*dict*) – Extra information from the simulation

classmethod from_data(*data: Optional[Union[Sequence[Union[TrackerBase, str]], TrackerBase, str]], **kwargs*) → *TrackerCollection*

create tracker collection from given data

Parameters
data – Data describing the tracker collection

Returns
An instance representing the tracker collection

Return type
TrackerCollection

handle(*state: FieldBase, t: float, atol: float = 1e-08*) → float

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: FieldBase*, *info: Optional[Dict[str, Any]] = None*) → float
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

get_named_trackers() → Dict[str, Type[*TrackerBase*]]
returns all named trackers

Returns
a mapping of names to the actual tracker classes.

Return type
dict

4.7.2 pde.trackers.interactive module

Special module for defining an interactive tracker that uses napari to display fields

class InteractivePlotTracker(*interval: Union[ConstantIntervals, float, int, str] = '0:01'*, *close: bool = True*, *show_time: bool = 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

```python
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

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control.
- **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: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters
info (*dict*) – Extra information from the simulation

handle(*state: FieldBase, t: float*) → None

handle data supplied to this tracker

Parameters

- **state** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

initialize(*state: FieldBase, info: Optional[Dict[str, Any]] = None*) → float

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: FieldBase, t_initial: float = 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: bool = 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: FieldBase, t: float*)

update the state in the napari viewer

Parameters

- **state** (*pde.fields.base.FieldBase*) – The new state
- **t** (*float*) – Current time

napari_process (*data_channel: Queue, initial_data: Dict[str, Dict[str, Any]], t_initial: float = None, viewer_args: Optional[Dict[str, Any]] = None*)

`multiprocessing.Process` running napari

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

4.7.3 pde.trackers.intervals module

Module defining classes for time intervals for trackers

The provided interval classes are:

ConstantIntervals	class representing equidistantly spaced time intervals
LogarithmicIntervals	class representing logarithmically spaced time intervals
RealtimeIntervals	class representing time intervals spaced equidistantly in real time

class ConstantIntervals (*dt: float = 1, t_start: float = None*)

Bases: `object`

class representing equidistantly spaced time intervals

Parameters

- **dt** (*float*) – The duration between subsequent intervals. 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.

copy()

return a copy of this instance

next(*t: float*) → float

computes the next time point based on the current time t

Parameters

t (*float*) – The current time point of the simulation

IntervalType

alias of `ConstantIntervals`

class LogarithmicIntervals(*dt_initial: float = 1, factor: float = 1, t_start: float = None*)

Bases: `ConstantIntervals`

class representing logarithmically spaced time intervals

Parameters

- **dt_initial** (*float*) – The initial duration between subsequent intervals. This is measured in simulation time units.
- **factor** (*float*) – The factor by which the time between intervals is increased every time. Values larger than one lead to time intervals that are increasingly further apart.
- **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.

next(*t: float*) → float

computes the next time point based on the current time t

Parameters

t (*float*) – The current time point of the simulation

class RealtimeIntervals(*duration: Union[float, str], dt_initial: float = 0.01*)

Bases: `ConstantIntervals`

class representing time intervals 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 realtime seconds) that the intervals 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 intervals. This is measured in simulation time units.

next(*t: float*) → float

computes the next time point based on the current time t

Parameters

t (*float*) – The current time point of the simulation

get_interval(*interval: Union[ConstantIntervals, float, int, str]*) → *ConstantIntervals*

create IntervalType from various data formats

If interval is of type `IntervalType` it is simply returned

4.7.4 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
`ProgressTracker`	Tracker showing the progress of the simulation
`PrintTracker`	Tracker printing data to a stream (default: stdout)
`PlotTracker`	Tracker plotting data on screen, to files, or writes a movie
`LivePlotTracker`	PlotTracker with defaults for live plotting
`DataTracker`	Tracker storing custom data obtained by calling a function
`SteadyStateTracker`	Tracker interrupting the simulation once steady state is reached
`RuntimeTracker`	Tracker interrupting the simulation once a duration has passed
`ConsistencyTracker`	Tracker interrupting the simulation when the state is not finite
`MaterialConservationTracker`	Tracking interrupting the simulation when material conservation is broken

class CallbackTracker(*func: Callable*, *interval: Union[ConstantIntervals, float, int, str] = 1*)

Bases: `TrackerBase`

Tracker calling a function periodically

Parameters

- **func** – 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`.
- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control.

handle(*field: FieldBase*, *t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (`FieldBase`) – The current state of the simulation
- **t** (`float`) – The associated time

class ConsistencyTracker(*interval: Optional[Union[ConstantIntervals, float, int, str]] = None*)

Bases: `TrackerBase`

Tracker interrupting the simulation when the state is not finite

Parameters

interval – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control. The default value *None* checks for consistency approximately every (real) second.

handle (*field:* FieldBase, *t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

name = 'consistency'

class DataTracker (*func: Callable*, *interval: Union[*ConstantIntervals*, float, int, str] = 1*, *filename: str = None*)

Bases: *CallbackTracker*

Tracker storing custom data obtained by calling a function

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** – 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.
- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.
- **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`.

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: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters

info (*dict*) – Extra information from the simulation

handle(*field:* FieldBase, *t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

to_file(*filename: str*, ***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(*interval: Union[*ConstantIntervals*, float, int, str] = '0:03'*, ***, *show: bool = True*, *max_fps: float = 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 *interval* is set something more suitable for interactive plotting. In particular, this tracker can be enabled by simply listing 'plot' as a tracker.

Parameters

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.
- **title** (*str*) – Text to show in the title. The current time point will be appended to this text, so include a space for optimal results.
- **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., *np.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.

name = 'plot'

class MaterialConservationTracker(*interval: Union[ConstantIntervals, float, int, str] = 1*, *atol: float = 0.0001*, *rtol: float = 0.0001*)

Bases: *TrackerBase*

Tracking interrupting the simulation when material conservation is broken

Parameters

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.
- **atol** (*float*) – Absolute tolerance for amount deviations
- **rtol** (*float*) – Relative tolerance for amount deviations

handle(*field: FieldBase*, *t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

initialize(*field: FieldBase*, *info: Optional[Dict[str, Any]] = None*) → float

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 PlotTracker(*interval: Union[ConstantIntervals, float, int, str] = 1*, ***, *title: Union[str, Callable] = 'Time: {time:g}'*, *output_file: str = None*, *movie: Union[str, Path, Movie] = None*, *show: bool = None*, *max_fps: float = inf*, *plot_args: Dict[str, Any] = None*)

Bases: *TrackerBase*

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.

Parameters

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control.
- **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 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 *False*, the files are created in the background. This option can slow down a simulation severely. For the default value of *None*, the images are only shown if neither *output_file* nor *movie* is set.
- **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., *np.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.

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, e.g., by setting the *dpi* argument or the *frame_rate*.

finalize(*info: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters
info (`dict`) – Extra information from the simulation

handle(*state:* FieldBase, *t: float*) → None

handle data supplied to this tracker

Parameters

- **state** (`FieldBase`) – The current state of the simulation
- **t** (`float`) – The associated time

initialize(*state:* FieldBase, *info: Optional[Dict[str, Any]] = None*) → float

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

class PrintTracker(*interval: ~Union[~ConstantIntervals, float, int, str] = 1, stream: ~IO[str] = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Bases: `TrackerBase`

Tracker printing data to a stream (default: stdout)

Parameters

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control.
- **stream** – The stream used for printing

handle(*field: FieldBase, t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (`FieldBase`) – The current state of the simulation
- **t** (`float`) – The associated time

name = 'print'

class ProgressTracker(*interval: Optional[Union[ConstantIntervals, float, int, str]] = None, ndigits: int = 5, leave: bool = True*)

Bases: `TrackerBase`

Tracker showing the progress of the simulation

Parameters

- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control. The default value *None* updates the progress bar approximately every (real) second.
- **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)

finalize(*info: Optional[Dict[str, Any]] = None*) → None

finalize the tracker, supplying additional information

Parameters
info (`dict`) – Extra information from the simulation

handle(*field: FieldBase, t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (`FieldBase`) – The current state of the simulation

- **t** (*float*) – The associated time

initialize (*field:* FieldBase, *info: Optional[Dict[str, Any]] = None*) → float

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 RuntimeTracker (*max_runtime: Union[int, float, str], interval: Union[ConstantIntervals, float, int, str] = 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()*.
- **interval** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in *intervals* can be given for more control.

handle (*field:* FieldBase, *t: float*) → None

handle data supplied to this tracker

Parameters

- **field** (FieldBase) – The current state of the simulation
- **t** (*float*) – The associated time

initialize (*field:* FieldBase, *info: Optional[Dict[str, Any]] = None*) → float

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 SteadyStateTracker (*interval: Optional[Union[ConstantIntervals, float, int, str]] = None, atol: float = 1e-08, rtol: float = 1e-05, progress: bool = False*)

Bases: *TrackerBase*

Tracker interrupting the simulation once steady state is reached

Steady state is obtained when the state does not change anymore. This is the case when the derivative is close to zero. Concretely, the current state *cur* is compared to the state *prev* at the previous time step. Convergence is assumed when `abs(prev - cur) <= dt * (atol + rtol * cur)` for all points in the state. Here, *dt* denotes the time that elapsed between the two states that are compared.

Parameters

- **`interval`** – Determines how often the tracker interrupts the simulation. Simple numbers are interpreted as durations measured in the simulation time variable. Alternatively, a string using the format 'hh:mm:ss' can be used to give durations in real time. Finally, instances of the classes defined in `intervals` can be given for more control. 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

`handle`(*field:* FieldBase, *t: float*) → None

handle data supplied to this tracker

Parameters

- **`field`** (`FieldBase`) – The current state of the simulation
- **`t`** (`float`) – The associated time

`name = 'steady_state'`

`progress_bar_format = 'Convergence: {percentage:3.0f}%|{bar}|[{elapsed}<{remaining}]'`

determines the format of the progress bar shown when *progress* = *True*

4.8 pde.visualization package

Functions and classes for visualizing simulations.

`movies`	Functions for creating movies of simulation results
`plotting`	Functions and classes for plotting simulation data

4.8.1 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: str*, *framerate: float = 30*, *dpi: float = 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
- ****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() → bool

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: StorageBase*, *filename: str*, ***, *progress: bool = True*, *dpi: float = 150*, *show_time: bool = True*, *plot_args: Optional[Dict[str, Any]] = None*, *movie_args: Optional[Dict[str, Any]] = None*) → 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.

- **`dpi`** (*`float`*) – Resolution of the movie
- **`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*).

`movie_multiple`(*storage:* StorageBase, *filename:* *str*, *quantities=None*, *scale: Union[str, float, Tuple[float, float]] = 'automatic'*, *progress:* *bool* = *True*) → None

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.

`movie_scalar`(*storage:* StorageBase, *filename:* *str*, *scale: Union[str, float, Tuple[float, float]] = 'automatic'*, *extras: Optional[Dict[str, Any]] = None*, *progress:* *bool* = *True*, *tight:* *bool* = *False*, *show:* *bool* = *True*) → None

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

4.8.2 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: FieldBase, quantities=None, scale: Union[str, float, Tuple[float, float]] = 'automatic', fig=None, title: str = None, tight: bool = False, show: bool = 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*) – 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: StorageBase, quantities=None, scale: Union[str, float, Tuple[float, float]] = 'automatic', tight: bool = False, show: bool = True*) → *ScalarFieldPlot*

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`

make_movie(*storage: StorageBase*, *filename: str*, *progress: bool = True*) → None

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.

savefig(*path: str*, ***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: FieldBase*, *title: str = None*) → None

update the plot with the given fields

Parameters

- **fields** – 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.

extract_field(*fields: FieldBase*, *source: Union[None, int, Callable] = None*, *check_rank: int = None*) → *DataFieldBase*

Extracts a single field from a possible collection.

Parameters

- **fields** (`FieldBase`) – The field from which data is extracted

- **source** (*int or callable, 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 = ScalarField, 1 = VectorField, …).

Returns
The extracted field

Return type
`DataFieldBase`

plot_interactive(*storage:* StorageBase, *time_scaling:* str *= 'exact'*, *viewer_args: Optional[Dict[*str*, Any]] = 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*) – Arguments passed to `napari.viewer.Viewer` to affect the viewer.
- ****kwargs** – Extra arguments passed to the plotting function

plot_kymograph(*storage:* StorageBase, *field_index:* int *= None*, *scalar:* str *= 'auto'*, *extract:* str *= 'auto'*, *colorbar:* bool *= True*, *transpose:* bool *= False*, **args*, *title:* str *= None*, *filename:* str *= None*, *action:* str *= 'auto'*, *ax_style: Optional[Dict[*str*, Any]] = None*, *fig_style: Optional[Dict[*str*, Any]] = None*, *ax=None*, ***kwargs*) → *PlotReference*

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*) – 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 *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.
- **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 in 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: StorageBase, scalar: str = 'auto', extract: str = 'auto', colorbar: bool = True, transpose: bool = False, resize_fig: bool = True, *args, title: str = None, constrained_layout: bool = True, filename: str = None, action: str = 'auto', fig_style: Optional[Dict[str, Any]] = None, fig=None, **kwargs*) → List[*PlotReference*]

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: StorageBase, quantities=None, *args, title: str = None, filename: str = None, action: str = 'auto', ax_style: Optional[Dict[str, Any]] = None, fig_style: Optional[Dict[str, Any]] = None, ax=None, **kwargs*) → *PlotReference*

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** – 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 *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.
- **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 in 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

INDEX