1 Getting started
1.1 Install using pip ....................................................... 3
1.2 Install using conda ................................................... 3
1.3 Install from source .................................................. 3
1.4 Package overview .................................................... 4

2 Examples
2.1 Plotting a vector field .............................................. 7
2.2 Solving Laplace’s equation in 2d ................................... 8
2.3 Plotting a scalar field in cylindrical coordinates ................. 9
2.4 Solving Poisson’s equation in 1d ................................ 10
2.5 Simple diffusion equation ......................................... 11
2.6 Kuramoto-Sivashinsky - Using PDE class ....................... 13
2.7 Spherically symmetric PDE ......................................... 14
2.8 Diffusion on a Cartesian grid ...................................... 16
2.9 Stochastic simulation ................................................. 17
2.10 Setting boundary conditions ...................................... 18
2.11 1D problem - Using PDE class ................................. 19
2.12 Brusselator - Using the PDE class ............................... 21
2.13 Diffusion equation with spatial dependence ..................... 22
2.14 Using simulation trackers .......................................... 23
2.15 Schrödinger’s Equation ............................................ 24
2.16 Kuramoto-Sivashinsky - Using custom class .................. 26
2.17 Custom Class for coupled PDEs ................................. 27
2.18 1D problem - Using custom class ............................... 29
2.19 Solver comparison .................................................. 30
2.20 Visualizing a scalar field .......................................... 31
2.21 Kuramoto-Sivashinsky - Compiled methods .................... 33
2.22 Custom PDE class: SIR model .................................. 34
2.23 Brusselator - Using custom class ............................... 36

3 User manual
3.1 Mathematical basics ............................................... 39
3.2 Basic usage .......................................................... 40
3.3 Advanced usage ..................................................... 42
3.4 Performance .......................................................... 47
3.5 Contributing code .................................................... 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Citing the package</td>
<td>50</td>
</tr>
<tr>
<td>3.7</td>
<td>Code of Conduct</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>Reference manual</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>pde.fields package</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>pde.grids package</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>pde.pdes package</td>
<td>126</td>
</tr>
<tr>
<td>4.4</td>
<td>pde.solvers package</td>
<td>138</td>
</tr>
<tr>
<td>4.5</td>
<td>pde.storage package</td>
<td>144</td>
</tr>
<tr>
<td>4.6</td>
<td>pde.tools package</td>
<td>150</td>
</tr>
<tr>
<td>4.7</td>
<td>pde.trackers package</td>
<td>181</td>
</tr>
<tr>
<td>4.8</td>
<td>pde.visualization package</td>
<td>194</td>
</tr>
</tbody>
</table>

Python Module Index

Index
The \textit{py-pde} python package provides methods and classes useful for solving partial differential equations (PDEs) of the form

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

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 \( x \) and time \( t \). The framework also supports stochastic differential equations in the Itô representation, where the noise is represented by \( \eta \) 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, \textit{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.

\textbf{Contents}
CHAPTER 1

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:

<table>
<thead>
<tr>
<th>Package</th>
<th>Minimal version</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib</td>
<td>3.1</td>
<td>Visualizing results</td>
</tr>
<tr>
<td>numba</td>
<td>0.50</td>
<td>Just-in-time compilation to accelerate numerics</td>
</tr>
<tr>
<td>numpy</td>
<td>1.18</td>
<td>Handling numerical data</td>
</tr>
<tr>
<td>scipy</td>
<td>1.4</td>
<td>Miscellaneous scientific functions</td>
</tr>
<tr>
<td>sympy</td>
<td>1.5</td>
<td>Dealing with user-defined mathematical expressions</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Package</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>h5py</td>
<td>Storing data in the hierarchical file format</td>
</tr>
<tr>
<td>napari</td>
<td>Displaying images interactively</td>
</tr>
<tr>
<td>pandas</td>
<td>Handling tabular data</td>
</tr>
<tr>
<td>pyfftw</td>
<td>Faster Fourier transforms</td>
</tr>
<tr>
<td>tqdm</td>
<td>Display progress bars during calculations</td>
</tr>
</tbody>
</table>

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*, *PolarGrid*, *SphericalGrid*, and *CylindricalGrid*, 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.
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 \( \mathbf{u} = (\sin(x), \cos(x)) \).
This example shows how to solve a 2d Laplace equation with spatially varying boundary conditions.
```python
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 1.036 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 CylindricalGrid, ScalarField

grid = CylindricalGrid(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 0.369 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 0.076 seconds)

### 2.5 Simple diffusion equation

This example solves a simple diffusion equation in two dimensions.
from pde import DiffusionPDE, ScalarField, UnitGrid

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

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

Total running time of the script: ( 0 minutes 2.701 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:

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

![](image.png)

Out:

| 0% | 0/10.0 [00:00<?, ?it/s] |
| Initializing: 0% | 0/10.0 [00:00<?, ?it/s] |
| 0% | 0/10.0 [00:04<?, ?it/s] |
| 0% | 0.01/10.0 [00:05<1:27:02, 522.78s/it] |
| 0% | 0.02/10.0 [00:05<48:35, 292.09s/it] |
| 0% | 0.03/10.0 [00:05<32:21, 194.74s/it] |
| 4% | 0.42/10.0 [00:05<02:13, 13.91s/it] |

(continues on next page)
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 6.007 seconds

2.7 Spherically symmetric PDE

This example illustrates how to solve a PDE in a spherically symmetric geometry.
```python
from pde import DiffusionPDE, ScalarField, SphericalGrid

grid = SphericalGrid(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)
```

(continues on next page)
result.plot(kind="image")

Total running time of the script: 0 minutes 1.768 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 [00:02<?, ?it/s]
  1% | 0.01/1.0 [00:02<04:15, 257.58s/it]
  2% | 0.02/1.0 [00:02<02:24, 147.87s/it]
  4% | 0.04/1.0 [00:02<01:10, 73.94s/it]
  4% | 0.04/1.0 [00:02<01:10, 73.96s/it]
100% | 1.0/1.0 [00:02<00:00, 2.96s/it]
100% | 1.0/1.0 [00:02<00:00, 2.96s/it]
```

x

---

Chapter 2. Examples
```
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 3.088 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
```

(continues on next page)
state = ScalarField.random_harmonic(grid)  # generate initial condition

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

Total running time of the script: ( 0 minutes 3.446 seconds)

2.10 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 [00:01<?, ?it/s]
  0% | 0.01/10.0 [00:02<37:38, 226.04s/it]
  0% | 0.02/10.0 [00:02<22:39, 136.23s/it]
  0% | 0.035/10.0 [00:02<12:55, 77.86s/it]
```
from pde import DiffusionPDE, ScalarField, UnitGrid

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

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

result = 
result.plot()

Total running time of the script: ( 0 minutes 2.857 seconds)

2.11 1D problem - Using PDE class

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

\[
\partial_t \phi = 6 \phi \partial_x \phi - \partial_x^3 \phi
\]

which we implement using the PDE.
from math import pi
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

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

# solve the equation and store the trajectory
storage = MemoryStorage()
res = 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 1.203 seconds)
2.12 Brusselator - Using the PDE class

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

\[
\begin{align*}
\partial_t u &= D_0 \nabla^2 u + a - (1 + b)u + vu^2 \\
\partial_t v &= D_1 \nabla^2 v + bu - vu^2
\end{align*}
\]

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

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

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

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

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

# simulate the pde
tracker = PlotTracker(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 11.002 seconds)
2.13 Diffusion equation with spatial dependence

This example solves the Diffusion equation with a heterogeneous diffusivity:

\[
\frac{\partial c}{\partial t} = \nabla (D(r) \nabla c)
\]

using the \texttt{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, \texttt{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,

\[
\frac{\partial c}{\partial t} = D \nabla^2 c + \nabla D \cdot \nabla c
\]

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

# Expanded definition of the PDE
diffusivity = "1.01 + tanh(x)"
term_1 = f"$\{\text{diffusivity}\} \ast \text{laplace}(c)$"
term_2 = f"$\text{dot}(\text{gradient}(\{\text{diffusivity}\}), \text{gradient}(c))$"
eq = PDE("$c$": $f$$\{\text{term}_1\} + \{\text{term}_2\}$", bc="$\text{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 5.924 seconds)

### 2.14 Using simulation trackers

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

![Space-time plot](image-url)

Out:

<table>
<thead>
<tr>
<th>Time</th>
<th>0/3.0 [00:00&lt;?, ?it/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initializing</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td>7%</td>
</tr>
</tbody>
</table>

(continues on next page)
import pde

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

storage = pde.MemStorage()

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 3.424 seconds)

### 2.15 Schrödinger’s Equation

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

\[ i\partial_t \psi = -\nabla^2 \psi \]

Note that the example imposes Neumann conditions at the wall, so the wave packet is expected to reflect off the wall.
from math import sqrt
from pde import PDE, CartesianGrid, MemoryStorage, ScalarField, plot_kymograph

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

# 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({"\psi": f"I * laplace(\psi)"})  # 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 2.049 seconds)
2.16 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:

$$\partial_t u = -\frac{1}{2} |\nabla u|^2 - \nabla^2 u - \nabla^4 u$$
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="natural")
        state_lap2 = state_lap.laplace(bc="natural")
        state_grad = state.gradient(bc="natural")
        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 4.326 seconds)

2.17 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:

\[
\begin{align*}
\partial_t u & = \nabla^2 u + u(u - \alpha)(1 - u) + w \\
\partial_t w & = \epsilon u
\end{align*}
\]

Here, \(\alpha\) denotes the external stimulus and \(\epsilon\) defines the recovery time scale. We implement this as a custom PDE class below.
from pde import FieldCollection, PDEBase, UnitGrid

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

    def __init__(self, stimulus=0.5, \tau=10, a=0, b=0, bc="natural"):
        self.bc = bc
        self.stimulus = stimulus
        self.\tau = \tau
        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.\tau

        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()
2.18 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("natural")[0]
        return 6 * state * grad_x - grad_x.laplace("natural")
```

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

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

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

Total running time of the script: 0 minutes 1.626 seconds

## 2.19 Solver comparison

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

Deviation: 8.7e-09

![explicit solver](image1)

![scipy solver](image2)

Out:

Diagnostic information from first run:

```
Diagnostic information from first run:
{'controller': {'package_version': '0.13.3', 't_start': 0, 't_end': 1.0, 'solver_class': 'ExplicitSolver', 'solver_start': '2021-03-15 15:35:37.963128', 'profiler': {'solver': 0.7058655839999943, 'tracker': 3.7398000003463494e-05}, 'successful': True, 'stop_reason': 'Reached final time', 'solver_duration': '0:00:00.705791', 't_final': 1.0}, 'solver': {'class': 'ExplicitSolver', 'pde_class': 'DiffusionPDE', 'dt': 0.001, 'steps': 1000, 'scheme': 'euler', 'state_modifications': 0.0, 'backend': 'numba', 'stochastic': False}}
```

Diagnostic information from second run:

```
Diagnostic information from second run:
{'controller': {'package_version': '0.13.3', 't_start': 0, 't_end': 1.0, 'solver_class': 'ScipySolver', 'solver_start': '2021-03-15 15:35:37.963128', 'profiler': {'solver': 3.404099992157595e-05}, 'successful': True, 'stop_reason': 'Reached final time', 'solver_duration': '0:00:00.705912', 't_final': 1.0}, 'solver': {'class': 'ScipySolver', 'pde_class': 'DiffusionPDE', 'dt': None, 'steps': 50, 'stochastic': False, 'backend': 'numba'}}
```

(continues on next page)
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 the standard scipy solver
solver2 = pde.ScipySolver(eq)
controller2 = pde.Controller(solver2, t_range=1, tracker=None)
sol2 = controller2.run(field)
sol2.label = "scipy solver"
print("Diagnostic information from second run:")
print(controller2.diagnostics)
print()

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

Total running time of the script: ( 0 minutes 3.224 seconds)

### 2.20 Visualizing a scalar field

This example displays methods for visualizing scalar fields.
```python
import matplotlib.pyplot as plt
import numpy as np

from pde import CylindricalGrid, ScalarField

# create a scalar field with some noise
grid = CylindricalGrid(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)
```

Total running time of the script: ( 0 minutes 0.315 seconds)
2.21 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 [00:06<?, ?it/s]
0%  | 0.01/10.0 [00:07<1:58:19, 710.66s/it]
0%  | 0.02/10.0 [00:07<1:04:15, 386.37s/it]
0%  | 0.03/10.0 [00:07<42:48, 257.59s/it]
4%  | 0.36/10.0 [00:07<03:26, 21.47s/it]
4%  | 0.36/10.0 [00:07<03:27, 21.50s/it]
100% | 10.0/10.0 [00:07<00:00, 1.29it/s]
100% | 10.0/10.0 [00:07<00:00, 1.29it/s]
```
import numba as nb
from pde import PDEBase, ScalarField, UnitGrid

class KuramotoSivashinskyPDE(PDEBase):
    """ Implementation of the normalized Kuramoto-Sivashinsky equation """
    def __init__(self, bc="natural"):
        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):
        """ numba-compiled implementation of the PDE """
        gradient_squared = state.grid.get_operator("gradient_squared", bc=self.bc)
        laplace = state.grid.get_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 7.864 seconds)

### 2.22 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:

\[
\begin{align*}
\partial_t s &= D \nabla^2 s - \beta is \\
\partial_t i &= D \nabla^2 i + \beta is - \gamma i \\
\partial_t r &= D \nabla^2 r + \gamma i
\end{align*}
\]

Here, \(D\) is the diffusivity, \(\beta\) the infection rate, and \(\gamma\) the recovery rate.
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="natural"):
        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
\[ \begin{align*}
    r &= \text{ScalarField}(s.\text{grid}, \text{data}=1 - s - i, \text{label}="\text{Recovered}") \\
    \text{return FieldCollection([s, i, r])}
\end{align*} \]

```python
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 5.366 seconds)

## 2.23 Brusselator - Using custom class

This example implements the Brusselator with spatial coupling,

\[ \begin{align*}
    \partial_t u &= D_0 \nabla^2 u + a - (1 + b)u + vu^2 \\
    \partial_t v &= D_1 \nabla^2 v + bu - vu^2
\end{align*} \]

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 \texttt{PDE} class; see \textit{the example}. However, that implementation is less flexible and might be more difficult to extend later.
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="natural"):
        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
        return rhs
    def _make_pde_rhs_numba(self, state):
        """ numba-compiled implementation of the PDE """
        d0, d1 = self.diffusivity
        a, b = self.a, self.b
        laplace = state.grid.get_operator("laplace", bc=self.bc)

@nb.jit  
(continues on next page)
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 8.833 seconds)
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 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 $\Delta x$ is constant. If a given axis covers values in a range $[x_{\text{min}}, x_{\text{max}}]$, a discretization with $N$ support points can then be thought of as covering the axis with $N$ equal-sized boxes; see inset. Field values are then specified for each box, i.e., the support points lie at the centers of the box:

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

$$\Delta x = \frac{x_{\text{max}} - x_{\text{min}}}{N}$$

which is also indicated in the inset. Differential operators are implemented using the usual second-order central 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.
3.1.2 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 (`PolarGrid`) and three dimensions (`SphericalGrid`), as well as the special class `CylindricalGrid` 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

```python
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

```python
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 defines the geometry on which the PDE is solved. In the simplest case, the solution is then obtained by running

```python
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

```python
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:

```python
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

Boundary conditions can be specified for both sides of each axis individually. 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:

```python
bc_lower = {'type': 'value', 'value': 4}
bcs_upper = {'type': 'derivative', 'value': 2}
bc = [bc_lower, bc_upper]
g = pde.UnitGrid([16])
f = pde.ScalarField(g)
f.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 = {'type': 'value', 'value': 3}`.

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

```python
bc_x = [{"derivative": 0.1}, {"value": "sin(y / 2)"}]
bcs_y = {"value": "sqrt(1 + cos(x))"}

g = UnitGrid([32, 32])
f = pde.ScalarField(g)
f.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.

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:

```python
grid = pde.UnitGrid([16, 16], periodic=[True, False])
f = pde.ScalarField(grid)
bc = ['periodic', {'derivative': 0}]
f.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 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

```python
class KuramotoSivashinskyPDE(PDEBase):
    def evolution_rate(self, state, t=0):
        r""" numpy implementation of the evolution equation """
        state_lapacian = state.laplace(bc='natural')
        return (- state_lapacian.laplace(bc='natural')
                - state_lapacian
                - 0.5 * state.gradient(bc='natural').to_scalar('squared_sum'))
```

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

```python
class KuramotoSivashinskyPDE(PDEBase):
    def __init__(self, diffusivity=1, bc='natural', bc_laplace='natural'):
        r""" 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):
        r""" 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.

3.3.3 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.get_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.get_operator('gradient', bc='natural')
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 get_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='natural')
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:

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

### 3.3.4 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:

```python
from pde.tools.numba import jit
class KuramotoSivashinskyPDE(PDEBase):
    def __init__(self, diffusivity=1, bc='natural', bc_laplace='natural'):
        """ 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='natural')
        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 class attributes locally available
        diffusivity = self.diffusivity
```

(continues on next page)
# create operators
laplace_u = state.grid.get_operator('laplace', bc=self.bc)
gradien_u = state.grid.get_operator('gradient', bc=self.bc)
laplace2_u = state.grid.get_operator('laplace', bc=self.bc_laplace)

@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 class 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 class attributes locally available
    diffusivity = self.diffusivity

    # create operators
    laplace_u = state.grid.get_operator('laplace', bc=self.bc)
    gradient_u = state.grid.get_operator('gradient', bc=self.bc)
    laplace2_u = state.grid.get_operator('laplace', bc=self.bc_laplace)
    dot = VectorField(state.grid).get_dot_operator()
    dim = state.grid.dim

    @jit
def pde_rhs(state_data, t=0):
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.

### 3.3.5 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.parallel_threshold** Minimal number of support points before multithreading or multiprocessing is enabled in the numba compilations. (Default value: 65536)
- **numba.parallel** Determines whether multiple cores are used in numba-compiled code. (Default value: True)
- **numba.fastmath** Determines whether the fastmath flag is set during compilation. This affects the precision of the mathematical calculations. (Default value: True)
- **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)

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

```python
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 or even better than 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 (lower is better) for two different boundary conditions:

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 CylindricalGrid.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 run 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:

```bibtex
@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:

```ris
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
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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScalarField</td>
<td>Single scalar field on a grid</td>
</tr>
<tr>
<td>VectorField</td>
<td>Single vector field on a grid</td>
</tr>
<tr>
<td>Tensor2Field</td>
<td>Single tensor field of rank 2 on a grid</td>
</tr>
<tr>
<td>FieldCollection</td>
<td>Collection of fields defined on the same grid</td>
</tr>
</tbody>
</table>

Inheritance structure of the classes:

```
FieldBase
  |     |
  v     v
DataFieldBase  ScalarField  Tensor2Field
  |     |     |
  v     v     v
FieldCollection  VectorField
```

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

```python
class DataFieldBase(grid: GridBase, data: ArrayLike = None, *, label: str = None, dtype=None)
    Bases: pde.fields.base.FieldBase
    abstract base class for describing fields of single entities
    grid
        The underlying grid defining the discretization
        Type GridBase
    data
        Data values at the support points of the grid
        Type ndarray
    shape
        Shape of the data field
        Type tuple
    label
        Name of the field
        Type str
```

**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 data is copied from the supplied array. The resulting field will contain real data unless the `data` argument contains complex values.
- `label` (`str`, optional) – Name of the field
- `dtype` (numpy `dtype`) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from `data` automatically.

**Methods**

- `add_interpolated(point: numpy.ndarray, amount: ArrayLike) → None`  
  deprecated alias of method `insert`

- `average`  
  determine the average of data
  
  This is calculated by integrating each component of the field over space and dividing by the grid volume

- `copy(data: ArrayLike = None, *, label: str = None, dtype=None) → TDataField`  
  return a copy of the data, but not of the grid

  **Parameters**

  - `data` (ndarray, optional) – Data values at the support points of the grid that define the field.
  - `label` (`str`, optional) – Name of the copied field
  - `dtype` (numpy `dtype`) – The data type of the field. If omitted, it will be determined from `data` automatically.
data_shape
the shape of the data at each grid point
Type tuple

fluctuations
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 \text{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 \text{ScalarField}.
Return type ndarray
Type ndarray
classmethod from_state(attributes: dict, data: numpy.ndarray = None) -> DataFieldBase
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: BoundariesData = 'natural') -> NumberOrArray
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. \{ARG\_BOUNDARIES\}

Returns The discretized values on the boundary
Return type ndarray

get_image_data(scalar: str = 'auto', transpose: bool = False, **kwargs) -> dict
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 be 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
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`

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: numpy.ndarray, amount: ArrayLike) -> 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.

`Integral`

`interpolate(point: numpy.ndarray, *, backend: str = 'numba', method: str = 'linear', fill: Number = None, **kwargs) -> NumberOrArray`

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: 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.

**magnitude**

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_get_boundary_values**(*axis: int, upper: bool, bc: BoundariesData = 'natural') → Callable[[numpy.ndarray, numpy.ndarray], NumberOrArray]

make a function calculating field values 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. {ARG_BOUNDARIES}

**Returns** A function returning the values on the boundary. The function has the signature (*data=None, out=None*), which allows specifying an input and an output *ndarray*. If *data* is omitted, the data of the current field is used. The resulting interpolation is written to *out* if it is present. Otherwise, a new array is created.

**Return type** callable

**make_interpolator**(*backend: str = 'numba', method: str = 'linear', fill: Number = None, **kwargs) → Callable[[numpy.ndarray, numpy.ndarray], NumberOrArray]

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: dict = None, fig_style: dict = 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 figure. If the argument is set to show, matplotlib.pyplot.show() will be called to show the plot, if the value is create, the figure will be created, but not shown, and 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. If None, a new figure with a single axes is created.
- ****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, seed: int = None)
create a field of random values with colored noise
```

The spatially correlated values obey

\[ \langle c_i(k)c_j(k') \rangle = \Gamma^2 |k|^{\nu} \delta_{ij} \delta(k - k') \]

in spectral space. 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 $\nu$ of the power spectrum
• `scale (float)` – Scaling factor $\Gamma$ determining noise strength
• `label (str, optional)` – Name of the field
• `seed (int, optional)` – Seed of the random number generator. If `None`, the current state is not changed.

```python
class method random_harmonic(grid: GridBase, modes: int = 3, harmonic=<ufunc 'cos'>, axis_combination=<ufunc 'multiply'>, label: str = None, seed: int = None)
create a random field build from harmonics

The resulting fields will be highly correlated in space and can thus serve for testing differential operators.

With the default settings, the resulting field $c_i(x)$ is given by

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

where $N$ is the number of spatial dimensions, each with length $L_\alpha$, $M$ is the number of modes given by `modes`, and $a_{ij}$ 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 axes 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
- `seed (int, optional)` – Seed of the random number generator. If `None`, the current state is not changed.

```python
class method random_normal(grid: GridBase, mean: float = 0, std: float = 1, scaling: str = 'physical', label: str = None, seed: int = None)
create field with normal distributed random values

These values are uncorrelated in space.

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 useful for physical quantities, which vary less in larger volumes).

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

• **seed** (*int*, optional) – Seed of the random number generator. If None, the current state is not changed.

```
classmethod random_uniform(grid: GridBase, vmin: float = 0, vmax: float = 1, label: str = None, seed: int = None)

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

• **seed** (*int*, optional) – Seed of the random number generator. If None, the current state is not changed.

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

applies Gaussian smoothing with the given standard deviation

This function respects periodic boundary conditions of the underlying grid, using reflection when no periodicity is specified.

**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.

```
to_scalar(scalar: str = 'auto', *, label: str = None)
```

```
classmethod unserialize_attributes(attributes: Dict[str, str])
```

unserializes the given attributes

**Parameters** attributes (*dict*) – The serialized attributes

**Returns** The unserialized attributes

**Return type** *dict*

```
class FieldBase(grid: GridBase, data: ArrayLike = None, *, label: str = None)
```

abstract base class for describing (discretized) fields

**Parameters**

• **grid** (*GridBase*) – Grid defining the space on which this field is defined

• **data** (*array*, optional) – Field values at the support points of the grid
- **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)* – Other field this is compared to
- **accept_scalar** *(bool, optional)* – Determines whether it is acceptable that `other` is an instance of `ScalarField`.

`attributes`

describes the state of the instance (without the data)

**Type** `dict`

`attributes_serialized`

serialized version of the attributes

**Type** `dict`

`conjugate() → TField`

returns complex conjugate of the field

`copy(data: ArrayLike = None, *, label: str = None, dtype=None) → TField`

**data**

discretized data at the support points

**Type** `ndarray`

`dtype`

returns the numpy dtype of the underlying data

`classmethod from_file(filename: str) → FieldBase`

create field by reading file

**Parameters** filename *(str)* – Path to the file being read

`classmethod from_state(attributes: dict, data: numpy.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

`get_image_data() → dict`

`get_line_data(scalar: str = 'auto', extract: str = 'auto') → dict`
grid
   The grid on which the field is defined
   Type  GridBase

imag
   Imaginary part of the field
   Type  FieldBase

is_complex
   whether the field contains real or complex data
   Type  bool

label
   the name of the field
   Type  str

plot(*args, **kwargs)
plot_interactive(viewer_args: dict = 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

readonly = False

real
   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.

   Parameters
   - filename (str) – Path where the data is stored
   - metadata (dict) – A dictionary of additional information that is stored with the file. Note that not all formats support metadata.
   - **kwargs – Additional parameters may be supported for some formats

classmethod unserialize_attributes(attributes: Dict[str, str]) → dict
   unserializes the given attributes

   Parameters attributes (dict) – The serialized attributes
   Returns The unserialized attributes
   Return type  dict
### 4.1.2 pde.fields.collection module

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

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

Bases: pde.fields.base.FieldBase

Collection of fields defined on the same grid

**Parameters**
- **fields** – Sequence of the individual fields
- **data** (ndarray) – Data of the fields. If None, the data is instead taken from the individual fields given by `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.

**Warning:** If `data` is given and `copy_fields == False`, the data in the individual fields is overwritten by the associated `data`.
```

```python
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.
```

**attributes**

describes the state of the instance (without the data)

Type dict

**attributes_serialized**

serialized version of the attributes

Type dict

**averages**

averages of all fields

**copy**

return a copy of the data, but not of the grid

**Parameters**
- **data** (ndarray, optional) – Data values at the support points of the grid that define the field. Note that the data is not copied but used directly.
- **label** *(str, optional)* – Name of the copied field
- **dtype** *(numpy dtype)* – The data type of the field. If omitted, it will be determined from *data* automatically.

**fields**

the fields of this collection

**Type** list

**classmethod from_scalar_expressions**(grid: GridBase, expressions: Sequence[str], *, label: str = None, labels: Optional[Sequence[str]] = None, **kwargs**) → 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.
- **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, data: 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

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

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*

**integrals**
integrals of all fields

**interpolate_to_grid** *(grid: GridBase, *, backend: str = 'numba', method: str = 'linear', fill: 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*

**labels**
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*

**magnitudes**
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: dict = 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 figure. If the argument is set to *show* *matplotlib.pyplot.show()* will be called to show the plot, if the value is *create*, the figure will be created, but not shown, and 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 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*) – Smallest random value
• **vmax** (*float*) – Largest random value
• **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
```

unserializes the given attributes

- **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: ArrayLike = None, *, label: str = None, dtype=None)
```

Bases: `pde.fields.base.DataFieldBase`

Single scalar field on a grid

- **grid**
  - The underlying grid defining the discretization
  - Type `GridBase`

- **data**
  - Scalar values at the support points of the grid
  - Type `np.ndarray`

- **label**
  - Name of the field
  - Type `str`

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 data is copied from the supplied array. The resulting field will contain real data unless the `data` argument contains complex values.
- **label** (*str, optional*) – Name of the field
• **dtype (numpy dtype)** – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

```python
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.
- **label** (*str*, *optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from *data* automatically.

```python
classmethod from_image(path: Union[pathlib.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
- **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

```python
gradient(bc: BoundariesData, out: Optional[VectorField] = None, *, label: str = 'gradient') → 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`). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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*.

- **out** (*VectorField*, *optional*) – Optional vector field to which the result is written.
gradient_squared

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

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'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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`.

- `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.

- `out` (ScalarField, optional) – Optional scalar field to which the result is written.

- `label` (str, optional) – Name of the returned field

Returns the result of applying the operator

Return type `ScalarField`

laplace

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'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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`.

- `out` (ScalarField, optional) – Optional scalar field to which the result is written.

Returns the result of applying the operator

Return type `ScalarField`
• **label** *(str, optional)* – Name of the returned field

**Returns**  the result of applying the operator

**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*

**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

**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. Other alternatives are abs, norm, or norm_squared. The default auto is to return a (unchanged) copy of the field.

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

**Returns**  the scalar field after applying the operation

**Return type**  *pde.fields.scalar.ScalarField*
4.1.4 pde.fields.tensorial module

Defines a tensorial field of rank 2 over a grid

class Tensor2Field(grid: GridBase, data: ArrayLike = None, *, label: str = None, dtype=None)

Single tensor field of rank 2 on a grid

grid
The underlying grid defining the discretization

   Type GridBase
data
Tensor components at the support points of the grid

   Type ndarray

label
Name of the field

   Type str

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 data is copied from the supplied array. The resulting field will contain real data unless the data argument contains complex values.
- label (str, optional) – Name of the field
- dtype (numpy dtype) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from data automatically.

divergence(bc: BoundariesData, out: Optional[VectorField] = None, *, label: str = 'divergence') -> VectorField

apply (tensor) divergence 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.
- out (VectorField, optional) – Optional scalar field to which the result is written.
- label (str, optional) – Name of the returned field

Returns the result of applying the operator

Return type VectorField
```python
dot(other: Union[VectorField, Tensor2Field], out: Union[VectorField, Tensor2Field, None] = 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**

the result of applying the dot operator

**Return type** *VectorField or Tensor2Field*

`get_dot_operator()` → Callable

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.

**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.

**integral**

integral of each component over space

**Type** *ndarray*

`make_dot_operator(backend: str = 'numba', *, conjugate: bool = True) → Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray], numpy.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.
**rank** = 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

**to_scalar**(*scalar*: str = 'auto', *, **label**: str = 'scalar') → ScalarField
return a scalar field by applying method

The invariants of the tensor field \( A \) are

\[
I_1 = \text{tr}(A) \\
I_2 = \frac{1}{2} [(\text{tr}(A)^2 - \text{tr}(A^2)] \\
I_3 = \text{det}(A)
\]

where \( \text{tr} \) denotes the trace and \( \text{det} \) 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), 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** pde.fields.scalar.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** holding the trace

**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** holding the 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**: ArrayLike = None, *, **label**: str = None, dtype=None)

**Bases**: pde.fields.base.DataFieldBase

Single vector field on a grid
grid
The underlying grid defining the discretization
Type `GridBase`
data
Vector components at the support points of the grid
Type `ndarray`
label
Name of the field
Type `str`

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 data is copied from the supplied array. The resulting field will contain real data unless the `data` argument contains complex values.
- `label` (str, optional) – Name of the field
- `dtype` (numpy dtype) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from `data` automatically.

divergence
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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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`.
- `out` (ScalarField, optional) – Optional scalar field to which the result is written.
- `label` (str, optional) – Name of the returned field

Returns the result of applying the operator
Return type `ScalarField`
dot
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 the result of applying the dot operator

Return type *ScalarField or VectorField*

### 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.
- **label** (*str, optional*) – Name of the field
- **dtype** (*numpy dtype*) – The data type of the field. All the numpy dtypes are supported. If omitted, it will be determined from data automatically.

### 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. All the numpy dtypes are supported. If omitted, it will be determined from data automatically.

Returns the resulting vector field

Return type *VectorField*

### get_dot_operator() → Callable

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.
**get_vector_data**

__signature__
```
get_vector_data(transpose: bool = False, max_points: int = None, **kwargs) → dict
```

__returns__
Information useful for plotting an vector field

__return_type__
dict

__description__
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`

**gradient**

__signature__
```
gradient(bc: BoundariesData, out: Optional[Tensor2Field] = None, *, label: str = 'gradient') → Tensor2Field
```

__returns__
the result of applying the operator

__return_type__
Tensor2Field

**integral**

__type__
ndarray

**laplace**

__signature__
```
laplace(bc: BoundariesData, out: Optional[VectorField] = None, *, label: str = 'vector laplacian') → VectorField
```

__returns__
apply vector 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.
- **out** (Tensor2Field, optional) – Optional tensorial field to which the result is written.
- **label** (str, optional) – Name of the returned field

**Chapter 4. Reference manual**
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`.

- **out** *(VectorField, optional)* – Optional vector field to which the result is written.
- **label** *(str, optional)* – Name of the returned field

**Returns** the result of applying the operator

**Return type** *VectorField*

```python
make_dot_operator(backend: str = 'numba', *, conjugate: bool = True) → Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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.

```python
make_outer_prod_operator(backend: str = 'numba') → Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

**Warning:** This function does not check types or dimensions.

**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.

```python
outer_product(other: VectorField, out: Tensor2Field = None, *, label: str = None) → Tensor2Field
```

calculate the outer product of this vector field with another

**Parameters**
- **other** *(VectorField)* – The second vector field
- **out** *(pde.fields.tensorial.Tensor2Field, optional)* – Optional tensorial field to which the result is written.
- **label** *(str, optional)* – Name of the returned field

**rank** = 1
```python
def to_scalar(scalar='auto', *, label='scalar', scalar=str):
    """return a scalar field by applying method"
    The two tensor invariants are given by

    Parameters
    • **scalar** (str) – Choose the method to use. Possible choices are `norm` (the default),
      `max`, `min`, `squared_sum`, or `norm_squared`.
    • **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`. Points given in these coordinates can be mapped to coordinates in Cartesian space using
the methods `point_to_cartesian()` and its inverse. Moreover, points can be mapped to cell indices using
the methods `point_to_cell()`.

<table>
<thead>
<tr>
<th>Grid Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitGrid</td>
<td>d-dimensional Cartesian grid with unit discretiza-</td>
</tr>
<tr>
<td>CartesianGrid</td>
<td>tion in all directions</td>
</tr>
<tr>
<td>PolarGrid</td>
<td>d-dimensional Cartesian grid with uniform discret-</td>
</tr>
<tr>
<td>SphericalGrid</td>
<td>ization for each axis</td>
</tr>
<tr>
<td>CylindricalGrid</td>
<td>2-dimensional polar grid assuming angular symme-</td>
</tr>
<tr>
<td></td>
<td>try</td>
</tr>
<tr>
<td></td>
<td>3-dimensional spherical grid assuming spherical</td>
</tr>
<tr>
<td></td>
<td>symmetry</td>
</tr>
<tr>
<td></td>
<td>3-dimensional cylindrical grid assuming polar sym-</td>
</tr>
<tr>
<td></td>
<td>metry</td>
</tr>
</tbody>
</table>

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**

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:

```python
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 condition is ‘periodic’. 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

```python
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

\[
\text{bc}_y = \{\{'\text{value}\': 'y**2'\}, \{'\text{derivative}\': '-\sin(x)'\}\}
\]

**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

\[
\text{bc}_y = \{\{'\text{type}\': 'mixed', 'value': 2, 'const': 7}, \\
\{'\text{curvature}\': 2\}\}
\]

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, \(\text{bc}_y = \{'\text{value}\': 2\}\) 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

\[
\text{field} = \text{ScalarField}(\text{UnitGrid}([16, 16], \text{periodic}=\text{False})) \\
\text{field}.\text{laplace}(\text{bc}='\text{value}=2')
\]

imposes a value of 2 on all sides of the grid. Finally, the special value ‘natural’ imposes periodic boundary conditions for periodic axis and a vanishing derivative otherwise. For example,

\[
\text{field} = \text{ScalarField}(\text{UnitGrid}([16, 16], \text{periodic}=\text{[True, False]})) \\
\text{field}.\text{laplace}(\text{bc}=\text{'natural'})
\]

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, which corresponds to an inwards flux. Conversely, negative derivatives are associated with effluxes.

**Boundaries overview**

The `boundaries` package defines the following classes:

**Local 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

• **ExtrapolateBC**: Extrapolate boundary points linearly from the two points closest to the boundary

**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`.

```python
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)
        check whether the values at the boundaries have the correct rank
```
Parameters **rank** *(tuple)* – The rank of the value that is stored with this boundary condition

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

**copy***(value=None) → Boundaries*

create a copy of the current boundaries

**Parameters**

- **value** *(float or array, optional)* – If given, this changes the value stored with the boundary conditions. The interpretation of this value depends on the type of boundary condition.

- **copy.grid** *(bool)* – Whether the grid should also be copied

**differentiated**

with differentiated versions of all boundary conditions

**Type** Domain

**extract_component** *(*indices*) → Boundaries*

extracts the boundary conditions of the given extract_component.

**Parameters** *

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

**classmethod from_data** *(grid: GridBase, boundaries, rank: int = 0) → 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 value associated with the boundary conditions.

**classmethod get_help** () → str

Return information on how boundary conditions can be set

**grid = None**

The grid for which the boundaries are defined

**Type** GridBase

**periodic**

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

**Type** ndarray

**scale_value** *(factor: float = 1)*

scales the value of the boundary condition with the given factor

**Parameters** **value** *(float)* – Scales the value associated with the boundary condition by the factor
`set_value(value=0)`
set the value of all non-periodic boundaries

**Parameters**

- `value (float or array)` — Sets the value stored with the boundary conditions. The interpretation of this value depends on the type of boundary condition.

### `pde.grids.boundaries.axis` module

This module handles the boundaries of a single axis of a grid. There are generally only two options, depending on whether the axis of the underlying grid is defined as periodic or not. If it is periodic, the class `BoundaryPeriodic` should be used, while non-periodic axes have more option, which are represented by `BoundaryPair`.

#### `BoundaryAxisBase`

**Bases:** object

base class for defining boundaries of a single axis in a grid

**axis = None**
The axis along which the boundaries are defined

<table>
<thead>
<tr>
<th>Type</th>
<th>int</th>
</tr>
</thead>
</table>

**grid = None**
The grid for which the boundaries are defined

<table>
<thead>
<tr>
<th>Type</th>
<th>GridBase</th>
</tr>
</thead>
</table>

#### `BoundaryPair`

**Bases:** `pde.grids.boundaries.axis.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)**
check whether the values at the boundaries have the correct rank

**Parameters**

- `rank (int)` — The rank of the value that is stored with 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

**differentiated**
differentiated version of this boundary condition

<table>
<thead>
<tr>
<th>Type</th>
<th>BoundaryPair</th>
</tr>
</thead>
</table>

**extract_component(*indices)**
evaluates 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) → 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 value associated with the boundary conditions.

**Returns** the instance created from the data

**Return type** `BoundaryPair`

**Throws:** `ValueError` if `data` cannot be interpreted as a boundary pair

```python
def 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`
```

classmethod get_help() → str
Return information on how boundary conditions can be set
```

def get_point_evaluator(fill: numpy.ndarray = None) → Callable[[numpy.ndarray, Tuple[int, ...]], NumberOrArray]
return a function to evaluate values at a given point

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`
```

def make_region_evaluator() → Callable[[numpy.ndarray, Tuple[int, ...]], Tuple[NumberOrArray, NumberOrArray, NumberOrArray]]
return a function to evaluate values in a neighborhood of a point

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 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`
```

def make_virtual_point_evaluators() → Tuple[Callable, Callable]
returns two functions evaluating the value at virtual support points

Parameters
- **size** (*int*) – Number of support points along the axis
- **dx** (*float*) – Discretization, i.e., distance between 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`

```plaintext
periodic = False
```

```plaintext
scale_value(factor: float = 1)
```

 scales the value of the boundary condition with the given factor

**Parameters**
- **value** (*float*) – Scales the value associated with the boundary condition by the factor

```plaintext
set_value(value=0)
```

 set the value of both boundary conditions

**Parameters**
- **value** (*float or array*) – Sets the value stored with the boundary conditions. The interpretation of this value depends on the type of boundary condition.

```plaintext
class BoundaryPeriodic(grid: GridBase, axis: int)
```

**Bases:** `pde.grids.boundaries.axis.BoundaryAxisBase`

represent a periodic axis

**Parameters**
- **grid** (*GridBase*) – The grid for which the boundary conditions are defined
- **axis** (*int*) – The axis to which this boundary condition is associated

```plaintext
check_value_rank(rank: int)
```

 check whether the values at the boundaries have the correct rank

**Parameters**
- **rank** (*int*) – The rank of the value that is stored with this boundary condition

```plaintext
copy()
```

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

**Type** `BoundaryPeriodic`

```plaintext
extract_component(*indices)
```

 extracts the boundary pair of the given extract_component.

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

```plaintext
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`

---

4.2. `pde.grids` package 85
get_point_evaluator(fill: float = None) → Callable[[numpy.ndarray, Tuple[int, ...]], NumberOrArray]
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 – This argument is ignored.

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

make_region_evaluator() → Callable[[numpy.ndarray, Tuple[int, ...]], Tuple[NumberOrArray, NumberOrArray, NumberOrArray]]
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, Callable]
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

periodic = True

generate_axis(grid: GridBase, axis: int, data, rank: int = 0) → 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 value associated with the boundary conditions.

Returns The boundary condition for the axis

Return type BoundaryAxisBase

dpe 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
- **ExtrapolateBC**: Extrapolate boundary points linearly from the two points closest to the boundary

Derivatives are given in the direction of the outward normal vector, such that positive derivatives correspond to a function that increases across the boundary, which corresponds to an inwards flux. Conversely, negative derivatives are associated with effluxes.

```python
class BCBase:
    def __init__(self, grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0):
        self.axis = axis
        self.upper = upper
        self.rank = rank
        self.value = value

        self.check_value_rank(rank=rank)
```

**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 value associated with the boundary condition.
- **value** (float or str or `ndarray`) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

```python
axis_coord
value of the coordinate that defines this boundary condition
Type float
```

```python
check_value_rank(rank: int)
check whether the values at the boundaries have the correct rank
```

**Parameters**

- **rank** (`int`) – The rank of the value that is stored with this boundary condition

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

```python
copy(upper: Optional[bool] = None, rank: int = None, value: Union[float, numpy.ndarray, str] = None) -> BCBase
return a copy of itself, but with a reference to the same grid
```
differentiated

differentiated version of this boundary condition

Type `BCBase`

```python
extract_component(*indices)
```

extracts the boundary conditions for the given component

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

```python
classmethod from_data(grid: GridBase, axis: int, upper: bool, data: BoundaryData, rank: int = 0) -> 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 value associated with the boundary condition.

**Returns**
the instance created from the data

**Return type** `BCBase`

**Throws:** ValueError if `data` cannot be interpreted as a boundary condition

```python
classmethod from_dict(grid: GridBase, axis: int, upper: bool, data: dict, rank: int = 0) -> 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 value associated with the boundary condition.

```python
classmethod from_str(grid: GridBase, axis: int, upper: bool, condition: str, rank: int = 0, value=0, **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 value associated with the boundary condition.
- value (float or str or array) – Sets the associated value
- **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_virtual_point(arr, idx: Tuple[int, ...] = None) → float
   homogeneous = None
doens whether the boundary condition depends on space
      Type bool

link_value(value: numpy.ndarray)
   link value of this boundary condition to external array

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

make_virtual_point_evaluator() → Callable[[numpy.ndarray, Tuple[int, ...]], Union[float, numpy.ndarray]]

names = None
   identifiers used to specify the given boundary class
      Type list

value_is_linked = None
   flag that indicates whether the value associated with this boundary condition is linked to ndarray managed by external code.
      Type bool

class BCBase1stOrder(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0)
   Bases: pde.grids.boundaries.local.BCBase
   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 value associated with the boundary condition.
• **value** (float or str or *ndarray*) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

```python
def 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
```

```python
def get_virtual_point(arr, idx: 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
```

```python
def get_virtual_point_data(compiled: bool = False) -> Tuple[Any, Any, int]
```

```python
def make_adjacent_evaluator() -> Callable[[numpy.ndarray, int, Tuple[int, ...]], Union[float, numpy.ndarray]]:
    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
```

```python
def make_virtual_point_evaluator() -> Callable[[numpy.ndarray, Tuple[int, ...]], Union[float, numpy.ndarray]]:
    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
```

```python
class BCBase2ndOrder(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0):
```

```python
Bases: pde.grids.boundaries.local.BCBase
```
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 value associated with the boundary condition.
- **value** (float or str or `ndarray`) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

**`get_data(idx: Tuple[int, ...])`**

- **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: Tuple[int, ...] = None)`**

- **calculate the value of the virtual point outside the boundary**

  **Parameters**

  - **arr** (*array*) – The data values associated with the grid
  - **idx** (*tuple*) – The index of the point to evaluate. This is a tuple of length `grid.num_axes` with the either -1 or `dim` as the entry for the axis associated with this boundary condition. Here, `dim` is the dimension of the axis. The index is optional if `dim == 1`.

  **Returns**

  Value at the virtual support point

  **Return type**

  float

**`get_virtual_point_data()`**

- **return data suitable for calculating virtual points**

  **Returns**

  the data associated with this virtual point

  **Return type**

  tuple

**`make_adjacent_evaluator()`**

- **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[[numpy.ndarray, Tuple[int, ...]], Union[float, numpy.ndarray]]
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
class CurvatureBC(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0)
Bases: pde.grids.boundaries.local.BCBase2ndOrder
represents a boundary condition imposing the 2nd 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 value associated with the boundary condition.
- value (float or str or ndarray) – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

get_virtual_point_data() → Tuple[numpy.ndarray, numpy.ndarray, int, numpy.ndarray, int]
return data suitable for calculating virtual points

Returns the data structure associated with this virtual point

Return type tuple

names = ['curvature', 'second_derivative']
class DirichletBC(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0)
Bases: pde.grids.boundaries.local.BCBase1stOrder
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 value associated with the boundary condition.
- `value (float or str or ndarray)` – a value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If `value` is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

**differentiated**

differentiated version of this boundary condition

**Type** `BCBase`

**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`

**names** = ['value', 'dirichlet']

**class ExtrapolateBC**(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0)

**Bases:** `pde.grids.boundaries.local.BCBase2ndOrder`

represents a boundary condition that extrapolates the virtual point using two points close to the boundary

This imposes a vanishing second derivative.

**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

4.2. pde.grids package
• **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 value associated with the boundary condition.

• **value** *(float or str or ndarray)* – A value stored with the boundary condition. The interpretation of this value depends on the type of boundary condition. If value is a single value (or tensor in case of tensorial boundary conditions), the same value is applied to all points. Inhomogeneous boundary conditions are possible by supplying an expression as a string, which then may depend on the axes names of the respective grid.

```python
def get_virtual_point_data() -> Tuple[numpy.ndarray, numpy.ndarray, int, numpy.ndarray, int]
    return data suitable for calculating virtual points
```

**Returns**
the data structure associated with this virtual point

**Return type**
tuple

```python
names = ['extrapolate', 'extrapolation']
class MixedBC(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0, const: Union[float, numpy.ndarray, str] = 0)
```

Bases: `pde.grids.boundaries.local.BCBase1stOrder`

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, $\gamma$ quantifies the influence of the field and $\beta$ is the constant term. Note that $\gamma = 0$ corresponds to Dirichlet conditions imposing $\beta$ 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

```python
bc = {'mixed': VALUE}
bc = {'type': 'mixed', 'value': VALUE, 'const': CONST}
```

where $VALUE$ corresponds to $\gamma$ and $CONST$ to $\beta$.

**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 value associated with the boundary condition.

• **value** *(float or str or array)* – The parameter $\gamma$ 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 $\beta$ determining the constant term for the boundary condition. Supports the same input as value.
mixed BC

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

get_virtual_point_data(compile: 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

names = ['mixed', 'robin']

class NeumannBC(grid: GridBase, axis: int, upper: bool, rank: int = 0, value: Union[float, numpy.ndarray, str] = 0)

Bases: pde.grids.boundaries.local.BCBase1stOrder

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.
names = ['derivative', 'neumann']

### 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.

<table>
<thead>
<tr>
<th>Grid Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian</td>
<td>This module implements differential operators on Cartesian grids</td>
</tr>
<tr>
<td>cylindrical</td>
<td>This module implements differential operators on cylindrical grids</td>
</tr>
<tr>
<td>polar</td>
<td>This module implements differential operators on polar grids</td>
</tr>
<tr>
<td>spherical</td>
<td>This module implements differential operators on spherical grids</td>
</tr>
</tbody>
</table>

#### pde.grids.operators.cartesian module

This module implements differential operators on Cartesian grids

<table>
<thead>
<tr>
<th>Operator Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_laplace</td>
<td>make a laplace operator on a Cartesian grid</td>
</tr>
<tr>
<td>make_gradient</td>
<td>make a gradient operator on a Cartesian grid</td>
</tr>
<tr>
<td>make_divergence</td>
<td>make a divergence operator on a Cartesian grid</td>
</tr>
<tr>
<td>make_vector_gradient</td>
<td>make a vector gradient operator on a Cartesian grid</td>
</tr>
<tr>
<td>make_vector_laplace</td>
<td>make a vector Laplacian on a Cartesian grid</td>
</tr>
<tr>
<td>make_tensor_divergence</td>
<td>make a tensor divergence operator on a Cartesian grid</td>
</tr>
</tbody>
</table>

**make_laplace** *(bcs: Boundaries, method: str = 'auto')* $\rightarrow$ Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

make a laplace operator on a Cartesian grid

**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)* – Method used for calculating the laplace operator. If method='auto', a suitable method is chosen automatically.

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

**make_gradient** *(bcs: Boundaries, method: str = 'auto')* $\rightarrow$ Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

make a gradient operator on a Cartesian grid

**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) – Method used for calculating the gradient operator. If method='auto', a suitable method is chosen automatically.

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

```python
make_divergence(bcs: Boundaries, method: str = 'auto') → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

make a divergence operator on a Cartesian grid

**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)* – Method used for calculating the divergence operator. If method='auto', a suitable method is chosen automatically.

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

```python
make_vector_gradient(bcs: Boundaries, method: str = 'auto') → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

make a vector gradient operator on a Cartesian grid

**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)* – Method used for calculating the vector gradient operator. If method='auto', a suitable method is chosen automatically.

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

```python
make_vector_laplace(bcs: Boundaries, method: str = 'auto') → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

make a vector Laplacian on a Cartesian grid

**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)* – Method used for calculating the vector laplace operator. If method='auto', a suitable method is chosen automatically.

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

```python
make_tensor_divergence(bcs: Boundaries, method: str = 'auto') → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

make a tensor divergence operator on a Cartesian grid

**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)* – 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_poisson_solver(bcs: Boundaries, method: str = 'auto') -> Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

make a operator that solves Poisson’s equation

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)** – 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

pde.grids.operators.common module

Common functions that are used by many operators

make_general_poisson_solver(matrix, vector, method: str = 'auto') -> Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

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[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

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

pde.grids.operators.cylindrical module

This module implements differential operators on cylindrical grids

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_laplace</td>
<td>make a discretized laplace operator for a cylindrical grid</td>
</tr>
<tr>
<td>make_gradient</td>
<td>make a discretized gradient operator for a cylindrical grid</td>
</tr>
<tr>
<td>make_divergence</td>
<td>make a discretized divergence operator for a cylindrical grid</td>
</tr>
</tbody>
</table>

Continued on next page
Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make_vector_gradient</code></td>
<td>make a discretized vector gradient operator for a cylindrical grid</td>
</tr>
<tr>
<td><code>make_vector_laplace</code></td>
<td>make a discretized vector laplace operator for a cylindrical grid</td>
</tr>
<tr>
<td><code>make_tensor_divergence</code></td>
<td>make a discretized tensor divergence operator for a cylindrical grid</td>
</tr>
</tbody>
</table>

```python
make_divergence(bcs: Boundaries) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
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**

- **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()`.

**Returns**

A function that can be applied to an array of values

```python
make_gradient(bcs: Boundaries) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
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**

- **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()`.

**Returns**

A function that can be applied to an array of values

```python
make_gradient_squared(bcs: Boundaries, central: bool = True) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
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**

- **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()`.

- **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

```python
make_laplace(bcs: Boundaries) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
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** `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()`.

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

```python
make_tensor_divergence(bcs: Boundaries) -> Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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** `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()`.

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

```python
make_vector_gradient(bcs: Boundaries) -> Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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** `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()`.

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

```python
make_vector_laplace(bcs: Boundaries) -> Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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** `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()`.

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

```python
pde.grids.operators.polar module
```

This module implements differential operators on polar grids

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_laplace</td>
<td>make a discretized laplace operator for a polar grid</td>
</tr>
<tr>
<td>make_gradient</td>
<td>make a discretized gradient operator for a polar grid</td>
</tr>
<tr>
<td>make_divergence</td>
<td>make a discretized divergence operator for a polar grid</td>
</tr>
<tr>
<td>make_vector_gradient</td>
<td>make a discretized vector gradient operator for a polar grid</td>
</tr>
</tbody>
</table>

Continued on next page
### make_tensor_divergence

- **make_tensor_divergence**
  - **Description:** Make a discretized tensor divergence operator for a polar grid.

### make_divergence

- **make_divergence**
  - **Description:** Make a discretized divergence operator for a polar grid.
  - **Parameters**
    - `bcs`: Specifies the boundary conditions applied to the field.
  - **Returns**
    - A function that can be applied to an array of values.

### make_gradient

- **make_gradient**
  - **Description:** Make a discretized gradient operator for a polar grid.
  - **Parameters**
    - `bcs`: Specifies the boundary conditions applied to the field.
  - **Returns**
    - A function that can be applied to an array of values.

### make_gradient_squared

- **make_gradient_squared**
  - **Description:** Make a discretized gradient squared operator for a polar grid.
  - **Parameters**
    - `bcs`: Specifies the boundary conditions applied to the field.
    - `central`: Whether a central difference approximation is used for the gradient.
  - **Returns**
    - A function that can be applied to an array of values.

### make_laplace

- **make_laplace**
  - **Description:** Make a discretized laplace operator for a polar grid.
  - **Parameters**
    - `bcs`: Specifies the boundary conditions applied to the field.
  - **Returns**
    - A function that can be applied to an array of values.
**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()`.

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

```python
make_poisson_solver(bcs: Boundaries, method: str = 'auto') → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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 $r_i = r_{\text{min}} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\text{min}}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\text{max}} = r_{\text{min}} + N_r\Delta r$.

**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()`.

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

```python
make_tensor_divergence(bcs: Boundaries) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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 $r_i = r_{\text{min}} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\text{min}}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\text{max}} = r_{\text{min}} + N_r\Delta r$.

**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()`.

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

```python
make_vector_gradient(bcs: Boundaries) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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 $r_i = r_{\text{min}} + (i + \frac{1}{2})\Delta r$ for $i = 0, \ldots, N_r - 1$, where $r_{\text{min}}$ is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by $r_{\text{max}} = r_{\text{min}} + N_r\Delta r$.

**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()`.

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

```python
make_laplace
```

make a discretized laplace operator for a spherical grid

```python
make_gradient
```

make a discretized gradient operator for a spherical grid

Continued on next page
make_divergence(bcs: Boundaries) \to \text{Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]}
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 \( r_i = r_{\text{min}} + (i + \frac{1}{2}) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()`.

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

make_gradient(bcs: Boundaries) \to \text{Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]}
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 \( r_i = r_{\text{min}} + (i + \frac{1}{2}) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()`.

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

make_gradient_squared(bcs: Boundaries, central: bool = True) \to \text{Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]}
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 \( r_i = r_{\text{min}} + (i + \frac{1}{2}) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()`.
- **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(bcs: Boundaries, conservative: bool = True) \to \text{Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]}
make a discretized laplace operator for a spherical grid

4.2. pde.grids package
The spherical grid assumes spherical symmetry, so that fields only depend on the radial coordinate \( r \). The radial discretization is defined as \( r_i = r_{\text{min}} + \left( i + \frac{1}{2} \right) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()*.

- **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

```python
make_poisson_solver(bcs: Boundaries, method: str = 'auto') \rightarrow Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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 \( r_i = r_{\text{min}} + \left( i + \frac{1}{2} \right) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()*.

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

```python
make_tensor_divergence(bcs: Boundaries) \rightarrow Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

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 \( r_i = r_{\text{min}} + \left( i + \frac{1}{2} \right) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()*.

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

```python
make_vector_gradient(bcs: Boundaries) \rightarrow Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]
```

make a discretized vector 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 \( r_i = r_{\text{min}} + \left( i + \frac{1}{2} \right) \Delta r \) for \( i = 0, \ldots, N_r - 1 \), where \( r_{\text{min}} \) is the radius of the inner boundary, which is zero by default. Note that the radius of the outer boundary is given by \( r_{\text{max}} = r_{\text{min}} + N_r \Delta r \).

**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()*.

**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_bounds
      lower and upper bounds of each axis
      Type tuple

   axes_coords
      coordinates of the cells for each axis
      Type tuple

   axes_symmetric = []
      The names of the additional axes that the fields do not depend on, e.g. along which they are constant.
      Type list

   cell_coords
      the coordinates of each cell
      Type ndarray

   cell_to_point(cells: numpy.ndarray, cartesian: bool = True) -> numpy.ndarray

   cell_volumes
      volume of each cell
      Type ndarray

   compatible_with(other: GridBase) -> bool
      tests whether this class 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(point: numpy.ndarray) → numpy.ndarray

copy() → GridBase
    return a copy of the grid

difference_vector_real(p1: numpy.ndarray, p2: numpy.ndarray)

discretization
    the linear size of a cell along each axis

    Type numpy.array

distance_real(p1: numpy.ndarray, p2: numpy.ndarray) → float
    Calculate the distance between two points given in real coordinates
    This takes periodic boundary conditions into account if need be

    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]) → 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_boundary_conditions(bc: BoundariesData = 'natural', rank: int = 0) → Boundaries

get_image_data(data: numpy.ndarray) → dict

get_line_data(data: numpy.ndarray, extract: str = 'auto') → dict

get_operator(name: str, bc: Boundaries, **kwargs) → Callable[[numpy.ndarray,
    numpy.ndarray], numpy.ndarray]
    return a discretized operator defined on this grid

    Parameters
    • name (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’). For non-periodic axes, different boundary conditions can be spec-
    ified for the lower and upper end (by 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 spec-
    ial value ‘natural’ imposes periodic boundary conditions for periodic axis and a
    vanishing derivative otherwise. More information can be found in the boundaries
    documentation.
• **kwargs – Specifies extra arguments that can influence how the operator is created. Many operators support a method argument that can typically be set to 'numba', 'scipy', or auto.

**Returns** A function that takes the discretized data as an input and returns the data to which the operator name has been applied. This function optionally supports a second argument, which provides allocated memory for the output.

```python
get_random_point(boundary_distance: float = 0, cartesian: bool = True) → numpy.ndarray
```

```python
get_subgrid(indices: Sequence[int]) → GridBase
```

return a subgrid of only the specified axes

```python
integrate(data: NumberOrArray, axes: Union[int, Sequence[int]] = None) → numpy.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*

```python
iter_mirror_points(point: numpy.ndarray, with_self: bool = False, only_periodic: bool = True) → Generator[T_co, T_contra, V_co]
```

```python
make_add_interpolated_compiled() → Callable[[numpy.ndarray, numpy.ndarray, NumberOrArray], None]
```

deprecated alias of method *make_inserter_compiled*

```python
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*

```python
make_inserter_compiled() → Callable[[numpy.ndarray, numpy.ndarray, NumberOrArray], None]
```

return a compiled function to insert values at interpolated positions

**Returns** A function with signature (data, position, amount), where data is the numpy array containing the field data, position denotes the position in grid coordinates, and amount is the that is to be added to the field.

```python
make_integrator() → Callable[[numpy.ndarray], numpy.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_interpolator_compiled(bc: BoundariesData = 'natural', rank: int = 0, fill: Number = None) → Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]

return a compiled function for linear interpolation on the grid

This interpolator respects boundary conditions and can thus interpolate values in the whole grid volume. However, close to corners, the interpolation might not be optimal, in particular for periodic grids.

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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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, optional) – The tensorial rank of the value associated with the boundary condition.

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

Returns A function which returns interpolated values when called with arbitrary positions within the space of the grid. The signature of this function is (data, point), where data is the numpy array containing the field data and position is denotes the position in grid coordinates.

make_normalize_point_compiled(reflect: bool = True) → Callable[[numpy.ndarray], None]

return a compiled function that normalizes the points

Normalizing points is useful to respect periodic boundary conditions. Here, points are assumed to be specified by the physical values along the non-symmetric axes of the grid.

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

normalize_point(point: numpy.ndarray, reflect: bool = True) → numpy.ndarray

normalize coordinates by applying periodic boundary conditions

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`

**numba_type**
represents type of the grid data in numba signatures

**Type**: `str`

**operators** = {}
names of all operators defined for this grid

**Type**: `set`

**plot()**
visualize the grid

**point_from_cartesian(points: numpy.ndarray)** → `numpy.ndarray`

**point_to_cartesian(points: numpy.ndarray)** → `numpy.ndarray`

**point_to_cell(points: numpy.ndarray)** → `numpy.ndarray`

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

**classmethod register_operator(name: str, factory_func: Callable = None, rank_in: int = 0, rank_out: int = 0)**
register an operator for this grid

**Example**

The method can either be used directly:

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

or as a decorator for the factory function:

```python
@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

**shape**
the number of support points of each axis

**Type**: `tuple of int`

**state**
**state_serialized**
JSON-serialized version of the state of this grid

Type `str`

**typical_discretization**
the average side length of the cells

Type `float`

**uniform_cell_volumes**
returns True if all cell volumes are the same

Type `bool`

**volume**

**class Operator**
Bases: `tuple`
stores information about an operator

Create new instance of `Operator(factory, rank_in, rank_out)`

**factory**
Alias for field number 0

**rank_in**
Alias for field number 1

**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[numpy.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`
4.2.4 pde.grids.cartesian module

Cartesian grids of arbitrary dimension.

```python
class CartesianGrid:
    # Class definition...
```

The grids can be thought of as a collection of n-dimensional boxes, called cells, of equal length in each
dimension. The bounds then define the total volume covered by these cells, while the cell coordinates
give the location of the box centers. We index the boxes starting from 0 along each dimension. Consequently, the cell \(i - \frac{1}{2}\) corresponds to the left edge of the covered interval and the index \(i + \frac{1}{2}\)
corresponds to the right edge, when the dimension is covered by d boxes.

In particular, the discretization along dimension \(k\) is defined as

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

\[
\Delta x^{(k)} = \frac{x_{\text{max}}^{(k)} - x_{\text{min}}^{(k)}}{N^{(k)}}
\]

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

**Parameters**

- `bounds (list of tuple)` – Give the coordinate range for each axis. This should be
  a tuple of two number (lower and upper bound) for each axis. The length of `bounds`
  thus determines the grid dimension.

- `shape (list)` – The number of support points for each axis. The length of `shape`
  needs to match the grid dimension.

- `periodic (bool or list)` – Specifies which axes possess periodic boundary condi-
tions. 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.

**cell_to_point**

```python
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 coor-
dinates or grid coordinates. This does not have any effect for Cartesian coordinate
  systems, but the argument is retained to have a unified interface for all grids.

**Returns** The center points of the respective cells

**Return type** ndarray

**difference_vector_real**

```python
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

  **classmethod from_state(state: dict) → CartesianGrid**

  create a field from a stored state.

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

  **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

  **point_to_cell(points: numpy.ndarray) → numpy.ndarray**

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

  This function respects periodic boundary conditions, but it does not throw an error when coordinates lie outside the bcs (for non-periodic axes).

  **Parameters** points (ndarray) – Real coordinates

  **Returns** The indices of the respective cells

  **Return type** ndarray

  **state**

  the state of the grid

  **Type** dict

  **volume**

  total volume of the grid

  **Type** float

  **class CartesianGridBase(shape: Sequence[int], periodic: Union[Sequence[bool], bool] = False)**

  Bases: pde.grids.base.GridBase

  Base class for UnitGrid and CartesianGrid

  **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.

  **cell_volume_data**

  size associated with each cell

  **contains_point(point: numpy.ndarray) → numpy.ndarray**

  check whether the point is contained in the grid

  **Parameters** point (ndarray) – Coordinates of the point
from_polar_coordinates(distance: numpy.ndarray, angle: numpy.ndarray, origin: numpy.ndarray = None) → numpy.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

get_boundary_conditions(bc: BoundariesData = 'natural', rank: int = 0) → 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.

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: numpy.ndarray) → dict

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: numpy.ndarray, extract: str = 'auto') → dict

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, cartesian: bool = True) -> numpy.ndarray`

return a random point within the grid

**Parameters**

- **boundary_distance (float)** – The minimal distance this point needs to have from all boundaries.

- **cartesian (bool)** – Determines whether the point is returned in Cartesian coordinates or grid coordinates. This does not have any effect for Cartesian coordinate systems, but the argument is retained to have a unified interface for all grids.

**Returns** The coordinates of the point

**Return type** `ndarray`

`iter_mirror_points(point: numpy.ndarray, with_self: bool = False, only_periodic: bool = True) -> Generator[T_co, T_contra, V_co]`

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: dict = None, fig_style: dict = 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 figure. If the argument is set to show `matplotlib.pyplot.show()` will be called to show the plot, if the value is create, the figure will be created, but not shown, and 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. If `None`, a new figure with a single axes is created.

• `**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: numpy.ndarray) → numpy.ndarray
```

convert points given in Cartesian coordinates to this grid

Parameters

- **points** (ndarray) – Points in Cartesian coordinates.

Returns

- Points given in the coordinates of the grid

Return type

- ndarray

```
point_to_cartesian(points: numpy.ndarray) → numpy.ndarray
```

convert coordinates of a point to Cartesian coordinates

Parameters

- **points** (ndarray) – Points given in the coordinates of the grid

Returns

- The Cartesian coordinates of the point

Return type

- ndarray

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

```
class UnitGrid(shape: Sequence[int], periodic: Union[Sequence[bool], bool] = False)
Bases: pde.grids.cartesian.CartesianGridBase
```

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.

```
cell_to_point(cells: numpy.ndarray, cartesian: bool = True) → numpy.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. This does not have any effect for Cartesian coordinate systems, but the argument is retained to have a unified interface for all grids.

Returns The center points of the respective cells

Return type *ndarray*

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

return the vector pointing from p1 to p2.

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*

classmethod from_state(*state: dict*) → 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*

point_to_cell(*points: numpy.ndarray*) → numpy.ndarray

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

This function respects periodic boundary conditions, but it does not throw an error when coordinates lie outside the bcs (for non-periodic axes).

Parameters **points** (*ndarray*) – Real coordinates

Returns The indices of the respective cells

Return type *ndarray*

state

the state of the grid

Type *dict*

to_cartesian() → CartesianGrid

convert unit grid to CartesianGrid

volume
total volume of the grid
Type float

4.2.5 pde.grids.cylindrical module

Cylindrical grids with azimuthal symmetry

```python
class CylindricalGrid(radius: float, bounds_z: Tuple[float, float], shape: Tuple[int, int], periodic_z: bool = False)
    Bases: pde.grids.base.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

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

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

where \( R \) is the radius of the cylindrical grid, \( z_{\text{min}} \) and \( z_{\text{max}} \) denote the respective lower and upper bounds of the axial direction, so that \( z_{\text{max}} - z_{\text{min}} \) is the total height. The two axes are discretized by \( N_r \) and \( N_z \) support points, respectively.

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.
- **periodic_z** (bool) – Determines whether the z-axis has periodic boundary conditions.

```python
axes = ['r', 'z']
axes_symmetric = ['phi']
```

```python
cell_to_point(cells: numpy.ndarray, cartesian: bool = True) -> numpy.ndarray
```

convert cell coordinates to real coordinates

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

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

```python
cell_volume_data
```

the volumes of all cells

Type ndarray

```python
contains_point(point: numpy.ndarray) -> numpy.ndarray
```

check whether the point is contained in the grid

4.2. pde.grids package

117
Parameters **point** *(ndarray)* – Coordinates of the point

**coordinate_constraints** = [0, 1]

difference_vector_real *(p1: numpy.ndarray, p2: numpy.ndarray)* → numpy.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 = 3
classmethod **from_state**(state: *dict*) → CylindricalGrid

create a field from a stored state.

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

get_boundary_conditions *(bc: BoundariesData = 'natural', rank: int = 0)* → 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.

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: numpy.ndarray) → dict*

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: numpy.ndarray, extract: str = 'auto') → dict*

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, cartesian: bool = True, avoid_center: bool = False) → numpy.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.
- **cartesian** *(bool)* – Determines whether the point is returned in Cartesian coordinates or grid coordinates.
- **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.

**Returns**
The coordinates of the point

**Return type**
.ndarray

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

Return a subgrid of only the specified axes

**Parameters**
- **indices** *(list)* – Indices indicating the axes that are retained in the subgrid

**Returns**
CartesianGrid or PolarGrid: The subgrid

**iter_mirror_points** *(point: numpy.ndarray, with_self: bool = False, only_periodic: bool = True) → Generator[T_co, T_contra, V_co]*

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

length
length of the cylinder

Type float

num_axes = 2

point_from_cartesian(points: numpy.ndarray) → numpy.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: numpy.ndarray) → numpy.ndarray
convert coordinates of a point to Cartesian coordinates

Parameters points (ndarray) – Points given in the coordinates of the grid

Returns The Cartesian coordinates of the point

Return type ndarray

point_to_cell(points: numpy.ndarray) → numpy.ndarray
Determine cell(s) corresponding to given point(s)

This function respects periodic boundary conditions, but it does not throw an error when coordinates lie outside the bcs (for non-periodic axes).

Parameters points (ndarray) – Real coordinates

Returns The indices of the respective cells

Return type ndarray

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

radius
radius of the cylinder

Type float

state
the state of the grid

Type state
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.

**class PolarGrid**

```python
class PolarGrid(radius: Union[float, Tuple[float, float]], shape: Union[Tuple[int], int])
```

Bases: `pde.grids.spherical.SphericalGridBase`

2-dimensional polar grid assuming angular symmetry

The angular symmetry implies that states only depend on the radial coordinate \( r \), which is discretized uniformly as

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

where \( R_{\text{outer}} \) is the outer radius of the grid and \( R_{\text{inner}} \) 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 \( R_{\text{outer}} \) 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** = ['r']

**axes_symmetric** = ['phi']

**coordinate_constraints** = [0, 1]

**dim** = 2

**point_to_cartesian** *(points: numpy.ndarray) → numpy.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.

**Returns** The Cartesian coordinates of the point

**Return type** `ndarray`

**class SphericalGrid**

```python
class SphericalGrid(radius: Union[float, Tuple[float, float]], shape: Union[Tuple[int], int])
```

Bases: `pde.grids.spherical.SphericalGridBase`

3-dimensional spherical grid assuming spherical symmetry

The symmetry implies that states only depend on the radial coordinate \( r \), which is discretized as follows:

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

where \( R_{\text{outer}} \) is the outer radius of the grid and \( R_{\text{inner}} \) 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 $R_{\text{outer}}$ 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 = ['r']
axes_symmetric = ['theta', 'phi']
coordinate_constraints = [0, 1, 2]
dim = 3

point_to_cartesian(points: numpy.ndarray) → numpy.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) – Points given in the coordinates of the grid

Returns

The Cartesian coordinates of the point

Return type

ndarray

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

Bases: pde.grids.base.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 $R_{\text{outer}}$ is the outer radius of the grid and $R_{\text{inner}}$ 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 $R_{\text{outer}}$ 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

cell_to_point(cells: numpy.ndarray, cartesian: bool = True) → numpy.ndarray
convert cell coordinates to real coordinates

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

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

cell_volume_data
the volumes of all cells
contains_point(point: numpy.ndarray) → numpy.ndarray
check whether the point is contained in the grid

Parameters
point (ndarray) – Coordinates of the point

difference_vector_real(p1: numpy.ndarray, p2: numpy.ndarray) → numpy.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

classmethod from_state(state: dict) → SphericalGridBase
create a field from a stored state.

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

get_boundary_conditions(bc='natural', rank: int = 0) → 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.

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*

```python
get_image_data(data: numpy.ndarray, performance_goal: str = 'speed', fill_value: float = 0, masked: bool = True) → dict
```

**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.

```python
get_line_data(data: numpy.ndarray, extract: str = 'auto') → dict
```

**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.

```python
get_random_point(boundary_distance: float = 0, cartesian: bool = True, avoid_center: bool = False) → numpy.ndarray
```

**Parameters**

- **boundary_distance** *(float)* – The minimal distance this point needs to have from all boundaries.
- **cartesian** *(bool)* – Determines whether the point is returned in Cartesian coordinates or grid coordinates.
- **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.

**Returns** The coordinates of the point

**Return type** *ndarray*
has_hole
returns whether the inner radius is larger than zero

iter_mirror_points(point: numpy.ndarray, with_self: bool = False, only_periodic: bool = True) → Generator[T_co, T_contra, V_co]
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 = 1
periodic = [False]

plot(*args, title: str = None, filename: str = None, action: str = 'auto', ax_style: dict = None, fig_style: dict = 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 figure. If the argument is set to show matplotlib.pyplot.show() will be called to show the plot, if the value is create, the figure will be created, but not shown, and 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. If None, a new figure with a single axes is created.
- **kwargs – Extra arguments are passed on to the matplotlib plotting routines, e.g., to set the color of the lines

point_from_cartesian(points: numpy.ndarray) → numpy.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

point_to_cell(points: numpy.ndarray) → numpy.ndarray
Determine cell(s) corresponding to given point(s)

Parameters
points (ndarray) – Real coordinates
Returns
The indices of the respective cells

Return type ndarray

polar_coordinates_real(\text{origin}=\text{None}, *, \text{ret_angle}: \text{bool} = \text{False}, **\text{kwargs}) \rightarrow
Union[\text{numpy.ndarray}, \text{Tuple[\text{numpy.ndarray}, ...]}]

return spherical coordinates associated with the grid

Parameters

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

\text{radius}
radius of the sphere
Type float

\text{state}
the state of the grid
Type state

\text{volume}
total volume of the grid
Type float

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 \textit{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.

<table>
<thead>
<tr>
<th>PDE</th>
<th>PDE defined by mathematical expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllenCahnPDE</td>
<td>A simple Allen-Cahn equation</td>
</tr>
<tr>
<td>CahnHilliardPDE</td>
<td>A simple Cahn-Hilliard equation</td>
</tr>
<tr>
<td>DiffusionPDE</td>
<td>A simple diffusion equation</td>
</tr>
<tr>
<td>KPZInterfacePDE</td>
<td>The Kardar-Parisi-Zhang (KPZ) equation</td>
</tr>
<tr>
<td>KuramotoSivashinskyPDE</td>
<td>The Kuramoto-Sivashinsky equation</td>
</tr>
<tr>
<td>SwiftHohenbergPDE</td>
<td>The Swift-Hohenberg equation</td>
</tr>
<tr>
<td>WavePDE</td>
<td>A simple wave equation</td>
</tr>
</tbody>
</table>

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: BoundariesData = 'natural')
    Bases: pde.pdes.base.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 \( \gamma \) 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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) \rightarrow 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 = False

    expression
    the expression of the right hand side of this PDE

    Type str

4.3.2 pde.pdes.base module

Base classes

class PDEBase(noise: ArrayLike = 0)
    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.

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.

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

Type bool

complex_valued = 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

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

explicit_time_dependence = None
Flag indicating whether the right hand side of the PDE has an explicit time dependence.

Type bool

is_sde
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[[numpy.ndarray, float], 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[[numpy.ndarray, float], numpy.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 ‘python’ and ‘numba’. Alternatively, ‘auto’ lets the code decide for the most optimal backend.

Returns Function determining the right hand side of the PDE
make_sde_rhs(state: FieldBase, backend: str = 'auto') → Callable[[numpy.ndarray, float], Tuple[numpy.ndarray, numpy.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: TrackerData = ['progress', 'consistency'], method: str = 'auto', ret_info: bool = False, **kwargs)

Union[FieldBase, Tuple[FieldBase, dict]]

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 grid)
- **t_range** (float or tuple) – Sets the time range for which the PDE is solved. If only a single value t_end is given, 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.
- **tracker** – Defines a tracker that process the state of the simulation at fixed time intervals. Multiple trackers can be specified as a list. The default value is ['progress', 'consistency'], which displays a progress bar and checks the state for consistency, aborting the simulation when not-a-number values appear.
- **method** (SolverBase or str) – Specifies a 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().
- **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
Returns The state at the final time point. In the case 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: BoundariesData = 'natural', bc_mu: BoundariesData = 'natural')
Bases: pde.pdes.base.PDEBase

A simple Cahn-Hilliard equation

The mathematical definition is

$$\partial_t c = \nabla^2 (c^3 - c - \gamma \nabla^2 c)$$

where $c$ is a scalar field taking values on the interval $[-1, 1]$ and $\gamma$ sets the interfacial width.

Parameters

- **interface_width** (float) – The diffusivity of the described species
- **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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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 = False

expression
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: BoundariesData = 'natural')
  Bases: pde.pdes.base.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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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 = False

expression
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: BoundariesData = 'natural')
Bases: pde.pdes.base.PDEBase

The Kardar–Parisi–Zhang (KPZ) equation

The mathematical definition is

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

where $h$ is the height of the interface in Monge parameterization. The dynamics are governed by the two parameters $\nu$ and $\lambda$, while $\eta$ is Gaussian white noise, whose strength is controlled by the $\text{noise}$ argument.

Parameters

- **nu** (float) – Parameter $\nu$ for the strength of the diffusive term
- **lmbda** (float) – Parameter $\lambda$ for the strength 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'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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.

\text{evolution\_rate}(\text{state: ScalarField}, \text{t: float = 0}) \rightarrow \text{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

\text{expression}

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: BoundariesData = 'natural',
bc_lap: BoundariesData = None)

Bases: pde.pdes.base.PDEBase

The Kuramoto-Sivashinsky equation

The mathematical definition is
\[
\partial_t u = -\nu \nabla^4 u - \nabla^2 u - \frac{1}{2} (\nabla h)^2 + \eta(r, t)
\]

where \( u \) is the height of the interface in Monge parameterization. The dynamics are governed by the parameters \( \nu \), while \( \eta \) is Gaussian white noise, whose strength is controlled by the \( \text{noise} \) argument.

Parameters

- **nu (float)** – Parameter \( \nu \) 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 axes, only periodic boundary conditions are allowed (indicated by 'periodic'). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by a tuple of two conditions). For instance, Dirichlet conditions enforcing a value \( \text{NUM} \) (specified by \{‘value’: \text{NUM}\}) and Neumann conditions enforcing the value \( \text{DERIV} \) for the derivative in the normal direction (specified by \{‘derivative’: \text{DERIV}\}) are supported. Note that the special value ‘natural’ imposes periodic boundary conditions for periodic axes 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 \text{None}, the same boundary condition as \( bc \) is chosen. Otherwise, this supports the same options as \( bc \).

evolution_rate(state: ScalarField, t: float = 0) \rightarrow 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 = False

expression

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: BoundariesData, label: str = "Solution to Laplace’s equation") \rightarrow 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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: BoundariesData, label: str = ”Solution to Poisson’s equation”) → ScalarField

Solve Laplace’s equation on a given grid

Denoting the current field by $u$, we thus solve for $f$, defined by the equation

$$\nabla^2 u(r) = -f(r)$$

with boundary conditions specified by bc.

**Note:** In case of periodic or Neumann boundary conditions, the right hand side $f(r)$ needs to satisfy the following condition

$$\int f \, dV = \int g \, dS,$$

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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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

```python
class PDE(rhs: OrderedDict[str, str], noise: ArrayLike = 0, bc: BoundariesData = 'natural', bc_ops: OrderedDict[str, BoundariesData] = None, user_funcs: dict = None, consts: dict = None)
```

**Bases:** `pde.pdes.base.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** (`OrderedDict`) – 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 expressions 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, while an outer product is calculated using `outer(field1, field2)`.

- **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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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 \( \text{rhs} \) 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 \( \text{rhs} \).

- **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 \( \text{rhs} \) defines the order in which they need to appear in the \( \text{state} \) 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.

```python
evolution_rate(state: FieldBase, t: float = 0) \rightarrow FieldBase
```

**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

**expressions**
- show the expressions of the PDE

### 4.3.9 `pde.pdes.swift_hohenberg` module

The Swift-Hohenberg equation

```python
class SwiftHohenbergPDE(rate: float = 0.1, kc2: float = 1.0, delta: float = 1.0, bc: BoundariesData = 'natural', bc_lap: BoundariesData = None)
```

**Bases:** `pde.pdes.base.PDEBase`

The Swift-Hohenberg equation

The mathematical definition is

\[
\partial_t c = \left[ \epsilon - (k_c^2 + \nabla^2)^2 \right] c + \delta c^2 - c^3
\]

where \( c \) is a scalar field and \( \epsilon, k_c^2, \) and \( \delta \) are parameters of the equation.

**Parameters**
- **rate** (*float*) – The bifurcation parameter \( \epsilon \)
- **kc2** (*float*) – Squared wave vector \( k_c^2 \) of the linear instability
- **delta** (*float*) – Parameter \( \delta \) 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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by 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*) \( \rightarrow \) 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** = False

**expression**

    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: BoundariesData = ‘natural’)

    Bases: pde.pdes.base.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’). For non-periodic axes, different boundary conditions can be specified for the lower and upper end (by a tuple of two conditions). For instance, Dirichlet conditions enforcing a value NUM (specified by \{\text{'value': NUM}\}) and Neumann conditions enforcing the value DERIV for the derivative in the normal direction (specified by \{\text{'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.

\[\text{evolution_rate(state: FieldCollection, t: float = 0) } \rightarrow \text{FieldCollection}\]

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

\text{explicit_time_dependence } = \text{False}

expressions

the expressions of the right hand side of this PDE

Type dict

\[\text{get_initial_condition(u: ScalarField, v: ScalarField = None)}\]

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., advanced in time.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>class controlling a simulation</td>
</tr>
<tr>
<td>ExplicitSolver</td>
<td>class for solving partial differential equations explicitly</td>
</tr>
<tr>
<td>ImplicitSolver</td>
<td>class for solving partial differential equations implicitly</td>
</tr>
<tr>
<td>ScipySolver</td>
<td>class for solving partial differential equations using scipy</td>
</tr>
<tr>
<td>registered_solvers</td>
<td>returns all solvers that are registered in the package</td>
</tr>
</tbody>
</table>

**class Controller**

\[\text{controller: SolverBase, t_range: Union[float, Tuple[float, float]], tracker: Tracker-
\text{Data } = \{\text{'progress', 'consistency'}\}\]

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 trackers that process the state of the simulation at fixed time intervals. Multiple trackers can be specified as a list. The default value ‘auto’ is converted to `[‘progress’, ‘consistency’]` for normal simulations. This thus displays a progress bar and checks the state for consistency, aborting the simulation when not-a-number values appear. To disable trackers, set the value to `None`.

```python
def 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.
```

```python
class ExplicitSolver(pde: PDEBase, scheme: str = ‘euler’, backend: str = ‘auto’)
    Bases: pde.solvers.base.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’, ‘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.

```python
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 `make_stepper(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: pde.solvers.base.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: pde.solvers.base.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 \( t_{\text{start}} \) to time \( t_{\text{end}} \). The function call signature is \((\text{state}: \text{numpy.ndarray}, t_{\text{start}}: \text{float}, t_{\text{end}}: \text{float})\)

\[
\text{name} = '\text{scipy}'
\]

\[\text{registered_solvers}() \rightarrow \text{List}[\text{str}]\]
returns all solvers that are registered in the package

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

```python
class SolverBase(pde: \text{PDEBase})
```

Bases: \text{object}

base class for simulations

**Parameters**

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

```python
classmethod from_name(name: \text{str}, pde: \text{PDEBase}, **kwargs) \rightarrow \text{SolverBase}
```

create solver class based on its name

Solver classes are automatically registered when they inherit from \text{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 \(\text{An instance of a subclass of } \text{SolverBase}\)

```python
\text{make_stepper}(\text{state}, dt: \text{float} = \text{None}) \rightarrow \text{Callable}[[\text{FieldBase}, \text{float}, \text{float}], \text{float}]
```

```python
\text{registered_solvers} = ['ExplicitSolver', 'ImplicitSolver', 'ScipySolver', 'explicit', 'implicit', 'scipy']
```

### 4.4.2 pde.solvers.controller module

Defines the \text{Controller} class for solving pdes.

```python
class Controller(solver: \text{SolverBase}, t_range: \text{Union}[\text{float}, \text{Tuple}[\text{float}, \text{float}]], tracker: \text{Tracker}-\text{Data} = ['\text{progress}', '\text{consistency}'])
```

Bases: \text{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 \(t_{\text{end}}\) is given, the time range is assumed to be \([0, t_{\text{end}}]\).
• **tracker** – Defines trackers that process the state of the simulation at fixed time intervals. Multiple trackers can be specified as a list. The default value ‘auto’ is converted to ‘[‘progress’, ‘consistency’]’ for normal simulations. This thus displays a progress bar and checks the state for consistency, aborting the simulation when not-a-number values appear. To disable trackers, set the value to **None**.

```python
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.

**t_range**

### 4.4.3 `pde.solvers.explicit` module

Defines an explicit solver supporting various methods

```python
class ExplicitSolver(pde: PDEBase, scheme: str = 'euler', backend: str = 'auto')
```

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’, ‘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.

```python
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)

```python
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: pde.solvers.base.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 $t_{\text{start}}$ to time $t_{\text{end}}$. 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: pde.solvers.base.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 \(t_{\text{start}}\) to time \(t_{\text{end}}\). The function call signature is \((\text{state: numpy.ndarray, } t_{\text{start}}: \text{float, } t_{\text{end}}: \text{float})\)

name = 'scipy'

4.5 pde.storage package

Module defining classes for storing simulation data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_memory_storage</td>
<td>a context manager that can be used to create a MemoryStorage</td>
</tr>
<tr>
<td>MemoryStorage</td>
<td>store discretized fields in memory</td>
</tr>
<tr>
<td>FileStorage</td>
<td>store discretized fields in a hdf5 file</td>
</tr>
</tbody>
</table>

4.5.1 pde.storage.base module

Base classes for storing data

class StorageBase(info: dict = None, write_mode: str = 'truncate_once')

Bases: object

base class for storing time series of discretized fields

These classes store time series of subclasses 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\((\text{func: Callable, out: Optional[StorageBase]} = \text{None, *, progress: bool} = \text{False}) \rightarrow \text{StorageBase}\)

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 \text{func} has been applied

Return type \text{StorageBase}

clear\(\text{(clear\_data\_shape: bool} = \text{False}) \rightarrow \text{None}\)

truncate the storage by removing all stored data.

**Parameters**

- **clear\_data\_shape** *(bool)* – Flag determining whether the data shape is also deleted.

copy\((\text{out: Optional[StorageBase]} = \text{None, *, progress: bool} = \text{False}) \rightarrow \text{StorageBase}\)

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 \text{StorageBase}

data\_shape

the current data shape.

**Raises** \text{RuntimeError} – if data\_shape was not set

end\_writing() \rightarrow \text{None}

finalize the storage after writing

extract\_field(\text{field\_id: Union[int, str], label: str} = \text{None}) \rightarrow \text{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

grid

the grid associated with this storage

This returns None if grid was not stored in self.info.

Type GridBase

has_collection

whether the storage is storing a collection

Type bool

items() → Iterator[Tuple[float, FieldBase]]

iterate over all times and stored fields, returning pairs

shape

the shape of the stored data

start_writing(field: FieldBase, info: dict = 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

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 StorageTracker The tracker that fills the current storage

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

Bases: pde.trackers.base.TrackerBase

Tracker that stores data in special storage classes
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: dict = None`) → None  
finalizes 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: dict = 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: dict = None, write_mode: str = ’truncate_once’, max_length: int = None, compression: bool = True, keep_opened: bool = True`)  

**Bases:** `pde.storage.base.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.

```python
clear(clear_data_shape: bool = False)
```

truncates the storage by removing all stored data.

**Parameters**

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

```python
close() \rightarrow None
```

close the currently opened file

```python
data
```

The actual data for all time

Type: ndarray

```python
end_writing() \rightarrow None
```

finalize the storage after writing.

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

```python
start_writing(field: FieldBase, info: dict = None) \rightarrow 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

```python	
times
```

The times at which data is available

Type: ndarray

### 4.5.3 pde.storage.memory module

Defines a class storing data in memory.

```python
class MemoryStorage(times: Optional[Sequence[float]] = None, data: Optional[List[numpy.ndarray]] = None, field_obj: FieldBase = None, info: dict = None, write_mode: str = 'truncate_once')
```

Bases: `pde.storage.base.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 that can 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.

```python
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.

```python
classmethod from_collection(storages: Sequence[StorageBase], label: str = None) → 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

Returns

Storage containing all the data.

Return type

`MemoryStorage`

```python
classmethod from_fields(times: Optional[Sequence[float]] = None, fields: Optional[Sequence[FieldBase]] = None, info: dict = 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.

```python
start_writing(field: FieldBase, info: dict = 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
get_memory_storage(field: FieldBase, info: dict = None)
a context manager that can be used to create a MemoryStorage

Example

This can be used to quickly store data:

```python
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 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

Yields **MemoryStorage**

### 4.6 pde.tools package

Package containing several tools required in py-pde

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache</td>
<td>Functions, classes, and decorators for managing caches</td>
</tr>
<tr>
<td>config</td>
<td>Handles configuration variables of the package</td>
</tr>
<tr>
<td>cuboid</td>
<td>An n-dimensional, axes-aligned cuboid</td>
</tr>
<tr>
<td>docstrings</td>
<td>Methods for automatic transformation of docstrings</td>
</tr>
<tr>
<td>expressions</td>
<td>Handling mathematical expressions with sympy</td>
</tr>
<tr>
<td>math</td>
<td>Auxiliary mathematical functions</td>
</tr>
<tr>
<td>misc</td>
<td>Miscellaneous python functions</td>
</tr>
<tr>
<td>numba</td>
<td>Helper functions for just-in-time compilation with numba</td>
</tr>
<tr>
<td>output</td>
<td>Python functions for handling output</td>
</tr>
<tr>
<td>parameters</td>
<td>Infrastructure for managing classes with parameters</td>
</tr>
<tr>
<td>parse_duration</td>
<td>Parsing time durations from strings</td>
</tr>
<tr>
<td>plotting</td>
<td>Tools for plotting and controlling plot output using context managers</td>
</tr>
<tr>
<td>spectral</td>
<td>Functions making use of spectral decompositions</td>
</tr>
<tr>
<td>spherical</td>
<td>Module collecting functions for handling spherical geometry</td>
</tr>
<tr>
<td>typing</td>
<td>Provides support for mypy type checking of the package</td>
</tr>
</tbody>
</table>

#### 4.6.1 pde.tools.cache module

Functions, classes, and decorators for managing caches
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cached_property</td>
<td>Decorator to use a method as a cached property</td>
</tr>
<tr>
<td>cached_method</td>
<td>Decorator to enable caching of a method</td>
</tr>
<tr>
<td>hash_mutable</td>
<td>return hash also for (nested) mutable objects.</td>
</tr>
<tr>
<td>hash_readable</td>
<td>return human readable hash also for (nested) mutable objects.</td>
</tr>
<tr>
<td>make_serializer</td>
<td>returns a function that serialize data with the given method.</td>
</tr>
<tr>
<td>make_unserializer</td>
<td>returns a function that unserialize data with the given method</td>
</tr>
<tr>
<td>DictFiniteCapacity</td>
<td>cache with a limited number of items</td>
</tr>
<tr>
<td>SerializedDict</td>
<td>a key value database which is stored on the disk</td>
</tr>
</tbody>
</table>

```python
class DictFiniteCapacity(*args, **kwargs):
    Bases: collections.OrderedDict
    cache with a limited number of items

    check_length()
        ensures that the dictionary does not grow beyond its capacity

    default_capacity = 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: dict = None):
    Bases: collections.abc.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()
```

```python
class cached_method(factory=None, extra_args=None, ignore_args=None, hash_function='hash_mutable', doc=None, name=None):
    Bases: pde.tools.cache._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.
```

4.6. pde.tools package 151
Example

The decorator can be used like so:

```python
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:

```python
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:
```python
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

```python
class cached_property(factory=None, extra_args=None, ignore_args=None,
                       hash_function='hash_mutable', doc=None, name=None)
Bases: pde.tools.cache._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:

```python
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:

```python
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:

```python
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

\[\text{hash\_mutable}(\text{obj}) \rightarrow \text{int}\]

\[\text{return hash also for (nested) mutable objects.}\]

**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`

\[\text{hash\_readable}(\text{obj}) \rightarrow \text{str}\]

\[\text{return human readable hash also for (nested) mutable objects.}\]

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`

\[\text{make\_serializer}(\text{method: str}) \rightarrow \text{Callable}\]

\[\text{return a function that serialize data with the given method. Note that some of the methods destroy information and cannot be reverted.}\]

**Parameters** `method` *(str)* – An identifier determining the serializer that will be returned

**Returns** A function that serializes objects

**Return type** `callable`

\[\text{make\_unserializer}(\text{method: str}) \rightarrow \text{Callable}\]

\[\text{return a function that unserialize data with the given method}\]

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

```python
class Config(items: dict = None, mode: str = 'update')
    Bases: collections.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`

    convert the configuration to a simple dictionary

    Returns A representation of the configuration in a normal dict.

    Return type dict

class ParameterModuleConstant(name: str, module_path: str, variable: str, description: str = '')
    Bases: object

    special parameter class to access module constants as configuration values

    Parameters

    - **name** *(str)* – The name of the parameter
    - **module_path** *(str)* – The path of the module in which the constants is defined
    - **variable** – The name of the constant
get()
    obtain the value of the constant

environment(dict_type=<class 'dict'>) → dict
    obtain information about the compute environment

    Parameters  dict_type – The type to create the returned dictionaries. The default is dict,
                        but collections.OrderedDict is an alternative.

    Returns  information about the python installation and packages

    Return type  dict

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

    bounds

    buffer(amount: Union[float, numpy.ndarray] = 0, inplace=False) → Cuboid
        dilate the cuboid by a certain amount in all directions

    centroid

    contains_point(points: numpy.ndarray) → numpy.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

    corners
        return coordinates of two extreme corners defining the cuboid

    diagonal
        returns the length of the diagonal

    dim

    classmethod from_bounds(bounds: numpy.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: numpy.ndarray, size: numpy.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: numpy.ndarray, p2: numpy.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

mutable

size

surface_area

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

vertices

return the coordinates of all the corners

volume

asanyarray_flags(data: numpy.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

```python
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.

```mermaid
const parse_number = return a number compiled from an expression
class ScalarExpression
    describes a mathematical expression of a scalar quantity

class TensorExpression
    describes a mathematical expression of a tensorial quantity

const class ExpressionBase =
    expression: basic.Basic, signature: Sequence[Union[str, List[str]]] = None,
    user_funcs: dict = None, consts: dict = 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.
```

```python
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

**complex**

whether the expression contains the imaginary unit I

*Type* bool

**constant**

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

**expression**

the expression in string form

*Type* str

**getCompiled** *(single_arg: bool = False) -> Callable[[...], NumberOrArray]*

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

**rank**

the rank of the expression

*Type* int

**shape**

class ScalarExpression *(expression: Union[float, str, numpy.ndarray, ExpressionBase] = 0, signature: Optional[Sequence[Union[str, List[str]]]] = None, user_funcs: dict = None, consts: dict = None, allow_indexed: bool = False)*

Bases: `pde.tools.expressions.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
- **allow_indexed** (*bool*) – Whether to allow indexing of variables. If enabled, array variables are allowed to be indexed using square bracket notation.

**derivatives**

differentiate the expression with respect to all variables

differentiate(*var*: *str*) \rightarrow ScalarExpression

return the expression differentiated with respect to var

**is_zero**

returns whether the expression is zero

Type *bool*

**shape** = (*

value

the value for a constant expression

Type *float*

**class TensorExpression**(*expression*: *Union*[float, str, numpy.ndarray, ExpressionBase], *signature*: *Optional*[Sequence[Union[str, List[str]]]] = *None*, *user_funcs*: *dict* = *None*)

Bases: *pde.tools.expressions.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

```python
def derivatives(tensordata):
    # differentiate the expression with respect to all variables

def differentiate(var: str) -> TensorExpression:
    return the expression differentiated with respect to var

def get_compiled_array(single_arg: bool = True) -> Callable[[numpy.ndarray, numpy.ndarray], numpy.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.

```python
def shape(tensordata):
    return the shape of the tensor

def value(tensordata):
    return the value for a constant expression
```

### 4.6.6 pde.tools.math module

Auxiliary mathematical functions

```python
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 using units of x. If it is not given, the average distance of x values multiplied by `sigma_auto_scale` is used.

    **bounds**
    return minimal and maximal x values

    **sigma_auto_scale** = 10
    scale for setting automatic values for sigma
```

**Type** float
### 4.6.7 pde.tools.misc module

Miscellaneous python functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>environment</td>
<td>obtain information about the compute environment</td>
</tr>
<tr>
<td>module_available</td>
<td>check whether a python module is available</td>
</tr>
<tr>
<td>ensure_directory_exists</td>
<td>creates a folder if it not already exists</td>
</tr>
<tr>
<td>preserve_scalars</td>
<td>decorator that makes vectorized methods work with scalars</td>
</tr>
<tr>
<td>decorator_arguments</td>
<td>make a decorator usable with and without arguments:</td>
</tr>
<tr>
<td>skipUnlessModule</td>
<td>decorator that skips a test when a module is not available</td>
</tr>
<tr>
<td>import_class</td>
<td>import a class or module given an identifier</td>
</tr>
<tr>
<td>classproperty</td>
<td>decorator that can be used to define read-only properties for classes.</td>
</tr>
<tr>
<td>hybridmethod</td>
<td>descriptor that can be used as a decorator to allow calling a method both as a classmethod and an instance method</td>
</tr>
<tr>
<td>estimate_computation_speed</td>
<td>estimates the computation speed of a function</td>
</tr>
<tr>
<td>hdf_write_attributes</td>
<td>write (JSON-serialized) attributes to a hdf file</td>
</tr>
<tr>
<td>number</td>
<td>convert a value into a float or complex number</td>
</tr>
<tr>
<td>get_common_dtype</td>
<td>returns a dtype in which all arguments can be represented</td>
</tr>
<tr>
<td>number_array</td>
<td>convert an array with arbitrary dtype either to np.double or np.cdouble</td>
</tr>
</tbody>
</table>

#### class classproperty(fget, 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:

```python
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](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, pathlib.Path]`)  
creates a folder if it not already exists

**Parameters**

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

**environment**( `dict_type=<class 'dict'>`) → dict

obtain information about the compute environment

**Parameters**

- `dict_type` – The type to create the returned dictionaries. The default is `dict`, but `collections.OrderedDict` is an alternative.

**Returns**

information about the python installation and packages

**Return type**

`dict`

**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 `numpy.double`

**hdf_write_attributes**( `hdf_path, attributes: dict = 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](https://stackoverflow.com/a/28238047)

**classmethod**( `fclass`)
instancemethod(finstance)

import_class(identifier: str)
	no comment

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 equiv-
  alent to running from numpy.linalg import norm and would return a reference to norm.

module_available(module_name: str) → bool

test 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

test 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: ArrayLike, dtype=None, copy: bool = True) → numpy.ndarray

test 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 sup-
     ported. 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
convert_scalar
    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
    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)
    apply nb.jit with predefined arguments
    Parameters
    • 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)
    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 function 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 nb.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 to nb.jit
    Returns The decorated function

numba_environment() → dict
    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 |

Continued on next page
<table>
<thead>
<tr>
<th>Class</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in_jupyter_notebook</code></td>
<td>checks whether we are in a jupyter notebook</td>
</tr>
<tr>
<td><code>BasicOutput</code></td>
<td>class that writes text line to stdout</td>
</tr>
<tr>
<td><code>JupyterOutput</code></td>
<td>class that writes text lines as html in a jupyter cell</td>
</tr>
</tbody>
</table>

```python
class BasicOutput(stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>):
    Bases: pde.tools.output.OutputBase

class JupyterOutput(
    header: str = '',
    footer: str = '')[168]
    Bases: pde.tools.output.OutputBase

class SimpleProgress(
    iterable=None,
    *args,
    **kwargs):
    Bases: object

class OutputBase:
    Bases: object
    base class for output management

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)

display_progress:
    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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeprecatedParameter</td>
<td>a parameter that can still be used normally but is deprecated</td>
</tr>
<tr>
<td>HideParameter</td>
<td>a helper class that allows hiding parameters of the parent classes</td>
</tr>
<tr>
<td>Parameterized</td>
<td>a mixin that manages the parameters of a class</td>
</tr>
<tr>
<td>get_all_parameters</td>
<td>get a dictionary with all parameters of all registered classes</td>
</tr>
</tbody>
</table>

class DeprecatedParameter(name: str, default_value=None, cls=<class 'object'>, description: str = '', hidden: bool = False, extra: dict = None)

Bases: pde.tools.parameters.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: dict = 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: dict = 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 instance of Parameter with their names as keys.

**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** = []

**show_parameters**(*description: 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

All flags default to *False*.

```python
get_all_parameters(data: str = 'name') → dict
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.
```

```python
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:

```python
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.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
parse_duration(value: str) → datetime.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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_scaled_colorbar</td>
<td>Add a vertical color bar to an image plot</td>
</tr>
<tr>
<td>disable_interactive</td>
<td>Context manager disabling the interactive mode of.matplotlib</td>
</tr>
<tr>
<td>plot_on_axes</td>
<td>Decorator for a plot method or function that uses a single axes</td>
</tr>
<tr>
<td>plot_on_figure</td>
<td>Decorator for a plot method or function that fills an entire figure</td>
</tr>
<tr>
<td>PlotReference</td>
<td>Contains all information to update a plot element</td>
</tr>
<tr>
<td>BasicPlottingContext</td>
<td>Basic plotting using just matplotlib</td>
</tr>
<tr>
<td>JupyterPlottingContext</td>
<td>Plotting in a jupyter widget using the inline backend</td>
</tr>
</tbody>
</table>

```python
class BasicPlottingContext(fig_or_ax=None, title: str = None, show: bool = True)
    Bases: pde.tools.plotting.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
```

```python
class JupyterPlottingContext(title: str = None, show: bool = True)
    Bases: pde.tools.plotting.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
```

```python
close()  # Close the plot

supports_update = False  # Flag indicating whether the context supports that plots can be updated without 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: dict = 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:

```python
context = PlottingContext()
with context:
    plt.plot(...)  # Example plot
    plt.xlabel(...) # Example label
```

Parameters

• title (str) – The shown in the plot
• show (bool) – Flag determining whether plots are actually shown

close()
    close the plot

supports_update = 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, aspect: float = 20, pad_fraction: float = 0.5, **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
• **aspect** *(float)* – The target aspect ratio of the colorbar

• **pad_fraction** *(float)* – Width of the gap between colorbar and image

• **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*

napari_add_layers(*viewer: napari.viewer.Viewer*, *layers_data: Dict[str, dict]*)

adds layers to a napari viewer

**Parameters**

• *viewer* *(napari.viewer.Viewer)* – The napari application

• *layers_data* *(dict)* – Data for all layers that will be added.

napari_viewer(*grid: GridBase*, *close=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

• *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. These options can be described in the docstring, if the placeholder `{PLOT_ARGS}` is mentioned in the docstring of the wrapped function or method. 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):
        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 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.

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. These options are available via keyword arguments. These options can be described in the docstring, if the placeholder `{PLOT_ARGS}` is mentioned in the docstring of the wrapped function or method. 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:

```python
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

```python
make_colored_noise
```

Return a function creating an array of random values that obey

\[
\langle c(k)c(k') \rangle = \Gamma^2 |k|^\nu \delta(k - k')
\]

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 \( \nu \) of the power spectrum

- **scale** – Scaling factor \( \Gamma \) determining noise strength

**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 \((r, \phi)\), where \(r\) is the radial coordinate and \(\phi\) is the polar angle:

\[
\begin{align*}
x &= r \cos(\phi) \\
y &= r \sin(\phi)
\end{align*}
\]

for \(r \in [0, \infty)\) and \(\phi \in [0, 2\pi)\)

Similarly, for spherical coordinates \((r, \theta, \phi)\), where \(r\) is the radial coordinate, \(\theta\) is the azimuthal angle, and \(\phi\) is the polar angle, we use

\[
\begin{align*}
x &= r \sin(\theta) \cos(\phi) \\
y &= r \sin(\theta) \sin(\phi) \\
z &= r \cos(\theta)
\end{align*}
\]

for \(r \in [0, \infty)\), \(\theta \in [0, \pi]\), 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius_from_volume</td>
<td>Return the radius of a sphere with a given volume</td>
</tr>
<tr>
<td>volume_from_radius</td>
<td>Return the volume of a sphere with a given radius</td>
</tr>
<tr>
<td>surface_from_radius</td>
<td>Return the surface area of a sphere with a given radius</td>
</tr>
<tr>
<td>points_cartesian_to_spherical</td>
<td>Convert points from Cartesian to spherical coordinates</td>
</tr>
<tr>
<td>points_spherical_to_cartesian</td>
<td>Convert points from spherical to Cartesian coordinates</td>
</tr>
<tr>
<td>haversine_distance</td>
<td>Calculate the haversine-based distance between two points on the surface</td>
</tr>
<tr>
<td>get_spherical_polygon_area</td>
<td>Calculate the surface area of a polygon on the surface</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20 – continued from previous page

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointsOnSphere</td>
<td>class representing points on an n-dimensional unit sphere</td>
</tr>
<tr>
<td>spherical_index_k</td>
<td>returns the mode ( k ) from the degree ( \text{degree} ) and order ( \text{order} )</td>
</tr>
<tr>
<td>spherical_index_lm</td>
<td>returns the degree ( \text{degree} ) and the order ( \text{order} ) from the mode ( k )</td>
</tr>
<tr>
<td>spherical_index_count</td>
<td>return the number of modes for all indices ( \leq \text{l} )</td>
</tr>
<tr>
<td>spherical_index_count_optimal</td>
<td>checks whether the modes captures all orders for maximal degree</td>
</tr>
<tr>
<td>spherical_harmonic_symmetric</td>
<td>axisymmetric spherical harmonics with degree ( \text{degree} ), so ( m=0 ).</td>
</tr>
<tr>
<td>spherical_harmonic_real</td>
<td>real spherical harmonics of degree ( l ) and order ( m )</td>
</tr>
<tr>
<td>spherical_harmonic_real_k</td>
<td>real spherical harmonics described by mode ( k )</td>
</tr>
</tbody>
</table>

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

    • filename (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: numpy.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/HLHuiliersTheorem.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, point2) → float
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 on the sphere (given in Cartesian coordinates)
- point2 (ndarray) – Second point on the sphere
- radius (float) – Radius of the sphere

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: numpy.ndarray) → numpy.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: numpy.ndarray) → numpy.ndarray
Convert points from spherical to Cartesian coordinates

Parameters
points (ndarray) – Points in spherical coordinates \((r, \theta, \phi)\)

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, \(\theta\): float, \(\phi\): 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
- \(\theta\) (float) – Azimuthal angle (in \([0, \pi]\)) at which the spherical harmonics is evaluated.
- \(\phi\) (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, \(\theta\): float, \(\phi\): float) → float
real spherical harmonics described by mode \(k\)

Parameters

- k (int) – Combined index determining the degree and order of the spherical harmonics
- \(\theta\) (float) – Azimuthal angle (in \([0, \pi]\)) at which the spherical harmonics is evaluated.
- \(\phi\) (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\( (\text{degree:}\ int, \ \theta: \ float) \rightarrow float \)

axisymmetric spherical harmonics with degree \( \text{degree} \), so \( m=0 \).

**Parameters**

- \( \text{degree} \ (\text{int}) \) – Degree of the spherical harmonics
- \( \theta \ (\text{float}) \) – 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\( (l: \ int) \rightarrow int \)

return the number of modes for all indices \(<= l\)

The returned value is one less than the maximal mode \( k \) required.

**Parameters**

- \( \text{degree} \ (\text{int}) \) – Maximal degree of the spherical harmonics

**Returns**

The number of modes

**Return type**

**int**

spherical_index_count_optimal\( (k\_\text{count}: \ int) \rightarrow bool \)

checks whether the modes captures all orders for maximal degree

**Parameters**

- \( \text{k\_count} \ (\text{int}) \) – The number of modes considered

spherical_index_k\( (\text{degree:} \ int, \ \text{order:} \ int = 0) \rightarrow int \)

returns the mode \( k \) from the degree \( \text{degree} \) and order \( \text{order} \)

**Parameters**

- \( \text{degree} \ (\text{int}) \) – Degree of the spherical harmonics
- \( \text{order} \ (\text{int}) \) – Order of the spherical harmonics

**Raises**

**ValueError** – if \( \text{order} < -\text{degree} \) or \( \text{order} > \text{degree} \)

**Returns**

a combined index \( k \)

**Return type**

**int**

spherical_index_lm\( (k: \ int) \rightarrow \text{Tuple}[int, int] \)

returns the degree \( l \) and the order \( m \) from the mode \( k \)

**Parameters**

- \( \text{k} \ (\text{int}) \) – The combined index for the spherical harmonics

**Returns**

The degree \( l \) and order \( m \) of the spherical harmonics associated with the combined index

**Return type**

**tuple**

surface_from_radius\( (\text{radius:} \ T\text{NumArr}, \ \text{dim:} \ int) \rightarrow T\text{NumArr} \)

Return the surface area of a sphere with a given radius

**Parameters**

- \( \text{radius} \ (\text{float or ndarray}) \) – Radius of the sphere
- \( \text{dim} \ (\text{int}) \) – Dimension of the space

**Returns**

Surface area of the sphere

**Return type**

**float or ndarray**

volume_from_radius\( (\text{radius:} \ T\text{NumArr}, \ \text{dim:} \ int) \rightarrow T\text{NumArr} \)

Return the volume of a sphere with a given radius

**Parameters**

- \( \text{radius} \ (\text{float or ndarray}) \) – Radius of the sphere
- \( \text{dim} \ (\text{int}) \) – Dimension of the space

**Returns**

Volume of the sphere

**Return type**

**float or ndarray**
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 that calls a function periodically
- **ProgressTracker** Tracker that shows the progress of the simulation
- **PrintTracker** Tracker that prints data to a stream (default: stdout)
- **PlotTracker** Tracker that plots the state, either on screen or to a file
- **LivePlotTracker** Tracker that plots data on screen, to files, or writes a movie
- **DataTracker** Tracker that stores custom data obtained by calling a function
- **SteadyStateTracker** Tracker that interrupts the simulation once steady state is reached
- **RuntimeTracker** Tracker that interrupts the simulation once a duration has passed
- **ConsistencyTracker** Tracker that interrupts the simulation when the state is not finite
- **InteractivePlotTracker** Tracker that shows the state live in an interactive napari instance

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

Continued on next page
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**($info$: dict = None) \rightarrow None

finalize the tracker, supplying additional information

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

**classmethod from_data**($data$: Union[TrackerBase, str], **kwargs) \rightarrow 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

**handle**($field$: FieldBase, $t$: float) \rightarrow None

handle data supplied to this tracker

**Parameters**

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

**initialize**($field$: FieldBase, $info$: dict = None) \rightarrow 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: dict = None) → None
finalze the tracker, supplying additional information

   Parameters info (dict) – Extra information from the simulation

classmethod from_data(data: TrackerData, **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

   • field (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: dict = 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 = None
The time of the next interrupt of the simulation

   Type float

tracker_action_times = None
Times at which the trackers need to be handled next

   Type list
4.7.2 pde.trackers.interactive module

Special module for defining an interactive tracker that uses napari to display fields

```python
class InteractivePlotTracker(interval: Union[ConstantIntervals, float, int, str] = '0:01', close: bool = True, show_time: bool = False)
    Bases: pde.trackers.base.TrackerBase

Tracker that shows the state live in an interactive napari instance
```

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: dict = None) → None

dezel 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

- **field** *(FieldBase)* – The current state of the simulation

- **t** *(float)* – The associated time

initialize(state: FieldBase, info: dict = 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 = '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: multiprocessing.context.BaseContext.Queue, initial_data: Dict[str, dict], t_initial: float = None, viewer_args: dict = 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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstantIntervals</strong></td>
<td>class representing equidistantly spaced time intervals</td>
</tr>
<tr>
<td><strong>LogarithmicIntervals</strong></td>
<td>class representing logarithmically spaced time intervals</td>
</tr>
</tbody>
</table>

Continued on next page
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) \rightarrow 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 pde.trackers.intervals.ConstantIntervals

class LogarithmicIntervals(dt_initial: float = 1, factor: float = 1, t_start: float = None)
    Bases: pde.trackers.intervals.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) \rightarrow 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: pde.trackers.intervals.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.

```python
def 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

```python
def 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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallbackTracker</td>
<td>Tracker that calls a function periodically</td>
</tr>
<tr>
<td>ProgressTracker</td>
<td>Tracker that shows the progress of the simulation</td>
</tr>
<tr>
<td>PrintTracker</td>
<td>Tracker that prints data to a stream (default: stdout)</td>
</tr>
<tr>
<td>PlotTracker</td>
<td>Tracker that plots the state, either on screen or to a file</td>
</tr>
<tr>
<td>LivePlotTracker</td>
<td>Tracker that plots data on screen, to files, or writes a movie</td>
</tr>
<tr>
<td>DataTracker</td>
<td>Tracker that stores custom data obtained by calling a function</td>
</tr>
<tr>
<td>SteadyStateTracker</td>
<td>Tracker that interrupts the simulation once steady state is reached</td>
</tr>
<tr>
<td>RuntimeTracker</td>
<td>Tracker that interrupts the simulation once a duration has passed</td>
</tr>
<tr>
<td>ConsistencyTracker</td>
<td>Tracker that interrupts the simulation when the state is not finite</td>
</tr>
<tr>
<td>MaterialConservationTracker</td>
<td>Ensure that the amount of material is conserved</td>
</tr>
</tbody>
</table>

```python
class CallbackTracker(func: Callable, interval: Union[ConstantIntervals, float, int, str] = 1)
    Bases: pde.trackers.base.TrackerBase
    # Tracker that calls 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. Al-
ternatively, 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.

```python
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 ProgressTracker(interval: Union[ConstantIntervals, float, int, str] = None, ndigits: int = 5, leave: bool = True)
Bases: pde.trackers.base.TrackerBase

Tracker that shows the progress of the simulation

```python
class ProgressTracker(interval: Union[ConstantIntervals, float, int, str] = None, ndigits: int = 5, leave: bool = True)
Bases: pde.trackers.base.TrackerBase

Tracker that shows 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)

```python
finalize(info: dict = 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: dict = 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

```python
name = 'progress'
```
Tracker that prints 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

```python
def 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

```python
name = 'print'
class PlotTracker(interval: Union[ConstantIntervals, float, int, str] = 1, *
    title: Union[str, Callable] = 'Time: {time:g}',
    output_file: str = None, movie: Union[str, pathlib.Path, Movie] = None,
    show: bool = None, plot_args: dict = None)
```

Tracker that plots the state, either on screen or to a file

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.

- **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`.

```python
finalize(info: dict = 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

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

initialize(state: FieldBase, info: dict = 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
```

```python
class LivePlotTracker(interval: Union[ConstantIntervals, float, int, str] = '0:03', *, show: bool = True, **kwargs)

    Bases: pde.trackers.trackers.PlotTracker

    Tracker that plots data on screen, to files, or writes a movie

    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.

**plot_args** *(dict)* – Extra arguments supplied to the plot call

```python
def finalize(info: dict = None) -> None:
    finalize the tracker, supplying additional information
```

```python
def handle(field: FieldBase, t: float) -> None:
    handle data supplied to this tracker
```

---

**class** `DataTracker(func: Callable, interval: Union[ConstantIntervals, float, int, str] = 1, filename: str = None)`

Tracker that stores 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`.

**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*
• **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 SteadyStateTracker(**interval: Union[ConstantIntervals, float, int, str] = None, atol: float = 1e-08, rtol: float = 1e-05, progress: bool = False)**

**Bases:** `pde.trackers.base.TrackerBase`

Tracker that interrupts 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`

**class RuntimeTracker(**max_runtime: Union[int, float, str], interval: Union[ConstantIntervals, float, int, str] = 1)**

**Bases:** `pde.trackers.base.TrackerBase`

Tracker that interrupts 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.

```python
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
```

```python
initialize(field: FieldBase, info: dict = 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
```

```python
class ConsistencyTracker(interval: Union[ConstantIntervals, float, int, str] = None)
Bases: pde.trackers.base.TrackerBase
Tracker that interrupts 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.

```python
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
```

```python
name = 'consistency'
class MaterialConservationTracker(interval: Union[ConstantIntervals, float, int, str] = 1, atol: float = 0.0001, rtol: float = 0.0001)
Bases: pde.trackers.base.TrackerBase
Ensure that the amount of material is conserved

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

```python
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

```python
initialize(field: FieldBase, info: dict = 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*

```python
name = 'material_conservation'
```

### 4.8 pde.visualization package

Functions and classes for visualizing simulations.

<table>
<thead>
<tr>
<th>movies</th>
<th>Functions for creating movies of simulation results</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotting</td>
<td>Functions and classes for plotting simulation data</td>
</tr>
</tbody>
</table>

### 4.8.1 pde.visualization.movies module

Functions for creating movies of simulation results

```python
Movie(filename: str, framerate: float = 30, dpi: float = None, **kwargs)
```

<table>
<thead>
<tr>
<th>Bases: object</th>
</tr>
</thead>
</table>

Class for creating movies from matplotlib figures using ffmpeg

```python
movie_scalar |
```

`produce a movie for a simulation of a scalar field`

```python
movie_multiple |
```

`produce a movie for a simulation with n components`

```python
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.

**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 :class:`matplotlib.animation.FFMpegWriter`. Here, we can for instance set the bit rate of the resulting video using the `bitrate` parameter.

```python
add_figure(fig=None)
```
add the figure `fig` as a frame to the current movie

**Parameters**

- `fig` (*Figure*) – The plot figure that is added to the movie

```python
classmethod is_available()
```
check whether the movie infrastructure is available

**Returns**

- `True` if movies can be created

**Return type**

- `bool`

```python
save()
```
convert the recorded images to a movie using ffmpeg

```python
movie(storage: StorageBase, filename: str, progress: bool = True, dpi: float = 150, show_time: bool = True, plot_args: dict = 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

```python
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 \([0, \text{scale}]\) is assumed), and ‘cmap’ (defining the colormap being used).

- **scale** \((\text{str, float, tuple of float})\) – 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** \((\text{bool})\) – Flag determining whether the progress of making the movie is shown.

```py
movie_scalar(storage: StorageBase, filename: str, scale: Union[\text{str, float, Tuple[float, float]}] = \text{‘automatic’}, extras: \text{dict} = \text{None}, progress: \text{bool} = \text{True}, tight: \text{bool} = \text{False}, show: \text{bool} = \text{True}) \rightarrow \text{None}
```

produce a movie for a simulation of a scalar field

**Parameters**

- **storage** \((\text{StorageBase})\) – The storage instance that contains all the data for the movie.

- **filename** \((\text{str})\) – The filename to which the movie is written. The extension determines the format used.

- **scale** \((\text{str, float, tuple of float})\) – 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** \((\text{dict, optional})\) – Additional functions that are evaluated and shown for each time step. The key of the dictionary is used as a panel title.

- **progress** \((\text{bool})\) – Flag determining whether the progress of making the movie is shown.

- **tight** \((\text{bool})\) – Whether to call `matplotlib.pyplot.tight_layout()`.

- **show** \((\text{bool})\) – Flag determining whether images are shown during making the movie.

### 4.8.2 pde.visualization.plotting module

Functions and classes for plotting simulation data

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ScalarFieldPlot</code></td>
<td>class managing compound plots of scalar fields</td>
</tr>
<tr>
<td><code>plot_magnitudes</code></td>
<td>plot spatially averaged quantities as a function of time</td>
</tr>
<tr>
<td><code>plot_kymograph</code></td>
<td>plots a single kymograph from stored data</td>
</tr>
<tr>
<td><code>plot_kymographs</code></td>
<td>plots kymographs for all fields stored in <code>storage</code></td>
</tr>
<tr>
<td><code>plot_interactive</code></td>
<td>plots stored data interactively using the <code>napari</code> viewer</td>
</tr>
</tbody>
</table>
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)

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`

```python
def plot_interactive(storage: StorageBase, time_scaling: str = 'exact', viewer_args: dict = 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.

```python
def 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: dict = None, fig_style: dict = None, ax=None, **kwargs)
```

Plots a single kymograph from stored data.

The kymograph shows line data stacked along time. Consequently, the resulting image shows space along the horizontal axis and time along the vertical axis.

**Parameters**

- `storage (StorageBase)` – The storage instance that contains all the data.
- `field_index (int)` – 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 be 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 figure. If the argument is set to `show`, `matplotlib.pyplot.show()` will be called to show the plot, if the value is `create`, the figure will be created, but not shown, and 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 a 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. If `None`, a new figure with a single axes is created.

• **kwargs** – Additional keyword arguments are passed to `matplotlib.pyplot.imshow()`.

**Returns** The reference to the plot

**Return type** `PlotReference`

```python
```

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 be 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 figure. If the argument is set to `show` `matplotlib.pyplot.show()` will be called to show the plot, if the value is `create`, the figure will be created, but not shown, and 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: dict = None, fig_style: dict = 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 figure. If the argument is set to show matplotlib.pyplot.show() will be called to show the plot, if the value is create, the figure will be created, but not shown, and 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. If None, a new figure with a single axes is created.
- ****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

4.8. pde.visualization package
<table>
<thead>
<tr>
<th>Character</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>pde.fields</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>pde.fields.base</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>pde.fields.collection</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>pde.fields.scalar</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>pde.fields.tensorial</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>pde.fields.vectorial</td>
<td>73</td>
</tr>
<tr>
<td>g</td>
<td>pde.grids</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>pde.grids.base</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>pde.grids.boundaries</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>pde.grids.boundaries.axes</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>pde.grids.boundaries.axis</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>pde.grids.boundaries.local</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>pde.grids.cartesian</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>pde.grids.cylindrical</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators.cartesian</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators.common</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators.cylindrical</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators.polar</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>pde.grids.operators.spherical</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>pde.grids.spherical</td>
<td>121</td>
</tr>
<tr>
<td>p</td>
<td>pde.pdes</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.aliens_cahn</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.base</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.cahn_hilliard</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.diffusion</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.kpz_interface</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.kuramoto_sivashinsky</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.laplace</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.pde</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.swift_hohenberg</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>pde.pdes.wave</td>
<td>137</td>
</tr>
<tr>
<td>s</td>
<td>pde.solvers</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>pde.solvers.base</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>pde.solvers.controller</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>pde.solvers.explicit</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>pde.solvers.implicit</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>pde.solvers.scipy</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>pde.storage</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>pde.storage.base</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>pde.storage.file</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>pde.storage.memory</td>
<td>148</td>
</tr>
<tr>
<td>t</td>
<td>pde.tools</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>pde.tools.cache</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>pde.tools.config</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>pde.tools.cuboid</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>pde.tools.docstrings</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>pde.tools.expressions</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>pde.tools.math</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>pde.tools.misc</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>pde.tools.numba</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>pde.tools.output</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>pde.tools.parms</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>pde.tools.parse_duration</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>pde.tools.plotting</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>pde.tools.spectral</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>pde.tools.spectral</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>pde.tools.typing</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>pde.trackers</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>pde.trackers.base</td>
<td>182</td>
</tr>
<tr>
<td></td>
<td>pde.trackers.interactive</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>pde.trackers.intervals</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>pde.trackers.trackers</td>
<td>187</td>
</tr>
<tr>
<td>v</td>
<td>pde.visualization</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td>pde.visualization.movies</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td>pde.visualization.plotting</td>
<td>196</td>
</tr>
</tbody>
</table>
Index

A

add_figure() (Movie method), 195
add_interpolated() (DataFieldBase method), 54
add_scaled_colorbar() (in module pde.tools.plotting), 172
AllenCahnPDE (class in pde.pdes.allen_cahn), 127
append() (StorageBase method), 144
apply() (FieldBase method), 61
apply() (StorageBase method), 144
asanyarray_flags() (in module pde.tools.cuboid), 158
assert_field_compatible() (FieldBase method), 61
assert_field_compatible() (FieldCollection method), 63
assert_grid_compatible() (GridBase method), 105
attributes (FieldBase attribute), 61
attributes (FieldCollection attribute), 63
attributes_serialized (FieldBase attribute), 61
attributes_serialized (FieldCollection attribute), 63
average (DataFieldBase attribute), 54
averages (FieldCollection attribute), 63
ax (PlotReference attribute), 172
axes (CylindricalGrid attribute), 117
axes (PolarGrid attribute), 121
axes (SphericalGrid attribute), 122
axes_bounds (GridBase attribute), 105
axes_coords (GridBase attribute), 105
axes_symmetric (CylindricalGrid attribute), 117
axes_symmetric (GridBase attribute), 105
axes_symmetric (PolarGrid attribute), 121
axes_symmetric (SphericalGrid attribute), 122
axis (BoundaryAxisBase attribute), 83
axis_coord (BCBase attribute), 87

B

BasicOutput (class in pde.tools.output), 167
BasicPlottingContext (class in pde.tools.plotting), 171
BCBase (class in pde.grids.boundaries.local), 87
BCBase1stOrder (class in pde.grids.boundaries.local), 89
BCBase2ndOrder (class in pde.grids.boundaries.local), 90
Boundaries (class in pde.grids.boundaries.axes), 81
BoundaryAxisBase (class in pde.grids.boundaries.axes), 83
BoundaryPair (class in pde.grids.boundaries.axes), 83
BoundaryPeriodic (class in pde.grids.boundaries.axes), 85
bounds (Cuboid attribute), 157
bounds (SmoothData1D attribute), 162
buffer() (Cuboid method), 157

C

cached_method (class in pde.tools.cache), 151
cached_property (class in pde.tools.cache), 153
CahnHilliardPDE (class in pde.pdes.cahn_hilliard), 130
CallbackTracker (class in pde.trackers.trackers), 187
CartesianGrid (class in pde.grids.cartesian), 111
CartesianGridBase (class in pde.grids.cartesian), 112
cell_coords (GridBase attribute), 105
cell_to_point() (CartesianGrid method), 111
cell_to_point() (CylindricalGrid method), 117
cell_to_point() (GridBase method), 105
cell_to_point() (PolarGridBase method), 122
cell_to_point() (UnitGrid method), 115
cell_volume_data (CartesianGridBase attribute), 112
cell_volume_data (CylindricalGrid attribute), 117
cell_volume_data (SphericalGridBase attribute), 122
cell_volumes (GridBase attribute), 105
centroid (Cuboid attribute), 157
check implementation (PDEBase attribute), 128
check_length () (DictFiniteCapacity method), 151
check_value_rank () (BCBase method), 87
check_value_rank () (Boundaries method), 81
check_value_rank () (BoundaryPair method), 83
check_value_rank () (BoundaryPeriodic method), 85
copy () (GridBase method), 106
copy () (MixedBC method), 94
copy () (StorageBase method), 145
corners (Cuboid attribute), 157
Cuboid (class in pde.tools.cuboid), 157
CurvatureBC (class in pde.grids.boundaries.local), 92
CylindricalGrid (class in pde.grids.cylindrical), 117

D
data (DataFieldBase attribute), 54
data (DataTracker attribute), 191
data (FieldBase attribute), 61
data (FileStorage attribute), 148
data (ScalarField attribute), 67
data (Tensor2Field attribute), 71
data (VectorField attribute), 74
data_shape (DataFieldBase attribute), 54
data_shape (StorageBase attribute), 145
DataFieldBase (class in pde.fields.base), 54
dataframe (DataTracker attribute), 191
DataTracker (class in pde.trackers.trackers), 191
decorator_arguments () (in module pde.tools.misc), 164
default_capacity (DictFiniteCapacity attribute), 151
deleter () (classproperty method), 163
depends_on () (ExpressionBase method), 160
DeprecationParameter (class in pde.tools.parameters), 168
derivatives (ScalarExpression attribute), 161
derivatives (TensorExpression attribute), 162
diagnostics (PDE attribute), 135
diagonal (Cuboid attribute), 157
DictFiniteCapacity (class in pde.tools.cache), 151
difference_vector_real () (CartesianGrid method), 111
difference_vector_real () (CylindricalGrid method), 118
difference_vector_real () (GridBase method), 106
difference_vector_real () (SphericalGridBase method), 123
difference_vector_real () (UnitGrid method), 116
differentiate () (ScalarExpression method), 161
differentiate () (TensorExpression method), 162
differentiated (BCBase attribute), 87
differentiated (Boundaries attribute), 82
differentiated (BoundaryPair attribute), 83
differentiated (BoundaryPeriodic attribute), 85
differentiated (DirichletBC attribute), 93
differentiated (NeumannBC attribute), 95
DiffusionPDE (class in pde.pdes.diffusion), 131
dim (Cuboid attribute), 157
dim (CylindricalGrid attribute), 118
Index

\texttt{dim (PolarGrid attribute)}, 121  
\texttt{dim (SphericalGrid attribute)}, 122  
\texttt{DimensionError}, 105  
\texttt{DirichletBC (class in \texttt{pde.grids.boundaries.local})}, 92  
\texttt{disable_interactive()} (in \texttt{module pde.tools.plotting}), 173  
\texttt{discretization (GridBase attribute)}, 106  
\texttt{discretize_interval()} (in \texttt{module pde.grids.base}), 110  
\texttt{display_progress()} (in \texttt{module pde.tools.output}), 167  
\texttt{distance_real()} (GridBase method), 106  
\texttt{divergence()} (Tensor2Field method), 71  
\texttt{divergence()} (VectorField method), 74  
\texttt{DomainError}, 105  
\texttt{dot()} (Tensor2Field method), 71  
\texttt{dot()} (VectorField method), 74  
\texttt{dtype (FieldBase attribute)}, 61  

\texttt{E}  
\texttt{element (PlotReference attribute)}, 172  
\texttt{end_writing()} (FileStorage method), 148  
\texttt{end_writing()} (StorageBase method), 145  
\texttt{ensure_directory_exists()} (in \texttt{module pde.tools.misc}), 164  
\texttt{environment variable \texttt{PYTHONPATH}}, 4  
\texttt{environment()} (in \texttt{module pde.tools.config}), 157  
\texttt{environment()} (in \texttt{module pde.tools.misc}), 164  
\texttt{estimate_computation_speed()} (in \texttt{module pde.tools.misc}), 164  
\texttt{evolution_rate()} (AllenCahnPDE method), 127  
\texttt{evolution_rate()} (CahnHilliardPDE method), 130  
\texttt{evolution_rate()} (DiffusionPDE method), 131  
\texttt{evolution_rate()} (KPZInterfacePDE method), 132  
\texttt{evolution_rate()} (KuramotoSivashinskyPDE method), 133  
\texttt{evolution_rate()} (PDE method), 136  
\texttt{evolution_rate()} (PDEBase method), 128  
\texttt{evolution_rate()} (SwiftHohenbergPDE method), 137  
\texttt{evolution_rate()} (WavePDE method), 138  
\texttt{explicit_time_dependence (AllenCahnPDE attribute)}, 127  
\texttt{explicit_time_dependence (CahnHilliardPDE attribute)}, 131  
\texttt{explicit_time_dependence (DiffusionPDE attribute)}, 131  
\texttt{explicit_time_dependence (KPZInterfacePDE attribute)}, 132  
\texttt{explicit_time_dependence (KuramotoSivashinskyPDE attribute)}, 133  
\texttt{explicit_time_dependence (PDEBase attribute)}, 128  
\texttt{explicit_time_dependence (SwiftHohenbergPDE attribute)}, 137  
\texttt{explicit_time_dependence (WavePDE attribute)}, 138  
\texttt{ExplicitSolver (class in \texttt{pde.solvers})}, 139  
\texttt{ExplicitSolver (class in \texttt{pde.solvers.explicit})}, 142  
\texttt{expr_prod()} (in \texttt{module pde.pdes.base}), 130  
\texttt{expression (AllenCahnPDE attribute)}, 127  
\texttt{expression (CahnHilliardPDE attribute)}, 131  
\texttt{expression (DiffusionPDE attribute)}, 131  
\texttt{expression (ExpressionBase attribute)}, 160  
\texttt{expression (KPZInterfacePDE attribute)}, 132  
\texttt{expression (KuramotoSivashinskyPDE attribute)}, 133  
\texttt{expression (SwiftHohenbergPDE attribute)}, 137  
\texttt{ExpressionBase (class in \texttt{pde.tools.expressions})}, 159  
\texttt{expressions (PDE attribute)}, 136  
\texttt{expressions (WavePDE attribute)}, 138  
\texttt{extract_component()} (BCBase method), 88  
\texttt{extract_component()} (Boundaries method), 82  
\texttt{extract_component()} (BoundaryPair method), 83  
\texttt{extract_component()} (BoundaryPeriodic method), 85  
\texttt{extract_field()} (in \texttt{module pde.visualization.plotting}), 198  
\texttt{extract_field()} (StorageBase method), 145  
\texttt{extract_time_range()} (StorageBase method), 146  
\texttt{ExtrapolateBC (class in \texttt{pde.grids.boundaries.local})}, 93  

\texttt{F}  
\texttt{factory (Operator attribute)}, 110  
\texttt{FieldBase (class in \texttt{pde.fields.base})}, 60  
\texttt{FieldCollection (class in \texttt{pde.fields.collection})}, 63  
\texttt{fields (FieldCollection attribute)}, 64  
\texttt{FileStorage (class in \texttt{pde.storage.file})}, 147  
\texttt{fill_in_docstring()} (in \texttt{module pde.tools.docstrings}), 159  
\texttt{finalize()} (DataTracker method), 191  
\texttt{finalize()} (InteractivePlotTracker method), 184  
\texttt{finalize()} (PlotTracker method), 190  
\texttt{finalize()} (ProgressTracker method), 188  
\texttt{finalize()} (StorageTracker method), 147  
\texttt{finalize()} (TrackerBase method), 182  
\texttt{finalize()} (TrackerCollection method), 183  
\texttt{FinishedSimulation}, 182  
\texttt{flat_idx (in \texttt{module pde.tools.numba})}, 166  
\texttt{fluctuations (DataFieldBase attribute)}, 55  
\texttt{from_bounds()} (\texttt{pde.tools.cuboid.Cuboid} \texttt{class method}), 157  
\texttt{from_centerpoint()} (\texttt{pde.tools.cuboid.Cuboid} \texttt{class method}), 158  
\texttt{from_collection()} (\texttt{pde.storage.memory.MemoryStorage} \texttt{class method}), 149
from_data() (pde.grids.boundaries.axes.Boundaries class method), 82
from_data() (pde.grids.boundaries.axis.BoundaryPair class method), 83
from_data() (pde.grids.boundaries.local.BCBase class method), 88
from_data() (pde.tracks.base.TrackerBase class method), 182
from_data() (pde.tracks.base.TrackerCollection class method), 183
from_dict() (pde.grids.boundaries.local.BCBase class method), 88
from_expression() (pde.fields.scalar.ScalarField class method), 68
from_expression() (pde.fields.vectorial.VectorField class method), 75
from_fields() (pde.storage.memory.MemoryStorage class method), 149
from_file() (pde.fields.base.FieldBase class method), 61
from_image() (pde.fields.scalar.ScalarField class method), 68
from_name() (pde.solvers.base.SolverBase class method), 141
from_points() (pde.tools.cuboid.Cuboid class method), 158
from_polar_coordinates() (CartesianGridBase method), 112
from_scalar_expressions() (pde.fields.collection.FieldCollection class method), 64
from_scalars() (pde.fields.vectorial.VectorField class method), 75
from_state() (pde.fields.base.DataFieldBase class method), 55
from_state() (pde.fields.base.FieldBase class method), 61
from_state() (pde.fields.collection.FieldCollection class method), 64
from_state() (pde.grids.base.GridBase class method), 106
from_state() (pde.grids.cartesian.CartesianGrid class method), 112
from_state() (pde.grids.cartesian.UnitGrid class method), 116
from_state() (pde.grids.cylindrical.CylindricalGrid class method), 118
from_state() (pde.grids.spherical.SphericalGridBase class method), 123
from_storage() (pde.visualization.plotting.ScalarField class method), 197
from_str() (pde.grids.boundaries.local.BCBase class method), 88

g
get() (ParameterModuleConstant method), 156
get_all_parameters() (in module pde.tools.parameters), 170
get_area_weights() (PointsOnSphere method), 177
get_boundary_axis() (in module pde.grids.boundaries.axis), 86
get_boundary_conditions() (CartesianGridBase method), 113
get_boundary_conditions() (CylindricalGrid method), 118
get_boundary_conditions() (GridBase method), 106
get_boundary_conditions() (SphericalGridBase method), 123
get_boundary_values() (DataFieldBase method), 55
get_cartesian_grid() (CylindricalGrid method), 118
get_cartesian_grid() (SphericalGridBase method), 123
get_common_dtype() (in module pde.tools.misc), 164
get_common_numba_dtype() (in module pde.tools.numba), 166
get_compiled() (ExpressionBase method), 160
get_compiled_array() (TensorExpression method), 162
get_data() (BCBase method), 89
get_data() (BCBase1stOrder method), 90
get_data() (BCBase2ndOrder method), 91
get_data() (BoundaryPair method), 84
get_data() (BoundaryPeriodic method), 85
get_distance_matrix() (PointsOnSphere method), 177
get_dot_operator() (Tensor2Field method), 72
get_dot_operator() (VectorField method), 75
get_help() (pde.grids.boundaries.axes.Boundaries class method), 82
get_help() (pde.grids.boundaries.axis.BoundaryPair class method), 84
get_help() (pde.grids.boundaries.local.BCBase class method), 89
get_image_data() (CartesianGridBase method), 113
get_image_data() (CylindricalGrid method), 118
get_image_data() (DataFieldBase method), 55
get_image_data() (FieldBase method), 61
get_image_data() (FieldCollection method), 64
get_image_data() (GridBase method), 106
get_image_data() (SphericalGridBase method), 124
get_initial_condition() (WavePDE method), 138
get_interval() (in module pde.tracks.intervals), 187
get_line_data() (CartesianGridBase method), 113
get_line_data() (CylindricalGrid method), 119
get_line_data() (DataFieldBase method), 56
get_line_data() (FieldBase method), 61
get_line_data() (FieldCollection method), 64
get_line_data() (GridBase method), 106
get_line_data() (SphericalGridBase method), 124
get_mean_separation() (PointsOnSphere method), 177
get_memory_storage() (in module pde.storage.memory), 149
get_operator() (GridBase method), 106
get_parameters() (pde.tools.parameters.Parameterized class method), 169
get_parameters() (pde.tools.parameters.Parameterized class method), 109
get_plotting_context() (in module pde.tools.plotting), 173
get_point_evaluator() (BoundaryPair method), 84
get_point_evaluator() (BoundaryPeriodic method), 85
get_progress_bar_class() (in module pde.tools.output), 167
get_random_point() (CartesianGridBase method), 114
get_random_point() (CylindricalGrid method), 119
get_random_point() (GridBase method), 107
get_random_point() (SphericalGridBase method), 124
get_spherical_polygon_area() (in module pde.tools.spherical), 177
get_subgrid() (CartesianGrid method), 112
get_subgrid() (CylindricalGrid method), 119
get_subgrid() (GridBase method), 107
get_subgrid() (UnitGrid method), 116
get_text_block() (in module pde.tools.docstrings), 159
get_vector_data() (DataFieldBase method), 56
get_vector_data() (VectorField method), 76
get_virtual_point() (BCBase method), 89
get_virtual_point() (BCBase1stOrder method), 90
get_virtual_point() (BCBase2ndOrder method), 91
get_virtual_point_data() (BCBase1stOrder method), 90
get_virtual_point_data() (BCBase2ndOrder method), 91
get_virtual_point_data() (CurvatureBC method), 92
get_virtual_point_data() (DirichletBC method), 93
get_virtual_point_data() (ExtrapolateBC method), 94
get_virtual_point_data() (MixedBC method), 95
get_virtual_point_data() (NeumannBC method), 95
getter() (classproperty method), 163
gradient() (ScalarField method), 68
gradient() (VectorField method), 76
gradient_squared() (ScalarField method), 69
grid (Boundaries attribute), 82
grid (BoundaryAxisBase attribute), 83
grid (DataFieldBase attribute), 54
grid (FieldBase attribute), 61
grid (ScalarField attribute), 67
grid (StorageBase attribute), 146
grid (Tensor2Field attribute), 71
grid (VectorField attribute), 73
GridBase (class in pde.grids.base), 105

H
handle() (CallbackTracker method), 188
handle() (ConsistencyTracker method), 193
handle() (DataTracker method), 191
handle() (InteractivePlotTracker method), 184
handle() (MaterialConservationTracker method), 194
handle() (PlotTracker method), 190
handle() (PrintTracker method), 189
handle() (ProgressTracker method), 188
handle() (RuntimeTracker method), 193
handle() (SteadyStateTracker method), 192
handle() (StorageTracker method), 147
handle() (TrackerBase method), 182
handle() (TrackerCollection method), 183
has_collection (StorageBase attribute), 146
has_hole (SphericalGridBase attribute), 124
has mutable() (in module pde.tools.cache), 155
has_readable() (in module pde.tools.cache), 155
has_spherical_distance() (in module pde.tools.spherical), 178
hdf_write_attributes() (in module pde.tools.misc), 164
HideParameter (class in pde.tools.parameters), 168
homogeneous (BCBase attribute), 89
hybridmethod (class in pde.tools.misc), 164

I
imag (FieldBase attribute), 62
ImplicitSolver (class in pde.solvers), 140
ImplicitSolver (class in pde.solvers.implicit), 143
import_class() (in module pde.tools.misc), 165
in_jupyter_notebook() (in module pde.tools.output), 168
initialize() (InteractivePlotTracker method), 184
initialize() (MaterialConservationTracker method), 194
make_inserter_compiled()  \hspace{1em} (GridBase method), 107
make_integrator()  \hspace{1em} (GridBase method), 107
make_interpolator()  \hspace{1em} (DataFieldBase method), 57
make_interpolator_compiled()  \hspace{1em} (GridBase method), 107
make_laplace()  \hspace{1em} (in module pde.grids.operators.cartesian), 96
make_laplace()  \hspace{1em} (in module pde.grids.operators.cylindrical), 99
make_laplace()  \hspace{1em} (in module pde.grids.operators.polar), 101
make_laplace()  \hspace{1em} (in module pde.grids.operators.spherical), 103
make_laplace_from_matrix()  \hspace{1em} (in module pde.grids.operators.common), 98
make_modify_after_step()  \hspace{1em} (PDEBase method), 128
make_movie()  \hspace{1em} (ScalarFieldPlot method), 198
make_normalize_point_compiled()  \hspace{1em} (GridBase method), 108
make_outer_product_operator()  \hspace{1em} (VectorField method), 77
make_pde_rhs()  \hspace{1em} (PDEBase method), 128
make_poisson_solver()  \hspace{1em} (in module pde.grids.operators.cartesian), 97
make_poisson_solver()  \hspace{1em} (in module pde.grids.operators.polar), 102
make_poisson_solver()  \hspace{1em} (in module pde.grids.operators.spherical), 104
make_radius_from_volume_compiled()  \hspace{1em} (in module pde.tools.spherical), 178
make_region_evaluator()  \hspace{1em} (BoundaryPair method), 84
make_region_evaluator()  \hspace{1em} (BoundaryPeriodic method), 86
make_region_evaluator()  \hspace{1em} (PDEBase method), 128
make_serializer()  \hspace{1em} (in module pde.tools.cache), 155
make_unserializer()  \hspace{1em} (in module pde.tools.cache), 155
make_vector_gradient()  \hspace{1em} (in module pde.grids.operators.cartesian), 97
make_vector_gradient()  \hspace{1em} (in module pde.grids.operators.cylindrical), 100
make_vector_gradient()  \hspace{1em} (in module pde.grids.operators.polar), 102
make_vector_gradient()  \hspace{1em} (in module pde.grids.operators.spherical), 104
make_vector_laplace()  \hspace{1em} (in module pde.grids.operators.cartesian), 97
make_vector_laplace()  \hspace{1em} (in module pde.grids.operators.cylindrical), 100
make_vector_laplace()  \hspace{1em} (in module pde.grids.operators.spherical), 103
make_virtual_point_evaluator()  \hspace{1em} (BCBase method), 89
make_virtual_point_evaluator()  \hspace{1em} (BCBase1stOrder method), 90
make_virtual_point_evaluator()  \hspace{1em} (BCBase2ndOrder method), 92
make_virtual_point_evaluators()  \hspace{1em} (BoundaryPair method), 84
make_virtual_point_evaluators()  \hspace{1em} (BoundaryPeriodic method), 86
make_volume_from_radius_compiled()  \hspace{1em} (in module pde.tools.spherical), 178
MaterialConservationTracker  \hspace{1em} (class in pde.trackers.trackers), 193
MemoryStorage  \hspace{1em} (class in pde.storage.memory), 148
MixedBC  \hspace{1em} (class in pde.grids.boundaries.local), 94
module_available()  \hspace{1em} (in module pde.tools.misc), 165
Movie  \hspace{1em} (class in pde.visualization.movies), 194
movie()  \hspace{1em} (in module pde.visualization.movies), 195
movie_multiple()  \hspace{1em} (in module pde.visualization.movies), 195
movie_scalar()  \hspace{1em} (in module pde.visualization.movies), 196
mutable  \hspace{1em} (Cuboid attribute), 158
N
name  \hspace{1em} (ConsistencyTracker attribute), 193
name  \hspace{1em} (ExplicitSolver attribute), 139, 142
name  \hspace{1em} (ImplicitSolver attribute), 140, 143
name  \hspace{1em} (MaterialConservationTracker attribute), 194
name  \hspace{1em} (PrintTracker attribute), 189
name  \hspace{1em} (ProgressTracker attribute), 188
name  \hspace{1em} (ScipySolver attribute), 141, 144
name  \hspace{1em} (SteadyStateTracker attribute), 192
names  \hspace{1em} (BCBase attribute), 89
names  \hspace{1em} (CurvatureBC attribute), 92
names  \hspace{1em} (DirichletBC attribute), 93
names  \hspace{1em} (ExtrapolateBC attribute), 94

Index
pde.grid.cartesian (module), 111
pde.grid.cylindrical (module), 117
pde.grid.operators (module), 96
pde.grid.operators.cartesian (module), 96
pde.grid.operators.common (module), 98
pde.grid.operators.cylindrical (module), 98
pde.grid.operators.polar (module), 100
pde.grid.operators.spherical (module), 102
pde.grid.spherical (module), 121
pde.pdes (module), 126
pde.pdes.alien_cahn (module), 127
pde.pdes.base (module), 127
pde.pdes.cahn_hilliard (module), 130
pde.pdes.diffusion (module), 131
pde.pdes.kpz_interface (module), 132
pde.pdes.kuramoto_sivashinsky (module), 132
pde.pdes.laplace (module), 133
pde.pdes.pde (module), 135
pde.pdes.swift_hohenberg (module), 136
pde.pdes.wave (module), 137
pde.solvers (module), 138
pde.solvers.base (module), 141
pde.solvers.controller (module), 141
pde.solvers.explicit (module), 142
pde.solvers.implicit (module), 143
pde.solvers.scipy (module), 143
pde.storage (module), 144
pde.storage.base (module), 144
pde.storage.file (module), 147
pde.storage.memory (module), 148
pde.tools (module), 150
pde.tools.cache (module), 150
pde.tools.config (module), 156
pde.tools.cuboid (module), 157
pde.tools.docstrings (module), 159
pde.tools.expressions (module), 159
pde.tools.math (module), 162
pde.tools.misc (module), 163
pde.tools.numba (module), 165
pde.tools.output (module), 166
pde.tools.parameters (module), 168
pde.tools.parse_duration (module), 170
pde.tools.plotting (module), 171
pde.tools.spectral (module), 175
pde.tools.spherical (module), 176
pde.tools.typing (module), 181
pde.trackers (module), 181
pde.trackers.base (module), 182
pde.trackers.interactive (module), 184
pde.trackers.intervals (module), 185
pde.trackers.trackers (module), 187
pde.visualization (module), 194
pde.visualization.movies (module), 194
pde.visualization.plotting (module), 196
scalar_random_uniform()
  (pde.fields.collection.FieldCollection class method), 66
ScalarExpression (class in pde.tools.expressions), 160
ScalarField (class in pde.fields.scalar), 67
ScalarFieldPlot (class in pde.visualization.plotting), 196
scale_value() (Boundaries method), 82
scale_value() (BoundaryPair method), 85
ScipySolver (class in pde.solvers), 140
ScipySolver (class in pde.solvers.scipy), 143
SerializedDict (class in pde.tools.cache), 151
set_description() (SimpleProgress method), 167
set_value() (Boundaries method), 82
set_value() (BoundaryPair method), 85
setter() (classproperty method), 163
shape (DataFieldBase attribute), 54
shape (ExpressionBase attribute), 160
shape (GridBase attribute), 109
shape (ScalarExpression attribute), 161
shape (StorageBase attribute), 146
shape (TensorExpression attribute), 162
show() (BasicOutput method), 167
show() (JupyterOutput method), 167
show() (OutputBase method), 167
show_parameters() (Parameterized method), 169
SimpleProgress (class in pde.tools.output), 167
size (Cuboid attribute), 158
skipUnlessModule() (in module pde.tools.misc), 165
slice() (ScalarField method), 70
smooth() (DataFieldBase method), 60
smooth() (FieldCollection method), 67
SmoothData1D (class in pde.tools.math), 162
solve() (PDEBase method), 129
solve_poisson_equation() (in module pde.pdes.laplace), 133
solve_poisson_equation() (in module pde.pdes.laplace), 134
SolverBase (class in pde.solvers.base), 141
spherical_harmonic_real() (in module pde.tools.spherical), 179
spherical_harmonic_real_k() (in module pde.tools.spherical), 179
spherical_harmonic_symmetric() (in module pde.tools.spherical), 179
spherical_index_count() (in module pde.tools.spherical), 180
spherical_index_count_optimal() (in module pde.tools.spherical), 180
spherical_index_k() (in module pde.tools.spherical), 180
spherical_index_lm() (in module pde.tools.spherical), 180
SphericalGrid (class in pde.grids.spherical), 121
SphericalGridBase (class in pde.grids.spherical), 122
sphinx_display_parameters() (in module pde.tools.parameters), 170
start_writing() (FileStorage method), 148
start_writing() (MemoryStorage method), 149
start_writing() (StorageBase method), 146
state (CartesianGrid attribute), 112
state (CylindricalGrid attribute), 120
state (GridBase attribute), 109
state (SphericalGridBase attribute), 126
state (UnitGrid attribute), 116
state_serialized (GridBase attribute), 109
SteadyStateTracker (class in pde.trackers.trackers), 192
storage (StorageTracker attribute), 146
StorageBase (class in pde.storage.base), 144
StorageTracker (class in pde.storage.base), 146
supports_update (JupyterPlottingContext attribute), 171
supports_update (PlottingContextBase attribute), 172
surface_area (Cuboid attribute), 158
surface_from_radius() (in module pde.tools.spherical), 180
SwiftHohenbergPDE (class in pde.pdes.swift_hohenberg), 136
symmetrize() (Tensor2Field method), 73

T

t_range (Controller attribute), 139, 142
Tensor2Field (class in pde.fields.tensorial), 71
tensor_expression (class in pde.tools.tensorial), 161
time_next_action (TrackerCollection attribute), 183
times (DataTracker attribute), 191
times (FileStorage attribute), 148
to_cartesian() (UnitGrid method), 116
to_dict() (Config method), 156
to_file() (DataTracker method), 192
to_file() (FieldBase method), 62
to_scalar() (DataFieldBase method), 60
to_scalar() (ScalarField method), 70
to_scalar() (Tensor2Field method), 73
to_scalar() (VectorField method), 77
trace() (Tensor2Field method), 73
tracker() (StorageBase method), 146
tracker_action_times (TrackerCollection attribute), 183
TrackerBase (class in pde.trackers.trackers), 182

214 Index
TrackerCollection (class in pde.trackers.base), 182
trackers (TrackerCollection attribute), 182
transpose() (Tensor2Field method), 73
typical_discretization (GridBase attribute), 110

U
uniform_cell_volumes (GridBase attribute), 110
UnitGrid (class in pde.grids.cartesian), 115
unserialize_attributes()
  (pde.fields.base.DataFieldBase class method), 60
unserialize_attributes()
  (pde.fields.base.FieldBase class method), 62
unserialize_attributes()
  (pde.fields.collection.FieldCollection class method), 67
update() (DictFiniteCapacity method), 151
update() (NapariViewer method), 185
update() (ScalarFieldPlot method), 198

V
value (BCBase attribute), 89
value (ScalarExpression attribute), 161
value (TensorExpression attribute), 162
value_is_linked (BCBase attribute), 89
variables (PDE attribute), 135
VectorField (class in pde.fields.vectorial), 73
vertices (Cuboid attribute), 158
volume (CartesianGrid attribute), 112
volume (Cuboid attribute), 158
volume (CylindricalGrid attribute), 120
volume (GridBase attribute), 110
volume (SphericalGridBase attribute), 126
volume (UnitGrid attribute), 116
volume_from_radius() (in module pde.tools.spherical), 180

W
WavePDE (class in pde.pdes.wave), 137
write_to_xyz() (PointsOnSphere method), 177