nata

Release 0.0.2

Anton Helm

Aug 24, 2020

INTRODUCTION

nata is a python package for post-processing and visualizing simulation output for particle-in-cell codes. It utilizes the numpy interface to provide a simple way to read, manipulate, and represent simulation output.

CHAPTER

ONE

INSTALLING NATA

Nata is available on PyPI. You can install it by running the following command inside your terminal

```
pip install nata
```

It is intended to be used inside a jupyter together with ipywidgets. Hence, you might need to run after the installation

```
# can be skipped for notebook version 5.3 and above
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

and if you want to use it inside JupyterLab (note that this requires nodejs to be installed)

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

In case of issues, please visit the installation section of ipywidgets for further details.

CHAPTER

TWO

CONTRIBUTING TO NATA

Any type of contribution to nata is appreciated. If you have any issues, please report them here. But if you wish to directly contribute to nata, we recommend to setup a local development environment.

2.1 Getting the source code

The source code is hosted on GitHub. Simply create a fork and apply your changes. You can always push any changes to your local fork, but if you would like to share your contribution, please create a pull request, so that it can be reviewed.

2.2 Local development environment

For the local development environment, we use poetry. This allows us to deal better with dependencies issues and to ensure coding standards without the burden of manually fixing and checking for styles. To use poetry, simply install it using the latest version by running the following command in your terminal.

```
curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py␣
↪| python
```

Afterwards, simply run

```
poetry install
```

which will install all the dependencies (including development packages) in a virtual environment. If you wish to run a command inside the virtual environment, simply run it by `poetry run <your command>`, e.g. to run all the test

```
poetry run pytest
```

Alternatively, you can directly use the virtual environment by running the command

```
poetry shell
```

which will spawn a shell inside the virtual environment. With this, you can even run jupyter outside of the source directory.

In addition, we use pre-commit to help us keep consistency in our development without any additional burden. Please use it as well. Inside the root directory of the repository, run the command

```
poetry run pre-commit install
```

which will create commit hooks for you and modify files during git commits, keeping a consistent structure.

CHAPTER

THREE

BASIC CONCEPTS

Natas idea of generalizing data reading, data processing, and data visualization is build on top of three core concepts, *backends*, *datasets*, and *plugins*. It is important to understand the core principles of this concepts. All supported types are stored in `nata.types` and are Protocols which mimic static-typing-like behavior in a duck-typing environment. They don't provide any functionality except allowing to have a "ground truth" to the objects inside nata and to support of type checking. Most of the objects allow you to easily check if an object fulfills the protocol by `isinstance(instance_of_object, some_protocol)`.

3.1 Backends

Note: This section characterizes protocols for backends. The purpose of a protocol is to provide **support for nominal and structural subtyping**.

Backend provide access to stored data. In general, each backend is of type *BackendType*.

class nata.types.**BackendType**(**args*, ***kwds*)

Bases: typing_extensions.Protocol

General type for retreiving data.

The *BackendType* characterizes a general behavior of a backend. It is a Protocol which characterizes available attributes and their corresponding type annotation.

name: str

Name of the backend. The name can be chosen individually and is used for providing users with information about the underlaying data storage. It should follow the convention `"CODENAME_VERSION_DATATYPE_STORAGE"`, e.g. `"osiris_4.4.4_grid_hdf5"`

location: Optional[Union[str, pathlib.Path]]

Location of the data. This attribute can be used inside a backend to point to data, either to open a file or to retrieve it.

To avoid validity checks at initialization level, each backend has a simple static-method of receiving a location and deducing if the backend is a valid backend, given a location.

static BackendType.**is_valid_backend**(*location: Union[pathlib.Path, str]*) → bool

Determine if a backend is a valid backend.

Parameters **location** (str or pathlib.Path) – Checks if a location can be passed to backend to initiate it.

Returns **out** (bool) – Returns `True` if `location` is valid input parameter for instantiation and `False` otherwise.

3.1.1 Backends for grids

For retrieving grid based data, the *GridBackendType* extends the base backend type.

class nata.types.**GridBackendType**(**args*, ***kwds*)
Bases: *nata.types.BackendType*, typing_extensions.Protocol

Backend representing a grid.

GridBackendType is a protocol with the purpose of characterizing attributes being available for object to be recognized as a *GridBackendType*. Reading data is not part of this protocol but is characterized by *GridDataReader* which extends this protocol.

dataset_name: **str**
Name of the dataset. It has to be identifiable, e.g. `"some_dataset_name"`.

dataset_label: **str**
Descriptive label of the dataset. Can be an arbitrary string, e.g. `"some long label with space"`.

dataset_unit: **str**
Unit of the correspinding grid. Can be an string including some latex symbols, e.g. `"m_e c \\ omega_p e^{-1}"`.

axes_names: **Sequence[str]**
A sequence of strings for each grid axis. Each string of the sequence has to be identifiable, e.g. `["axis0", "axis1", "axis2"]`.

axes_labels: **Sequence[str]**
A sequence of strings for each grid axis. Each string of the sequence is a descriptive label for each axis, e.g. `["some axis 0", "some axis 1", "some axis 2"]`.

axes_units: **Sequence[str]**
A sequence of strings for each grid axis. Each string of the sequence is the unit for each axis including some latex symbols, e.g. `["c / \\omega_p", "mm", "\\omega_p^{-1}"]`.

axes_min: **numpy.ndarray**
An array representing the lower limits of each grid axis.

axes_max: **numpy.ndarray**
An array representing the upper limits of each grid axis.

iteration: **int**
Associated iteration step of the underlaying data.

time_step: **float**
::Associated time step of the underlaying data in code units.

time_unit: **str**
Unit for time. Can be an arbitrary string, e.g. `"1 / \\omega_p"`.

shape: **Tuple[int, ...]**
Tuple of grid array dimensions. Corresponds to `numpy.ndarray.shape`.

dtype: **numpy.dtype**
Data type object of the grid array. Corresponds to `numpy.dtype`.

ndim: **int**
Dimensionality of the grid. Corresponds to `numpy.ndarray.ndim`

class nata.types.**GridDataReader**(**args*, ***kwds*)
Bases: *nata.types.GridBackendType*, typing_extensions.Protocol

Extended backend which handles grid data reading

get_data(*indexing: Optional[Union[int, slice, Tuple[Union[int, slice], ...]]] = None*) → numpy.ndarray

Routine for reading underlaying grid data.

Parameters **indexing** (int, slice, typing.Tuple[Union[slice, int], ...]], optional) – Optional indexing for reading a section of the grid. Any basic slicing and indexing can be passed here.

Returns **out** (numpy.ndarray) – Data array of the underlaying grid.

3.1.2 Backends for particles

For retrieving particle based data, the *ParticleBackendType* extends the base backend type.

class nata.types.**ParticleBackendType**(**args*, ***kwds*)

Bases: *nata.types.BackendType*, typing_extensions.Protocol

Backend representing a particles.

ParticleBackendType is a protocol with the purpose of characterizing attributes being available for object to be recognized as a *ParticleBackendType*. Reading data is not part of this protocol but is characterized by *ParticleDataReader* which extends this protocol.

dataset_name: **str**
Name of the dataset. It has to be identifiable, e.g. "some_dataset_name".

num_particles: **int**
Number of particles for a given backend.

quantity_names: **Sequence[str]**
A sequence of strings for each quantity stored in a backend. Each string of the sequence has to be identifiable, e.g. ["quant0", "quant1", "quant2"].

quantity_labels: **Sequence[str]**
A sequence of strings for each quantity stored in a backend. Each string of the sequence is a descriptive label for each quantity, e.g. ["some quantity 0", "some quantity 1", "some quantity 2"].

quantity_units: **Sequence[str]**
A sequence of strings for the unit of each quantity. Each string of the sequence is the unit for each quantity including some latex symbols, e.g. ["m_e", "c / \\omega_p"].

iteration: **int**
Associated iteration step of the underlaying data.

time_step: **float**
Associated time step of the underlaying data in code units.

time_unit: **str**
Unit for time. Can be an arbitrary string, e.g. "1 / \\omega_p".

dtype: **numpy.dtype**
Structured type of the underlaying particle backend. Field names correspond to quantity names, and the type corresponds to the array type.

In addition, for reading the underlying particle array the *ParticleDataReader* extends the *ParticleBackendType*.

class nata.types.**ParticleDataReader**(**args*, ***kwds*)

Bases: *nata.types.ParticleBackendType*, typing_extensions.Protocol

Extended backend which handles particle data reading

get_data(*indexing: Optional[Union[int, slice]] = None*, *fields: Optional[Union[str, Sequence[str]]] = None*) → numpy.ndarray
Routine for reading underlaying grid data.

Parameters

- **indexing** (int, slice, typing.Tuple[Union[slice, int], ...], optional) – Optional indexing for reading a section of the grid. Any basic slicing and indexing can be passed here.
- **fields** (str, typing.Sequence[str], optional) – Optional field or Sequence of fields to read out. Each field correspond to a quantity stored in the backend.

Returns **out** (numpy.ndarray) – Data array of the underlaying particle array.

3.2 Datasets

Note: This section characterizes protocols for datasets. The purpose of a protocol is to provide **support for nominal and structural subtyping**.

Datasets, as a protocol, are mutable containers with the base type of *DatasetType*. The mutability arises naturally from the need of allowing data to be appended, e.g. a dataset which contain similar information, but at a different time. In addition to it, datasets have the possibility to interact with backends to obtain required data.

class nata.types.**DatasetType**(**args*, ***kwds*)
Bases: typing_extensions.Protocol

Base protocol for datasets.

Each dataset has one private attribute which stores the information about available backends and the possibility of storing/removing backends to the backend store.

_backends: AbstractSet[*nata.types.BackendType*]
Storage of available backends for a dataset.

To interact with the dataset store, each dataset has to provide the following class methods:

classmethod DatasetType.**add_backend**(*backend:* nata.types.BackendType) → None
Attach a new backend to backend store.

Parameters **backend** (*BackendType*) – Backend which will be stored in *_backends*.

classmethod DatasetType.**remove_backend**(*backend:* nata.types.BackendType) → None
Remove an attached backend from backend store.

Parameters **backend** (*BackendType*) – Backend which will be removed from *_backends*.

classmethod DatasetType.**is_valid_backend**(*backend:* nata.types.BackendType) → bool
Checks if a backend is a valid backend for a dataset.

Parameters **backend** (*BackendType*) – Backend which will be checked if it is a valid backend for a dataset.

Returns **out** (bool) – True if a backend is a valid backend for a Dataset and False otherwise.

classmethod DatasetType.**get_backends**() → Dict[str, *nata.types.BackendType*]
Obtain information over stored backends.

Returns a dict with information of the stored backends inside a dataset. The keys are of type str and are the names of the backends. The values of the dictionary are the backends *BackendType*.

Next to having a tight connection to backends, datasets provide a way of interacting with other datasets, especially with similar datasets. For this, two methods are part the *DatasetType* protocol.

DatasetType.**equivalent**(*other:* nata.types.DatasetType) → bool
Checks for equivalence of two datasets.

Parameters **other** (*DatasetType*) – The other dataset which will be checked.

Returns **out** (bool) – Returns True if an instance of a datasets is equivalent with another datasets, False otherwise.

DatasetType.**append**(*other:* nata.types.DatasetType) → None
Appends another dataset.

Parameters **other** (*DatasetType*) – The other dataset which will be appended.

3.2.1 Datasets for grids

GridDatasetType extends the *DatasetType* protocol to include additional information for grids.

class nata.types.**GridDatasetType**(**args*, ***kwds*)
Bases: *nata.types.DatasetType*, typing_extensions.Protocol

Base protocol for GridDatasets.

Extends *DatasetType* to include additional information for grids.

name: **str**
Name of the grid dataset. It has to be identifiable, e.g. "some_grid_name".

label: **str**
Descriptive label of the grid. Can be an arbitrary string, e.g. "some long label for a grid".

unit: **str**
Unit of the correspinding grid. Can be an string including some latex symbols, e.g. "m_e c \\ omega_p e^{-1}".

axes: ***nata.types.GridDatasetAxes***
Axes for *GridDatasetType*. It is a dictionary of type *GridDatasetAxes*.

grid_shape: **Tuple[int, ...]**
Shape of the grid. The grid shape corresponds to the underlaying grid and does not include temporal infortions.

3.2.2 Datasets for particles

ParticleDatasetType extends the *DatasetType* protocol to include additional information for grids.

class nata.types.**ParticleDatasetType**(**args*, ***kwds*)
Bases: *nata.types.DatasetType*, typing_extensions.Protocol

Base protocol for ParticleDatasets.

Extends *DatasetType* to include additional information for particles.

name: **str**
Name of the particle dataset. It has to be identifiable, e.g. "some_particle_species_name".

quantities: **Mapping[str, *nata.types.QuantityType*]**
Mapping storing information about stored quantaties. Keys represent are names for quantities and values are Store

axes: *nata.types.ParticleDatasetAxes*
Axes for *ParticleDatasetType*. It is a dictionary of type *ParticleDatasetAxes*.

Particle datasets are in general containers which store particle quantities which follow the *QuantityType* protocol.

class nata.types.**QuantityType**(**args*, ***kwds*)
Bases: typing_extensions.Protocol

Base protocol for particle quantities.

name: str
Name of the quantity. It has to be identifiable, e.g. "some_quantity_name".

label: str
Descriptive label of the particle quantity. Can be an arbitrary string, e.g. "some long label for a particle quantity".

unit: str
Unit of the correspinding particle quantity. Can be an string including some latex symbols, e.g. "m_e c \\omega_p e^{-1}".

Particle quantities following the *QuantityType* protocol have in addition methods to append more data to them.

QuantityType.**equivalent**(*other:* nata.types.QuantityType) → bool
Checks for equivalence of two particle quantaties.

Parameters **other** (*QuantityType*) – The other particle quantity which will be checked.

Returns **out** (bool) – Returns True if an instance of a particle quantity is equivalent with another particle quantity, False otherwise.

QuantityType.**append**(*other:* nata.types.QuantityType) → None
Appends another a particle quantity.

Parameters **other** (*QuantityType*) – The other quantity which will be appended.

3.3 Axes

Note: This section characterizes protocols for axes. The purpose of a protocol is to provide **support for nominal and structural subtyping**.

3.3.1 Axis protocols

One essential building block for datasets like *GridDatasetType* and *ParticleDatasetType* are *AxisType* and *GridAxisType*. Similar to datasets, protocols are in place to provide available attributes.

class nata.types.**AxisType**(**args*, ***kwds*)
Bases: typing_extensions.Protocol

Base protocol for an axis.

name: str
Name of the axis. It has to be identifiable, e.g. "some_axis_name".

label: str
Descriptive label of the axis. Can be an arbitrary string, e.g. "some long label for an axis".

unit: *str*
Unit of the correspinding axis. Can be an string including some latex symbols, e.g. `"m_e c \\ omega_p e^{-1}"`.

axis_dim: *int*
Dimensionality of an axis.

In addition, *AxisType* provides methods to check for equivalence between axes and to append another axis.

`AxisType.`**equivalent**(*other:* nata.types.AxisType) → bool
Checks for equivalence of two axes.

Parameters **other** (*AxisType*) – The other axis which will be checked.

Returns **out** (`bool`) – Returns `True` if an instance of a is equivalent with another particle quantity, `False` otherwise.

`AxisType.`**append**(*other:* nata.types.AxisType) → None
Appends another axis.

Parameters **other** (*AxisType*) – The other axis which will be appended.

As grid axes provide uniformity for each dimension of a grid, the *GridAxisType* extends *AxisType*.

class `nata.types.`**GridAxisType**(**args*, ***kwds*)
Bases: *nata.types.AxisType*, `typing_extensions.Protocol`

axis_type: *str*
Axis type of a grid axis.

3.3.2 Axes container

Axes serve the purpose of providing meta information for *DatasetType* and are in general occurring in a combination with other axes. For this, two special purpose axes container exist, *GridDatasetAxes* and *ParticleDatasetAxes*.

class `nata.types.`**GridDatasetAxes**(***kwargs*)
Bases: `dict`

Typed dictionary containing axes for grid datasets.

Typed dictionary `typing.TypedDict` which correspond to a `dict` at runtime are merely there for type checking. The attributes correspond to required keys inside a dictionary.

iteration: **Optional[*nata.types.AxisType*]**
Axis to store iteration information.

time: **Optional[*nata.types.AxisType*]**
Axis to store time information.

grid_axes: **Sequence[*nata.types.GridAxisType*]**
Sequence of *GridAxisType* representing an axis for each grid dimension.

class `nata.types.`**ParticleDatasetAxes**(***kwargs*)
Bases: `dict`

Typed dictionary containing axes for particle datasets.

Typed dictionary `typing.TypedDict` which correspond to a `dict` at runtime are merely there for type checking. The attributes correspond to required keys inside a dictionary.

iteration: **Optional[*nata.types.AxisType*]**
Axis to store iteration information.

time: **Optional[*nata.types.AxisType*]**
Axis to store time information.

3.4 Array interface

Note: This section characterizes protocols for backends. The purpose of a protocol is to provide **support for nominal and structural subtyping**.

Next to backends, dataset, and axes, nata provides objects to have an array interface. Inside nata, a interface is determined by *HasArrayInterface* protocol. It is especially important to allow dispatching for numpy.

class nata.types.**HasArrayInterface**(**args*, ***kwds*)
Bases: typing_extensions.Protocol

Base protocol for an object to be 'characterized' as an array.

__array__(*dtype: Optional[numpy.dtype] = None*) → numpy.ndarray
Array interface of an object.

Method which will be called when numpy.array or similar function called with the object provided as an input.

data: **numpy.ndarray**
Represents the stored data for an object. It is of type numpy.ndarray.

dtype: **numpy.dtype**
Data type object of the stored data. It is of type numpy.dtype.

shape: **Tuple[int, ...]**
Shape of the underlaying stored data. Similar to numpy.ndarray.shape

ndim: **int**
Dimensionality of the underlaying stored data. Similar to numpy.ndarray.ndim

3.5 Plugins

Plugins provide a way of extending the functionality of a dataset. In particular, they allow to add a method to a dataset on which they operator. This allows for developing a pipeline-like interface. For this, the decorator *register_container_plugin* is provided.

@nata.plugins.**register_container_plugin**(*callable_or_container=None*, *container=None*, *name: Optional[str] = None*)
Decorator for registering a plugin for a container.

CHAPTER

FOUR

PROPERTIES OF GRID CONTAINERS

The class `nata.containers.GridDataset` holds the information of an underlying grid.

CHAPTER

FIVE

GRIDDATASET CONTAINER

The GridDataset container provides a sophisticated way to store information about grids. In most cases, calling the class method *GridDataset.from_array()* is recommended. The common way of creating objects is as well available for GridDataset, but is reserved for advanced usage.

5.1 Creating a GridDataset container

classmethod GridDataset.**from_array**(*array: Union[numpy.ndarray, Sequence[Union[float, int]]], name: str = 'unnamed', label: str = 'unnamed', unit: str = '', time: Optional[Union[*nata.types.AxisType*, numpy.ndarray, Sequence[Union[float, int]]]] = None, iteration: Optional[Union[*nata.types.AxisType*, numpy.ndarray, Sequence[Union[float, int]]]] = None, grid_axes: Optional[Sequence[Union[*nata.types.GridAxisType*, numpy.ndarray, Sequence[Union[float, int]]]]] = None*)

Initialize GridDataset from an array.

As in general, `GridDataset` container provide a rich-API, *GridDataset.from_array()* allows to create naively a object with the source data coming from a numpy array and pre-defined objects for the axes.

Parameters

- **array** (*array-like object*) – Input data, in any form that can be converted to an numpy array or a numpy array itself. The array dimension correspond the grid dimension if no temporal information is provided. Otherwise, the first dimension is consumed.
- **name** (`str`, default value: `"unnamed"`) – Name of the grid container and expected to be identifiable.
- **label** (`str`, default value: `"unnamed"`) – Label of the grid container with a descriptive meaning. It is not expected to be identifiable.
- **unit** (`str`, default value: `""`) – Unit of the grid container.
- **time** (*array-like, axis object, optional*) – Time axis of the grid container. If an array-like object is provided, an axis object is created underneath. In addition, an axis object can be provided which has to fulfill the *nata.types.AxisType* protocol. If nothing is provided (default option), a time axis with the single value `0.0` is created.
- **iteration** (*array-like, axis object, optional*) – Iteration axis of the grid container. If an array-like object is provided, an axis object is created underneath. In addition, an axis object can be provided which has to fulfill the *nata.types.AxisType* protocol. If nothing is provided (default option), a iteration axis with a single value `0` is created.

- **grid_axes** (*sequence of array like objects and/or grid axis objects, optional*) – Sequence characterizing each grid axis. The length of the sequence has to correspond to the dimension of the array. In the absence of temporal axes, the length corresponds to the array dimension otherwise the first axes of the array is consumed.

CHAPTER

SIX

GRIDDATASET PLUGINS

6.1 Lineout

nata.containers.GridDataset.**lineout**(*dataset: nata.containers.GridDataset*, *fixed: Union[str, int]*, *value: float*) → nata.containers.GridDataset

Takes a lineout across a two-dimensional, single/multiple iteration nata.containers.GridDataset:

Parameters

- **fixed** (:class:str or :class:int) – Selection of the axes along which the taken lineout is constant.
  - if it is a string, then it must match the name property of an existing grid axis in dataset.
  - if it is an integer, then it must match the index of a grid axis in dataset (i.e. 0 or 1).
- **value** (*scalar*) – Value between the minimum and maximum of the axes selected through fixed over which the lineout is taken.

Returns nata.containers.GridDataset – One-dimensional nata.containers.GridDataset.

Examples

The following example shows how to obtain a lineout from a two-dimensional nata.containers.GridDataset. Since no axes are attributed to the dataset in this example, they are automatically generated with no names, and fixed must be an integer.

```
>>> from nata.containers import GridDataset
>>> import numpy as np
>>> arr = np.arange(25).reshape((5,5))
>>> ds = GridDataset(arr[np.newaxis])
>>> lo = ds.lineout(fixed=0, value=2)
>>> lo.data
array([10, 11, 12, 13, 14])
```

6.2 FFT

nata.containers.GridDataset.**fft**(*dataset: nata.containers.GridDataset*, *type: Optional[str] = 'abs'*) → nata.containers.GridDataset

Computes the Fast Fourier Transform (FFT) of a single/multiple iteration `nata.containers.GridDataset` along all grid axes using numpy's fft module.

Parameters **type** ({'abs', 'real', 'imag', 'full'}, optional) – Defines the component of the FFT selected for output. Available values are `'abs'` (default), `'real'`, `'imag'` and `'full'`, which correspond to the absolute value, real component, imaginary component and full (complex) result of the FFT, respectively.

Returns `nata.containers.GridDataset` – Selected FFT component along all grid axes of `dataset`.

Examples

To obtain the FFT of a `nata.containers.GridDataset`, a simple call to the `fft()` method is enough. In the following example, we compute the FFT of a one-dimensional `nata.containers.GridDataset`.

```
>>> from nata.containers import GridDataset
>>> import numpy as np
>>> x = np.linspace(100)
>>> arr = np.exp(-(x-len(x)/2)**2)
>>> ds = GridDataset(arr[np.newaxis])
>>> ds_fft = ds.fft()
```

6.3 Plot

nata.containers.GridDataset.**plot**(*dataset: nata.containers.GridDataset*, *fig: Optional[nata.plots.figure.Figure] = None*, *axes: Optional[nata.plots.axes.Axes] = None*, *style: Optional[dict] = {}*, *interactive: Optional[bool] = True*, *n: Optional[int] = 0*) → Optional[nata.plots.figure.Figure]

Plots a single/multiple iteration `nata.containers.GridDataset` using a *nata.plots.types.LinePlot* or *nata.plots.types.ColorPlot* if the dataset is one- or two-dimensional, respectively.

Parameters

- **fig** (*nata.plots.Figure*, optional) – If provided, the plot is drawn on `fig`. The plot is drawn on `axes` if it is a child axes of `fig`, otherwise a new axes is created on `fig`. If `fig` is not provided, a new *nata.plots.Figure* is created.
- **axes** (*nata.plots.Axes*, optional) – If provided, the plot is drawn on `axes`, which must be an axes of `fig`. If `axes` is not provided or is provided without a corresponding `fig`, a new *nata.plots.Axes* is created in a new *nata.plots.Figure*.
- **style** (`dict`, optional) – Dictionary that takes a mix of style properties of *nata.plots.Figure*, *nata.plots.Axes* and any plot type (see *nata.plots.types.LinePlot* or *nata.plots.types.ColorPlot*).
- **interactive** (`bool`, optional) – Controls wether interactive widgets should be shown with the plot to allow for temporal navigation. Only applicable if `dataset` has multiple iterations.

- **n** (`int`, optional) – Selects the index of the iteration to be shown initially. Only applicable if `dataset` has multiple iterations, .

Returns *nata.plots.Figure* or `None` – Figure with plot built based on `dataset`. Interactive widgets are shown with the figure if `dataset` has multiple iterations, in which case this method returns `None`.

Examples

To get a plot with default style properties in a new figure, simply call the `.plot()` method of the dataset.

```
>>> from nata.containers import GridDataset
>>> import numpy as np
>>> arr = np.arange(10)
>>> ds = GridDataset.from_array(arr)
>>> fig = ds.plot()
```

In case a *nata.plots.Figure* is returned by the method, it can be shown by calling the *nata.plots.Figure.show()* method.

```
>>> fig.show()
```

To draw a new plot on `fig`, we can pass it as an argument to the `.plot()` method. If `axes` is provided, the new plot is drawn on the selected axes.

```
>>> ds2 = GridDataset.from_array(arr**2)
>>> fig = ds2.plot(fig=fig, axes=fig.axes[0])
```

CHAPTER

SEVEN

PROPERTIES OF PARTICLE CONTAINERS

CHAPTER

EIGHT

PARTICLEDATASET PLUGINS

8.1 Filter

nata.containers.ParticleDataset.**filter**(*dataset: nata.containers.ParticleDataset*, *mask: List[bool] = None*, *quantities: List[str] = None*, *slicing: slice = None*) → nata.containers.ParticleDataset

Filters a `nata.containers.ParticleDataset` according to a selection of quantities.

Parameters

- **mask** (`np.ndarray`, optional) – Array of booleans indicating the particles to be filtered. Particles with `True` (`False`) mask entries are selected (hidden). The shape of `mask` must match that of each particle quantity.
- **slicing** (`slice`, optional) – Slice of particles to be filtered. Acts only on particle indices and not on time, as time slicing should be done on the dataset. When provided together with the `mask argument`, slicing is done on the masked array.
- **quantities** (`list`, optional) – List of quantities to be filtered, ordered by the way they should be sorted in the returned dataset.

Returns `nata.containers.ParticleDataset` – Filtered dataset with only the quantities selected in `quantities`.

Examples

The filter plugin is used to get dataset with only a selection, say `'x1'` and `'x2'`, of its quantities.

```
>>> from nata.containers import ParticleDataset
>>> ds = ParticleDataset("path/to"file")
>>> ds_flt = ds.filter(quantities=["x1","p1"])
```

8.2 Plot

nata.containers.ParticleDataset.**plot**(*dataset: nata.containers.ParticleDataset*, *fig: Optional[nata.plots.figure.Figure] = None*, *axes: Optional[nata.plots.axes.Axes] = None*, *style: dict = {}*, *interactive: bool = True*, *n: int = 0*)

Plots a single/multiple iteration `nata.containers.ParticleDataset` using a *nata.plots.types.ScatterPlot*.

Parameters

- **fig** (*nata.plots.Figure*, optional) – If provided, the plot is drawn on `fig`. The plot is drawn on `axes` if it is a child axes of `fig`, otherwise a new axes is created on `fig`. If `fig` is not provided, a new *nata.plots.Figure* is created.
- **axes** (*nata.plots.Axes*, optional) – If provided, the plot is drawn on `axes`, which must be an axes of `fig`. If `axes` is not provided or is provided without a corresponding `fig`, a new *nata.plots.Axes* is created in a new *nata.plots.Figure*.
- **style** (`dict`, optional) – Dictionary that takes a mix of style properties of *nata.plots.Figure*, *nata.plots.Axes* and any plot type (see *nata.plots.types.ScatterPlot*).
- **interactive** (`bool`, optional) – Controls wether interactive widgets should be shown with the plot to allow for temporal navigation. Only applicable if `dataset` has multiple iterations.
- **n** (`int`, optional) – Selects the index of the iteration to be shown initially. Only applicable if `dataset` has multiple iterations, .

Returns *nata.plots.Figure* or `None` – Figure with plot built based on `dataset`. Interactive widgets are shown with the figure if `dataset` has multiple iterations, in which case this method returns `None`.

Examples

To get a plot with default style properties in a new figure, simply call the `.plot()` method. The first two quantities in the dataset `quantities` dictionary will be represented in the horizontal and vertical plot axes, respectively. If a third quantity is available, it will be represented in colors.

```
>>> from nata.containers import ParticleDataset
>>> import numpy as np
>>> arr = np.arange(30).reshape(1,10,3)
>>> ds = ParticleDataset("path/to/file")
>>> fig = ds.plot()
```

The list of quantities in the dataset can be filtered with the *nata.containers.ParticleDataset.filter()* method.

```
>>> fig = ds.filter(quantities=["x1", "p1", "ene"]).plot()
```

CHAPTER

NINE

INTRODUCTION

nata provides a simple way of plotting the supported dataset types, such as `nata.containers.GridDataset` or `nata.containers.ParticleDataset`.

For example, to plot a `nata.containers.GridDataset` `dataset`, simply call

```
dataset.plot()
```

Apart from using the data itself in `dataset`, the plot will be built using all available metadata, such as derived axes labels and units or titles.

Plot calls on single iteration datasets return a *nata.plots.Figure* object. Figures can be also be shown by calling the *nata.plots.Figure.show()* method.

```
fig = dataset.plot()
fig.show()
```

By default, *nata.plots.Figure* objects are shown when represented in HTML. Since this is the default representation method in notebook environment calls, there is no need to call the *nata.plots.Figure.show()* method in order to show a *nata.plots.Figure* object in a jupyter notebook if `.plot()` is the last instruction in the cell.

9.1 Customizing plots

A big effort is continuously put into **nata** to produce out-of-the-box nearly publication-ready plots. However, the plots are highly customizable through the `style` parameter, a dictionary that takes a combination of figure, axes and plot style parameters. For example, we can set the figure size, the horizontal axes scale and label and the line color of our plot in `style` altogether:

```
dataset.plot(
    style=dict(
        figsize=(5,4),
        xscale="log",
        xlabel="$x$ [m]",
        color="blue",
    )
)
```

All style parameters that are by default inferred from the represented dataset(s) are overridden if specified in `style`.

Naturally, plot type specific style parameters will only be applicable if that plot type is drawn in the current call. A list of all available style parameters can be found on the description of all classes in *Plots Reference / API*.

9.2 Combining plots

Plots can be combined by setting the `fig` and `axes` attributes in the dataset plot plugin calls. If `fig` is provided and is an existing *nata.plots.Figure* instance, the current plot will be added to that instance. Additionally, if `axes` is provided and is an existing *nata.plots.Axes* child object of `fig`, the current plot will be added to that axes.

For example, to combine two line plots of the datasets `ds_1` and `ds_2` in one figure, but in different axes, we can do:

```
fig = ds_1.plot()
fig = ds_2.plot(fig=fig)
```

When only `fig` (and not `axes`) is provided in a `.plot()` call, a new axes is added to the existing figure. If the figure has all its axes occupied, a new row for axes is created.

We can also represent the two line plots in the same axes, by doing:

```
fig = ds_1.plot()
fig = ds_2.plot(fig=fig, axes=fig.axes[0])
```

For more details about combining plots and the automatic restyling of the corresponding axes and figures, see for example *nata.containers.GridDataset.plot()*.

Combining plots is also possible by applying addition and multiplication operators (`+` and `*`) to *nata.plots.Figure* objects. These are shortcuts for the instructions given above.

For example, to represent the two datasets `ds_1` and `ds_2` in the same figure, but in different axes, we can do:

```
fig = ds_1.plot() + ds_2.plot()
```

To represent the two datasets in the same axes, we can do instead:

```
fig = ds_1.plot() * ds_2.plot()
```

If the two figures involved in the `*` operation have more than one axes, then all plots in axes with matching indices will be combined.

CHAPTER

TEN

PLOTS REFERENCE / API

10.1 Figure

class nata.plots.**Figure**(*figsize: Optional[Tuple[float]] = None*, *nrows: Optional[int] = 1*, *ncols: Optional[int] = 1*, *style: Optional[str] = 'light'*, *fname: Optional[str] = None*, *rc: Optional[Dict[str, Any]] = None*)

Container of parameters and child objects (including plotting backend-related objects) relevant to draw a figure.

Parameters

- **figsize** (tuple of float, optional) – Tuple containing the width and height of the figure canvas in inches. If not provided, defaults to (6,4).
- **nrows** (int, optional) – Number of rows available for figure axes. If not provided, defaults to 1.
- **ncols** (int, optional) – Number of columns available for figure axes. If not provided, defaults to 1.
- **style** ({'light', 'dark'}, optional) – Selection of standard nata style. If not provided, defaults to 'light'.
- **fname** (str, optional) – Path to file with custom plotting backend parameters.
- **rc** (dict, optional) – Dictionary with custom plotting backend parameters. Overrides parameters given in fname.

show()

Shows the figure.

save(*path*, *format: Optional[str] = None*, *dpi: Optional[float] = 150*)

Saves the figure to a file.

Parameters

- **path** (tuple of float, optional) – Path in which to store the file.
- **format** (str, optional) – File format, e.g. 'png', 'pdf', 'svg'. If not provided, the output format is inferred from the extension of path.
- **dpi** (float, optional) – Resolution in dots per inch. If not provided, defaults to 150.

property axes

Dictionary of child *nata.plots.Axes* objects, where the key to each axes is its index property

10.2 Axes

class nata.plots.**Axes**(*xlim: Optional[tuple] = None*, *ylim: Optional[tuple] = None*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *title: Optional[str] = None*, *xscale: Optional[str] = 'linear'*, *yscale: Optional[str] = 'linear'*, *linthreshx: Optional[float] = None*, *linthreshy: Optional[float] = None*, *aspect: Optional[str] = 'auto'*, *legend_show: Optional[bool] = True*, *legend_loc: Optional[str] = 'upper right'*, *legend_frameon: Optional[bool] = False*, *cb_show: Optional[bool] = True*, *index: Optional[int] = 0*, *fig: Any = None*)

Container of parameters and parent and child objects (including plotting backend-related objects) relevant to draw a figure axes.

Parameters

- **xlim** (`tuple`, optional) – Limits of the horizontal axis in the format `(min,max)`. If not provided, it is inferred from the dataset(s) represented in the axes.
- **ylim** (`tuple`, optional) – Same as `xlim` for the vertical axis.
- **xlabel** (`str`, optional) – Label of the horizontal axis. If not provided, it is inferred from the dataset(s) represented in the axis.
- **ylabel** (`str`, optional) – Same as `xlabel` for the vertical axis.
- **xscale** (`{'linear','log', 'symlog'}`, optional) – Scale of the horizontal axis. If not provided, defaults to `'linear'`. If set to `'symlog'`, `linthreshx` is required.
- **yscale** (`{'linear','log', 'symlog'}`, optional) – Same as `xscale` for the vertical axis.
- **linthreshx** (`float`, optional) – Range within which the horizontal axis is linear. Applicable only when `xscale` is set to `'symlog'`.
- **linthreshy** (`float`, optional) – Same as `linthreshx` for the vertical axis.
- **title** (`str`, optional) – Axes title. If not provided, it is inferred from the dataset(s) represented in the axes.
- **legend_show** (`bool`, optional) – Controls the visibility of the axes legend, when applicable. If not provided, defaults to `True`.
- **legend_loc** (`str`, optional) – Controls the position of the axes legend, when applicable. See `matplotlib.axes.Axes.legend()` for available options. If not provided, defaults to `'upper right'`.
- **legend_frameon** (`bool`, optional) – Controls the visibility of the axes legend frame. If not provided, defaults to `False`.
- **cb_show** (`bool`, optional) – Controls the visibility of the axes colorbar, when applicable. If not provided, defaults to `True`.
- **index** (`int`, optional) – Position of the axes in the parent figure. Must be between `0` and `N-1`, where `N` is the number of child axes objects in the parent figure. Increases along rows before columns.

10.3 Plot Types

10.3.1 BasePlot

class nata.plots.types.**BasePlot**(*label: Optional[str] = None*, *data: nata.plots.data.PlotData = None*, *axes: Any = None*)

Base class for plot types.

Parameters **label** (str, optional) – Label of plot, used to identify the plot in the parent *nata.plots.Axes* object legend. If not provided, it is inferred from the child dataset object.

10.3.2 LinePlot

class nata.plots.types.**LinePlot**(*label: Optional[str] = None*, *data: nata.plots.data.PlotData = None*, *axes: Any = None*, *ls: Optional[str] = None*, *lw: Optional[float] = 1*, *color: Optional[str] = None*, *alpha: Optional[float] = None*, *marker: Optional[str] = None*, *ms: Optional[float] = None*, *antialiased: Optional[bool] = True*)

Line plot class.

Parameters

- **ls** (str, optional) – Linestyle of the line. See matplotlib.lines.Line2D.set_linestyle() for available options.
- **lw** (float, optional) – Line width in points. If not provided, defaults to 1.
- **color** (str, optional) – Color of the line. See matplotlib.colors for available options.
- **alpha** (float, optional) – Line alpha value. Must be between 0 and 1.
- **marker** (str, optional) – Marker to be used in defined line points. See matplotlib.markers for available options.
- **ms** (float, optional) – Marker size in points.
- **antialiased** (bool, optional) – Controls wether the plot should be antialiased. If not provided, defaults to True.

10.3.3 ColorPlot

class nata.plots.types.**ColorPlot**(*label: Optional[str] = None*, *data: nata.plots.data.PlotData = None*, *axes: Any = None*, *vmin: Optional[float] = None*, *vmax: Optional[float] = None*, *cb_map: Optional[str] = 'rainbow'*, *cb_scale: Optional[str] = 'linear'*, *cb_linthresh: Optional[float] = 1e-05*, *cb_title: Optional[str] = None*, *interpolation: Optional[str] = 'none'*)

Color plot class.

Parameters

- **vmin** (float, optional) – Minimum of the colorbar axis. If not provided, it is inferred from the dataset represented in the plot.
- **vmax** (float, optional) – Same as vmin for the maximum of the colorbar axis.

- **cb_title** (str, optional) – Colorbar title. If not provided, it is inferred from the dataset represented in the plot.
- **cb_scale** ({'linear','log', 'symlog'}, optional) – Scale of the colorbar. If not provided, defaults to 'linear'.
- **cb_map** (str, optional) – Colormap used to represent the data. See matplotlib.pyplot.colormaps() for available options. If not provided, defaults to rainbow.
- **cb_linthresh** (float, optional) – Range within which the colorbar axis is linear. Applicable only when cb_scale is set to 'symlog'. If not provided, defaults to 1e-5.
- **interpolation** (str, optional) – Interpolation method used. See matplotlib.pyplot.imshow() for available options. If not provided, defaults to none.

10.3.4 ScatterPlot

class nata.plots.types.**ScatterPlot**(*label: Optional[str] = None, data: nata.plots.data.PlotData = None, axes: Any = None, s: Optional[float] = 0.1, c: Optional[str] = None, marker: Optional[str] = None, alpha: Optional[float] = None, vmin: Optional[float] = None, vmax: Optional[float] = None, cb_map: Optional[str] = 'rainbow', cb_scale: Optional[str] = 'linear', cb_linthresh: Optional[float] = 1e-05, cb_title: Optional[str] = None*)

Color plot class.

Parameters

- **s** (float, optional) – Marker size in in points**2. If not provided, defaults to 0.1
- **c** (str, optional) – Color of the markers. See matplotlib.colors for available options.
- **marker** (str, optional) – Marker style. See matplotlib.markers for available options.
- **alpha** (float, optional) – Marker alpha value. Must be between 0 and 1.
- **vmin** (float, optional) – Minimum of the colorbar axis. If not provided, it is inferred from the dataset represented in the plot.
- **vmax** (float, optional) – Same as vmin for the maximum of the colorbar axis.
- **cb_title** (str, optional) – Colorbar title. If not provided, it is inferred from the dataset represented in the plot.
- **cb_scale** ({'linear','log', 'symlog'}, optional) – Scale of the colorbar. If not provided, defaults to 'linear'.
- **cb_map** (str, optional) – Colormap used to represent the data. See matplotlib.pyplot.colormaps() for available options. If not provided, defaults to rainbow.
- **cb_linthresh** (float, optional) – Range within which the colorbar axis is linear. Applicable only when cb_scale is set to 'symlog'. If not provided, defaults to 1e-5.

Notes

All colorbar parameters are only applicable if the dataset represented in the plot has a quantity to be represented in color. In this case, `c` is overriden if set.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

L

N

P

Q

R

S

T

U