
odl Documentation

Release beta

Jonas Adler, Holger Kohr, Ozan Öktem

March 11, 2016

1	Working with ODL	3
1.1	As a user	3
1.2	As a developer	3
2	Contents	5
2.1	User's guide to ODL	5
2.2	Contributing to odl	15
2.3	Frequently asked questions	32
2.4	Glossary	34
2.5	Mathematics behind ODL	35
2.6	Release Notes	39
2.7	References	41
2.8	odl	41
3	Indices and tables	605
	Bibliography	607

Operator Discretization Library (ODL) is a python library for fast prototyping focusing on (but not restricted to) inverse problems. ODL is being developed at KTH, Royal Institute of Technology.

The main intent of ODL is to enable mathematicians and applied scientists to use different numerical methods on real-world problems without having to implement all necessary parts from the bottom up. ODL provides some of the most heavily used building blocks for numerical algorithms out of the box, which enables users to focus on real scientific issues.

Working with ODL

1.1 As a user

User's guide to ODL

1.2 As a developer

Contributing to odl

Contents

2.1 User's guide to ODL

Welcome to the ODL users guide, this guide is intended to give you a simple introduction to ODL and how to work with it. If you need help on a specific function you should look at its documentation.

2.1.1 Introduction

About ODL

ODL is a Python library for inverse problems. It contains all the abstract mathematical tools needed from analysis, such as sets, vector spaces and operators as well as optimized implementations of many of the most common of these.

Set

A *Set* is the fundamental building block of odl objects. It is intended to mirror the mathematical concept of a *set*, and has methods to check if an element is contained in the set

```
>>> interv = Interval(0, 1)
>>> 0.5 in interv
True
>>> 2.0 in interv
False
```

The most commonly used sets in odl is the *RealNumbers* the set of all real numbers and *IntervalProd*, arbitrary n-dimensional hypercubes. Several convenience sub-classes such as *Interval*, *Rectangle* and *Cuboid* are also provided.

LinearSpace

A *LinearSpace* is a very important subclass of *Set* and is a general implementation of a mathematical vector space. In odl there are a few kinds of spaces that you will face.

- Continuous function spaces such as *FunctionSpace*, intended to represent mathematical *Function* space such as the lebesgue space *Lp* or the space of *Linear operators*. These are mostly used to represent abstract concepts, and are seldomly used in actual computation.
- \mathbb{F}^n type spaces such as *Rn* and *Cn*, but also the *CUDA* accelerated version *CudaRn*.

- Discretizations of continuous spaces. This may for example be a discretization of a cube using voxels. All discretizations inherit from [RawDiscretization](#), but the most important is [DiscreteLp](#).
- In addition to this, there are utility spaces such as [ProductSpace](#) which allows the composition of several spaces into a larger space.

In addition to the spaces, all elements in the spaces inherit from [LinearSpaceVector](#). Using these vectors, most standard mathematical operations can be expressed

```
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> y = r3.element([4, 5, 6])
```

Arithmetic such as addition and multiplication by scalars

```
>>> x + y
Rn(3).element([5.0, 7.0, 9.0])
```

Inner product etc are defined

```
>>> r3.inner(x, y)
32.0
```

See also in depth guide on [LinearSpace](#).

Operator

A operator is a [function](#) from some [Set](#) to another. In odl these inherit from the abstract class [Operator](#).

See also in depth guide on [Operators](#).

Discretizations

Discretizations ...

Diagnostics

Odl also offers tools to verify the correctness of operators and spaces. Examples include verifying that a [Operator](#)'s derivative is correct:

```
>>> op = MyOperator()
>>> odl.diagnostics.OperatorTest(op).derivative()
```

or verifying that a [LinearSpace](#) satisfies all expected properties

```
>>> r3 = odl.Rn(5)
>>> odl.diagnostics.SpaceTest(r3).run_tests()
```

See [SpaceTest](#) and [OperatorTest](#) for more details.

Installing

Installing odl is intended to be straightforward, and this guide is meant for new users. To install odl you need to do the following steps:

1. [Install Python](#)

2. *Install Git*
3. *Install ODL*
4. *Run tests*

If you have done any step before, you can ofc skip it.

Install Python

To begin with, you need a python distribution. If you are an experienced user, you can use any distribution you'd like.

Anaconda If you are a python novice using Windows, we recommend that you install a full package such as Anaconda. To install Anaconda

1. Download Anaconda from [anaconda's webpage](#)
2. Once installed run in a console

```
user$ conda update --all
```

to make sure you have the latest versions of all packages

Install Git

You also need to install Git to be able to download odl.

Overview	Debian / Ubuntu	<code>sudo apt-get install git</code>
	Fedora	<code>sudo yum install git</code>
	Windows	Download and install msysGit
	OS X	Use the git-osx-installer

In detail See the git page for the most recent information.

Have a look at the github install help pages available from [github help](#)

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Install ODL

You are now ready to install ODL! To do that, run the following where you want to install it

```
git clone https://github.com/odlgroup/odl
cd odl
```

For installation in a local user folder run

```
user$ pip install --user -e .
```

For system-wide installation, run (as root, e.g. using sudo or equivalent)

```
root# pip install -e .
```

(Optional) Install ODLpp

If you also wish to use the (optional) CUDA extensions you need to run

```
user$ git submodule update --init --recursive
user$ cd odlpp
```

From here follow the instructions in odlpp and install it. You then need to re-install ODL.

Run tests

To verify your installation you should run some basic tests. To run these:

```
user$ py.test
```

This requires the module `pytest`

Getting started

Welcome to the ODL users guide, this guide is intended to give you a simple introduction to ODL and how to work with it. If you need help on a specific function you should look at its documentation.

2.1.2 In depth

This is a more in depth guide to the different parts of ODL.

Operators

Operators in ODL are represented by the abstract `Operator` class. As an *abstract class*, it cannot be used directly but must be subclassed for concrete implementation. To define your own operator, you start by writing:

```
class MyOperator(odl.Operator):
    ...
```

`Operator` has a couple of *abstract methods* which need to be explicitly overridden by any subclass, namely

domain: `Set` Set of elements to which the operator can be applied

range [`Set`] Set in which the operator takes values

As a simple example, you can implement the matrix multiplication operator

$$\mathcal{A} : \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad \mathcal{A}(x) = Ax$$

for a matrix $A \in \mathbb{R}^{n \times m}$ as follows:

```
from builtins import super
import numpy as np

class MatVecOperator(odl.Operator):
    def __init__(self, matrix):
        assert isinstance(matrix, np.ndarray)
        self.matrix = matrix
        dom = odl.Rn(matrix.shape[1])
        ran = odl.Rn(matrix.shape[0])
        super().__init__(dom, ran)
```

In addition, an *Operator* needs at least one way of evaluation, *in-place* or *out-of-place*.

In place evaluation

In-place evaluation means that the operator is evaluated on a *Operator.domain* element, and the result is written to an *already existing Operator.range* element. To implement this behavior, create the (private) `Operator._apply` method with the following signature, here given for the above example:

```
class MatVecOperator(odl.Operator):
    ...
    def _apply(self, x, out):
        self.matrix.dot(x, out=out.asarray())
```

In-place evaluation is usually more efficient and should be used *whenever possible*.

Out-of-place evaluation

Out-of-place evaluation means that the operator is evaluated on a *domain* element, and the result is written to a *newly allocated range* element. To implement this behavior, create the (private) `Operator._call` method with the following signature, here given for the above example:

```
class MatVecOperator(odl.Operator):
    ...
    def _call(self, x):
        return self.range.element(self.matrix.dot(x))
```

Out-of-place evaluation is usually less efficient since it requires allocation of an array and a full copy and should be *generally avoided*.

Important: Do not call these methods directly. Use the call pattern `operator(x)` or `operator(x, out=y)`, e.g.:

```
matrix = np.array([[1, 0], [0, 1], [1, 1]])
operator = MatVecOperator(matrix)
x = odl.Rn(2).one()
y = odl.Rn(3).element()

# Out-of-place evaluation
y = operator(x)

# In-place evaluation
operator(x, out=y)
```

This public calling interface is type-checked, so the private methods do not need to implement type checks.

Operator arithmetic

It is common in applications to perform arithmetic with operators, for example the addition of matrices

$$[A + B]x = Ax + Bx$$

or multiplication of a functional by a scalar

$$[\alpha x^*](x) = \alpha x^*(x)$$

Another example is matrix multiplication, which corresponds to operator composition

$$[AB](x) = A(Bx)$$

All available operator arithmetic is shown below. A , B represent arbitrary *Operator*'s, f is an *Operator* whose *Operator.range* is a *Field* (sometimes called a *functional*), and a is a scalar.

Code	Meaning	Class
$(A + B)(x)$	$A(x) + B(x)$	<i>OperatorSum</i>
$(A * B)(x)$	$A(B(x))$	<i>OperatorComp</i>
$(a * A)(x)$	$a * A(x)$	<i>OperatorLeftScalarMult</i>
$(A * a)(x)$	$A(a * x)$	<i>OperatorRightScalarMult</i>
$(v * f)(x)$	$v * f(x)$	<i>FunctionalLeftVectorMult</i>
$(v * A)(x)$	$v * A(x)$	<i>OperatorLeftVectorMult</i>
$(A * v)(x)$	$A(v * x)$	<i>OperatorRightVectorMult</i>
not available	$A(x) * B(x)$	<i>OperatorPointwiseProduct</i>

There are also a few derived expressions using the above:

Code	Meaning
$(+A)(x)$	$A(x)$
$(-A)(x)$	$(-1) * A(x)$
$(A - B)(x)$	$A(x) + (-1) * B(x)$
$A ** n(x)$	$A(A ** (n-1)(x)), A^1(x) = A(x)$
$(A / a)(x)$	$A((1/a) * x)$
$(A @ B)(x)$	$(A * B)(x)$

Except for composition, operator arithmetic is generally only defined when *Operator.domain* and *Operator.range* are either instances of *LinearSpace* or *Field*.

LinearSpace

The *LinearSpace* represent abstract mathematical concepts of vector spaces. It cannot be used directly but are rather intended to be subclassed by concrete space implementations. The space provide default implementations of the most important vector space operations. See the documentation of the respective classes for more details.

The concept of linear vector spaces in ODL is largely inspired by the [Rice Vector Library \(RVL\)](#).

The abstract *LinearSpace* class is intended for quick prototyping. It has a number of abstract methods which must be overridden by a subclass. On the other hand, it provides automatic error checking and numerous attributes and methods for convenience.

Abstract methods

In the following, the abstract methods are explained in detail.

Element creation `element(inp=None)`

This public method is the factory for the inner *LinearSpaceVector* class. It creates a new element of the space, either from scratch or from an existing data container. In the simplest possible case, it just delegates the construction to the *LinearSpaceVector* class.

If no data is provided, the new element is **merely allocated, not initialized**, thus it can contain *any* value.

Parameters:

inp [object, optional] A container for values for the element initialization

Returns:

element [*LinearSpaceVector*] The new vector

Linear combination `_lincomb(a, x1, b, x2, out)`

This private method is the raw implementation (i.e. without error checking) of the linear combination $\text{out} = a * x1 + b * x2$. *LinearSpace._lincomb* and its public counterpart *LinearSpace.lincomb* are used to cover a range of convenience functions, see below.

Parameters:

a, b [scalars, must be members of the space's `field`] Multiplicative scalar factors for input vector `x1` or `x2`, respectively

x1, x2 [*LinearSpaceVector*] Input vectors

out [*LinearSpaceVector*] Element to which the result of the computation is written

Returns: None

Requirements:

- Aliasing of `x1`, `x2` and `out` **must** be allowed.
- The input vectors `x1` and `x2` **must not** be modified.
- The initial state of the output vector `out` **must not** influence the result.

Underlying scalar field `field`

The public attribute determining the type of scalars which underlie the space. Can be instances of either *RealNumbers* or *ComplexNumbers* (see *Field*).

Should be implemented as a `@property` to make it immutable.

Equality check `__eq__(other)`

LinearSpace inherits this abstract method from *Set*. Its purpose is to check two *LinearSpace* instances for equality.

Parameters:

other [object] The object to compare to

Returns:

equals [bool] True if `other` is the same *LinearSpace*, False otherwise

Distance (optional) `_dist(x1, x2)`

A raw (not type-checking) private method measuring the distance between two vectors `x1` and `x2`.

A space with a distance is called a **metric space**.

Parameters:

x1, x2 [*LinearSpaceVector*] Vectors whose mutual distance to calculate

Returns:

distance [float] The distance between `x1` and `x2`, measured in the space's metric

Requirements:

- `_dist(x, y) == _dist(y, x)`
- `_dist(x, y) <= _dist(x, z) + _dist(z, y)`
- `_dist(x, y) >= 0`
- `_dist(x, y) == 0` (approx.) if and only if `x == y` (approx.)

Norm (optional) `_norm(x)`

A raw (not type-checking) private method measuring the length of a space element `x`.

A space with a norm is called a **normed space**.

Parameters:

`x` [*LinearSpaceVector*] The vector to measure

Returns:

norm [float] The length of `x` as measured in the space's norm

Requirements:

- `_norm(s * x) = |s| * _norm(x)` for any scalar `s`
- `_norm(x + y) <= _norm(x) + _norm(y)`
- `_norm(x) >= 0`
- `_norm(x) == 0` (approx.) if and only if `x == 0` (approx.)

Inner product (optional) `_inner(x, y)`

A raw (not type-checking) private method calculating the inner product of two space elements `x` and `y`.

Parameters:

`x,y` [*LinearSpaceVector*] Vectors whose inner product to calculate

Returns:

inner [float or complex] The inner product of `x` and `y`. If *LinearSpace.field* is the set of real numbers, `inner` is a float, otherwise complex.

Requirements:

- `_inner(x, y) == _inner(y, x)^*` with `*` = complex conjugation
- `_inner(s * x, y) == s * _inner(x, y)` for `s` scalar
- `_inner(x + z, y) == _inner(x, y) + _inner(z, y)`
- `_inner(x, x) == 0` (approx.) if and only if `x == 0` (approx.)

Pointwise multiplication (optional) `_multiply(x1, x2, out)`

A raw (not type-checking) private method multiplying two vectors `x1` and `x2` element-wise and storing the result in `out`.

Parameters:

`x1,x2` [*LinearSpaceVector*] Vectors whose element-wise product to calculate

`out` [*LinearSpaceVector*] Vector to store the result

Returns: None

Requirements:

- `_multiply(x, y, out) <==> _multiply(y, x, out)`
- `_multiply(s * x, y, out) <==> _multiply(x, y, out); out *= s <==> _multiply(x, s * y, out)` for any scalar `s`
- There is a space element `one` with `out` after `_multiply(one, x, out)` or `_multiply(x, one, out)` equals `x`.

Notes

- A normed space is automatically a metric space with the distance function `_dist(x, y) = _norm(x - y)`.
- A Hilbert space (inner product space) is automatically a normed space with the norm function `_norm(x) = sqrt(_inner(x, x))`.
- The conditions on the pointwise multiplication constitute a *unital commutative algebra* in the mathematical sense.

References

See Wikipedia's mathematical overview articles [Vector space](#), [Algebra](#).

Vectorized functions

This section is intended as a small guideline on how to write functions which work with the vectorization machinery by Numpy which is used internally in ODL.

What is vectorization?

In general, *vectorization* means that a function can be evaluated on a whole array of values at once instead of looping over individual entries. This is very important for performance in an interpreted language like python, since loops are usually very slow compared to compiled languages.

Technically, vectorization in Numpy works through the [Universal functions \(ufunc\)](#) interface. It is fast because all loops over data are implemented in C, and the resulting implementations are exposed to Python for each function individually.

How to use Numpy's ufuncs?

The easiest way to write fast custom mathematical functions in Python is to use the [available ufuncs](#) and compose them to a new function:

```
def gaussian(x):
    # Negation, powers and scaling are vectorized, of course.
    return np.exp(-x ** 2 / 2)

def step(x):
    # np.where checks the condition in the first argument and
    # returns the second for `True`, otherwise the third. The
```

```
# last two arguments can be arrays, too.
# Note that also the comparison operation is vectorized.
return np.where(x[0] <= 0, 0, 1)
```

This should cover a very large range of useful functions already (basic arithmetic is vectorized, too!). An even larger list of [special functions](#) are available in the Scipy package.

Usage in ODL

Python functions are in most cases used as input to a discretization process. For example, we may want to discretize a two-dimensional Gaussian function:

```
def gaussian2(x):
    return np.exp(-(x[0]**2 + x[1]**2) / 2)
```

on the rectangle $[-5, 5] \times [-5, 5]$ with 100 pixels in each dimension. The code for this is simply:

```
# Note that the minimum and maximum coordinates are given as
# vectors, not one interval at a time.
discr = odl.uniform_discr([-5, -5], [5, 5], (100, 100))

# This creates an element in the discretized space ``discr``
gaussian_discr = discr.element(gaussian2)
```

What happens behind the scenes is that `discr` creates a [discretization](#) object which has a built-in method `element` to turn continuous functions into discrete arrays by evaluating them at a set of grid points. In the example above, this grid is a uniform sampling of the rectangle by 100 points per dimension.

To make this process fast, `element` assumes that the function is written in a way that not only supports vectorization, but also guarantees that the output has the correct shape. The function receives a [meshgrid](#) tuple as input, in the above case consisting of two vectors:

```
>>> mesh = discr.meshgrid()
>>> mesh[0].shape
(100, 1)
>>> mesh[1].shape
(1, 100)
```

When inserted into the function, the final shape of the output is determined by Numpy's [broadcasting rules](#). For the Gaussian function, Numpy will conclude that the output shape must be $(100, 100)$ since the arrays in `mesh` are added after squaring. This size is the same as expected by the discretization.

If, however, the function does not use all components of its input, the shape will be different:

```
>>> def gaussian_const_x0_bad(x):
...     return np.exp(-x[1]**2 / 2) # no x[0] -> no broadcasting

>>> gaussian_const_x0_bad(mesh).shape
(1, 100)
```

This array is too small for the discretization, and an exception will be raised, stating that this function cannot be discretized.

The solution to this issue is rather simple: just make sure that all components are used such that the broadcasting rules are triggered:

```
>>> def gaussian_const_x0_good(x):
...     return np.exp(-x[1]**2 / 2) + 0 * x[0] # broadcasting
```

```
>>> gaussian_const_x0_good(mesh).shape
(100, 100)
```

Further reading

Scipy Lecture notes on Numpy

2.2 Contributing to odl

2.2.1 How to document

ODL is documented using `sphinx` and a modified version of `numpydoc`. An example documentation is given below

```
class MyClass(object):

    """Calculate important things.

    The first line summarizes the class, after that comes a blank
    line followed by a more detailed description (both optional).
    Confine the docstring to 72 characters per line. In general, try
    to follow `PEP257`_ in the docstring style.

    Docstrings can have sections with headers, signaled by a
    single-dash underline, e.g. "References". Check out
    `Numpydoc`_ for the recognized section labels.

    References
    -----
    .. _PEP257: https://www.python.org/dev/peps/pep-0257/
    .. _Numpydoc: https://github.com/numpy/numpy/blob/master/doc/\
        HOWTO_DOCUMENT.rst.txt
    """

    def __init__(self, c, parameter=None):
        """Initializer doc goes here.

        Parameters
        -----
        c : `float`
            Constant to scale by
        parameter : `float`, optional
            Some extra parameter
        """
        self.c = c
        self.parameter = parameter

    def my_method(self, x, y):
        """Calculate ``c * (x + y)``.

        The first row is a summary, after that goes
        a more detailed description.

        Parameters
        -----
```

```
x : `float`
    First summand
y : `float`
    Second summand

Returns
-----
scaled_sum : `float`
    Result of ``c * (x + y)``

Examples
-----
Examples should be working pieces of code and are checked with
``doctest`` for consistent output.

>>> obj = MyClass(5)
>>> obj(3, 5)
8.0
"""
return self.c * (x + y)
```

Some short tips

- Text within backticks: ``some_target`` will create a link to the target.
- Make sure that the first line is short and descriptive.
- Examples are often better than long descriptions.

Quick summary of [PEP257](#)

- Write docstrings always with triple double quotes `"""`, even one-liners
- Class docstrings are separated from the class definition line by a blank line, functions and methods begin directly in the next line.
- Use imperative style (“Calculate”, not “Calculates”) in the summary (=first) line and end it with a full stop. Do not add a space after the opening triple quotes.
- For one-liners: put the closing quotes on the same line. Otherwise: make a new line for the closing quotes.
- Document at least all *public* methods and attributes.

Advanced

This section covers advanced topics for developers that need to change how the doc works.

Re-generating the doc

Autosummary currently does not support nested modules, so to handle this we auto-generate `.rst` files for each module. This is done using the `doc/source/generate_doc.py` script.

Modifications to numpydoc

Numpydoc has been modified in the following ways:

- The numpy sphinx domain has been removed.

- More `extra_public_methods` added.
- `:autoclass:` summaries now link to full name, which allows subclassing between packages.

2.2.2 Working with *odl* source code

Contents:

Introduction

These pages describe a [git](#) and [github](#) workflow for the *odl* project.

There are several different workflows here, for different ways of working with *odl*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see [git resources](#).

Install git

Overview

Debian / Ubuntu	<code>sudo apt-get install git</code>
Fedora	<code>sudo yum install git</code>
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the [git](#) page for the most recent information.

Have a look at the github install help pages available from [github help](#)

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Following the latest source

These are the instructions if you just want to follow the latest *odl* source, but you don't need to do any development for now.

The steps are:

- [Install git](#)
- get local copy of the [odl github](#) git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/odlgroupp/odl.git
```

You now have a copy of the code tree in the new `odl` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd odl
git pull
```

The tree in `odl` will now have the latest changes from the initial repository.

Making a patch

You've discovered a bug or something else you want to change in `odl` .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/odlgroupp/odl.git
# make a branch for your patching
cd odl
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the [odl mailing list](#) — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the [odl](#) repository:

```
git clone git://github.com/odlgroup/odl.git
cd odl
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#).

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the [odl mailing list](#).

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the [odl](#) repository on github — *Making your own copy (fork) of odl*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
```

```
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/odl.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

Git for development

Contents:

Making your own copy (fork) of odl

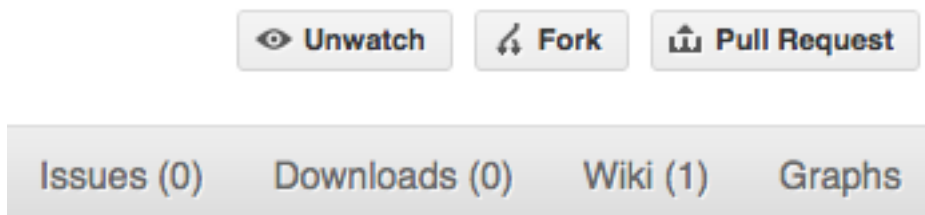
You need to do this only once. The instructions here are very similar to the instructions at <http://help.github.com/forking/> — please see that page for more detail. We’re repeating some of it here just to give the specifics for the odl project, and to suggest some default names.

Set up and configure a github account If you don’t have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the [Generating SSH keys help on github help](#).

Create your own forked copy of odl

1. Log into your github account.
2. Go to the odl github home at [odl github](#).
3. Click on the *fork* button:



Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of odl.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of odl*.

Overview

```
git clone git@github.com:your-user-name/odl.git
cd odl
git remote add upstream git://github.com/odlgroup/odl.git
```


In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/odl.git`
2. Investigate. Change directory to your new repo: `cd odl`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the master branch, and that you also have a remote connection to origin/master. What remote repository is remote/origin? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream [odl github](#) repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd odl
git remote add upstream git://github.com/odlgroup/odl.git
```

upstream here is just the arbitrary name we're using to refer to the main [odl](#) repository at [odl github](#).

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v` show, giving you something like:

```
upstream    git://github.com/odlgroup/odl.git (fetch)
upstream    git://github.com/odlgroup/odl.git (push)
origin      git@github.com:your-user-name/odl.git (fetch)
origin      git@github.com:your-user-name/odl.git (push)
```

Configure git

Overview Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
  name = Your Name
  email = you@yourdomain.example.com

[alias]
  ci = commit -a
  co = checkout
  st = status
  stat = status
  br = branch
  wdiff = diff --color-words

[core]
  editor = vim
```

```
[merge]
    summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email It is good practice to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an alias section in your `.gitconfig` file with contents like this:

```
[alias]
    ci = commit -a
    co = checkout
    st = status -a
    stat = status -a
```

```
br = branch
wdiff = diff --color-words
```

Editor You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging To enforce summaries when doing merges (~/.gitconfig file again):

```
[merge]
  log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output This is a very nice alias to get a fancy log output; it should go in the alias section of your .gitconfig file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%a'
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Matthias]
* d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks ago) [Hugo]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran W]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a ca
* 376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
|\
| * b605216 - updated joshua example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| * 956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the [odl](#) repository, by following [Making your own copy \(fork\) of odl](#). You have [Set up your fork](#). You have configured git by following [Configure git](#). Now you are ready for some real work.

Workflow summary In what follows we’ll refer to the upstream odl master branch, as “trunk”.

- Don’t use your master branch for anything. Consider deleting it.

- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — “one task, one branch” ([ipython git workflow](#)).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider [Rebasing on trunk](#)
- Ask on the [odl mailing list](#) if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you’ve done, and why you did it.

See [linux git workflow](#) and [ipython git workflow](#) for some explanation.

Consider deleting your master branch It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See [deleting master on github](#) for details.

Update the mirror of trunk First make sure you have done [Linking your repository to the upstream repo](#).

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don’t have, and set the remote branches to point to the right commit. For example, ‘trunk’ is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called ‘feature branches’.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public [github](#) fork of [odl](#). To do this, you [git push](#) this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git \geq 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see [git status](#)). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` ([git diff](#)).
4. Add any new files to version control `git add new_file_name` (see [git add](#)).
5. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#) — and the helpful use-case description in the [tangled working copy problem](#). The [git commit](#) manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see [git push](#)).

The commit message Bear in mind that the commit message will be part of the history of the repository, shown by typing `git log`, so good messages will make the history searchable. Don't see the commit message as an annoyance, but rather as an important part of your contribution.

We appreciate if you follow the following style:

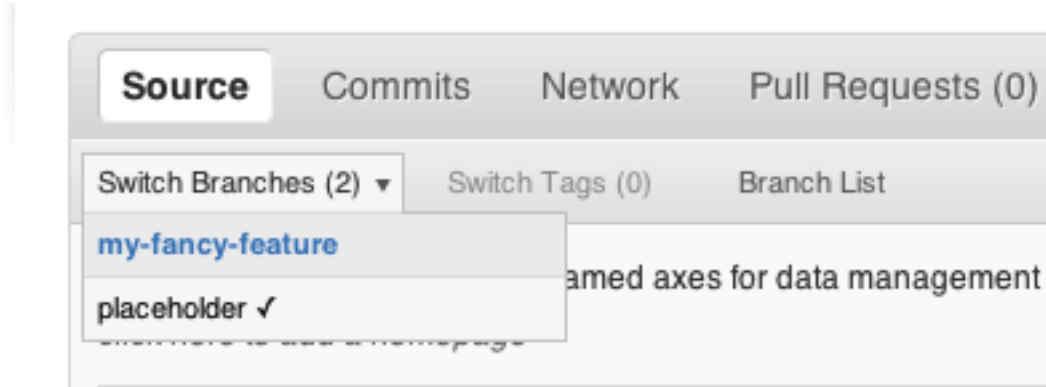
1. Start your commit with an [acronym](#), e.g., BUG, TST or STY to indicate what kind of modification you make.
2. Write a one-line summary of your modification no longer than 50 characters. If you have a hard time summarizing your changes, maybe you need to split up the commit into parts.

Use imperative style, i.e. write `add super feature` or `fix horrific bug` rather than `added`, `fixed` This saves two characters for something else.

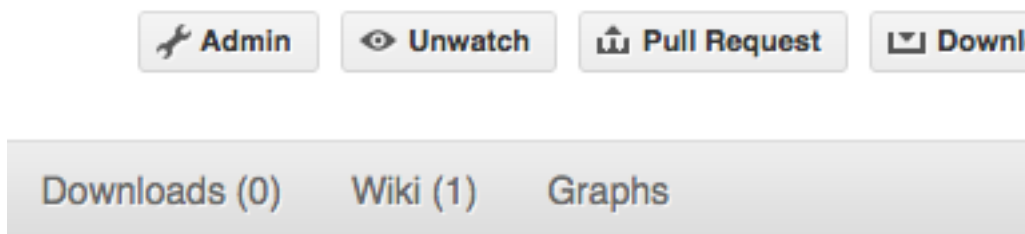
Don't use markdown. You can refer to issues by writing `#12`. You can even close an issue by writing `closes #12`, but do that only if you are sure.
3. (optional) Write an extended summary. Describe why these changes are necessary and what the new code does better than the old one. You can use markdown here, i.e. create lists, tables, ...

Ask for your changes to be reviewed or merged When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/odl`.
2. Use the ‘Switch Branches’ dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the ‘Pull request’ button:



Enter a title for the set of changes, and some explanation of what you’ve done. Say if there is anything you’d like particular attention for - like a complicated change or some code you are not happy with.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

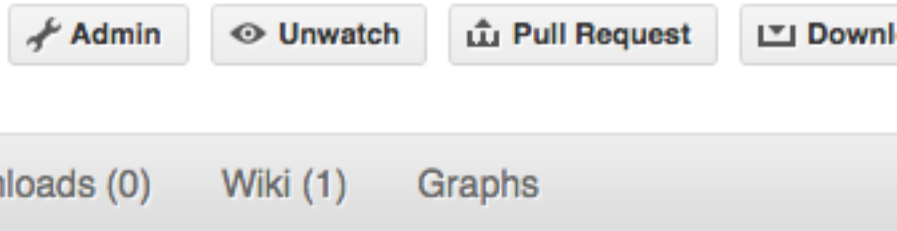
(Note the colon `:` before `test-branch`. See also: <http://github.com/guides/remove-a-remote-branch>)

Several people sharing a single repository If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork odl into your account, as from *[Making your own copy \(fork\) of odl](#)*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/odl`

Click on the ‘Admin’ button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@github.com:your-user-name/odl.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the [network graph visualizer](#) for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```

      A---B---C cool-feature
      /
D---E---F---G trunk

```

At this stage you consider merging trunk into your feature branch, and you remember that this page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```

      A'--B'--C' cool-feature
      /
D---E---F---G trunk

```

See [rebase without tears](#) for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at [Recovering from mess-ups](#).

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the [git rebase](#) man page for some instructions at the end of the “Description” section. There is some related help on merging in the git user manual - see [resolving a merge](#).

Recovering from mess-ups Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:


```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2declac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and 6ad92e5 is the last commit in the cool-feature branch. Suppose we want to make the following changes:

- Rewrite the commit message for 13d7934 to something more sensible.
- Combine the commits 2declac, a815645, eadc391 into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2declac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2declac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for 13d7934, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained [above](#).

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in [Development workflow](#).

The instructions in [Linking your repository to the upstream repo](#) add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:odlgroup/odl.git
git fetch upstream-rw
```

Integrating changes Let's say you have some changes that need to go into trunk (upstream-rw/master).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/odl.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

git resources

Tutorials and summaries

- [github help](#) has an excellent series of how-to guides.
- [learn.github](#) has an excellent series of tutorials
- The [pro git book](#) is a good in-depth book on git.
- A [git cheat sheet](#) is a page giving summaries of common commands.
- The [git user manual](#)
- The [git tutorial](#)
- The [git community book](#)
- [git ready](#) — a nice series of tutorials
- [git casts](#) — video snippets giving git how-tos.
- [git magic](#) — extended introduction with intermediate detail
- The [git parable](#) is an easy read explaining the concepts behind git.
- [git foundation](#) expands on the [git parable](#).
- Fernando Perez' git page — [Fernando's git page](#) — many links and tips
- A good but technical page on [git concepts](#)
- [git svn crash course](#): git for those of us used to [subversion](#)

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on [git management](#)
- Linus Torvalds on [linux git workflow](#) . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- `git add`
- `git branch`
- `git checkout`
- `git clone`
- `git commit`
- `git config`
- `git diff`
- `git log`
- `git pull`
- `git push`
- `git remote`
- `git status`

2.3 Frequently asked questions

Abbreviations: **Q**uestion – **P**roblem – **S**olution

2.3.1 General errors

1. **Q:** When importing `odl`, the following error is shown:

```
File "/path/to/odl/odl/__init__.py", line 36
    from . import diagnostics
ImportError: cannot import diagnostics
```

P: When you for the first time import (=execute) a module or execute a script, a `bytecode` file is created, basically to speed up execution next time. If you installed `odl` with `pip -e (--editable)`, these files can interfere with changes to your codebase.

S: Delete the bytecode files. In a standard GNU/Linux shell, you can simply invoke (in your `odl` working directory)

```
find . -name *.pyc | xargs rm
```

1. **Q:** When adding two vectors, the following error is shown:

```
TypeError: unsupported operand type(s) for -: 'DiscreteLpVector' and 'DiscreteLpVector'
```

P: The vectors you are trying to add are not in the same space, for example the following code gives the error

```
>>> x = odl.uniform_discr(0, 1, 10).one()
>>> y = odl.uniform_discr(0, 1, 11).one()
>>> x - y
```

In this case, the problem is that the vectors have a different number of elements. Other possible issues include that they are discretizations of different sets, have different data types (*dtype*), or implementation (for example cuda/cpu).

S: The vectors need to somehow be cast to the same space. How to do this depends on the problem at hand. To find what the issue is, inspect the `space` properties of both vectors. For example in the above we see that the issue lies in the number of discretization points

```
>>> x.space
odl.uniform_discr(0, 1, 10)
>>> y.space
odl.uniform_discr(0, 1, 11)
```

- In the case of spaces being discretizations of different underlying spaces, a transformation of some kind has to be applied (for example by using an operator). In general, errors like this indicates a conceptual issue with the code, for example a “we identify X with Y” step has been omitted.
- If the `dtype` or `impl` do not match, they need to be cast to each one of the others. The most simple way to do this is by using the `DiscreteIpVector.astype` method.

2.3.2 Errors related to Python 2/3

1. **Q:** I follow your recommendation to call `super().__init__(dom, ran)` in the `__init__()` method of `MyOperator`, but I get the following error:

```
File <...>, line ..., in __init__
    super().__init__(dom, ran)

TypeError: super() takes at least 1 argument (0 given)
```

P: The `super()` function in Python 2 has to be called with a type as first argument, whereas in Python 3, the type argument is optional and usually not needed.

S: We recommend to include `from builtins import super` in your module to backport the new Py3 `super()` function. This way, your code will run in both Python 2 and 3.

2.3.3 Usage

1. **Q:** I want to write an *Operator* with two input arguments, for example

$$op(x, y) := x + y$$

However, ODL only supports single arguments. How do I do this?

P: Mathematically, such an operator is defined as

$$\mathcal{A} : \mathcal{X}_1 \times \mathcal{X}_2 \rightarrow \mathcal{Z}$$

ODL adheres to the strict definition of this and hence only takes one parameter $x \in \mathcal{X}_1 \times \mathcal{X}_2$. This product space element x is then a tuple of elements $x = (x_1, x_2), x_1 \in \mathcal{X}_1, x_2 \in \mathcal{X}_2$.

S: Make the domain of the operator a *ProductSpace* if \mathcal{X}_1 and \mathcal{X}_2 are *LinearSpace*’s, or a *CartesianProduct* if they are mere *Set*’s. Mathematically, this corresponds to

$$op([x, y]) := x + y$$

Of course, a number of input arguments larger than 2 can be treated analogously.

2.4 Glossary

array-like Any data structure which can be converted into a `numpy.ndarray` by the `numpy.array` constructor. Includes all `NtuplesBaseVector` based classes.

dtype Short for data type, indicates the way data is represented internally. For example `float32` means 32-bit floating point numbers. See `numpy.dtype` for more details.

discretization Structure to handle the mapping between abstract objects (e.g. functions) and concrete, finite realization. It encompasses an abstract *Set*, a finite data container (`NtuplesBaseVector` in general) and the mappings between them, *restriction* and *extension*.

domain Set of elements to which an operator can be applied.

element Saying that `x` is an element of a given *Set* `my_set` means that `x in my_set` evaluates to `True`. The term is typically used as “element of <set>” or “<set>” element. When referring to a *LinearSpace* like, e.g., *DiscreteLp*, an element is of the corresponding type *LinearSpaceVector*, i.e. *DiscreteLpVector* in the above example. Elements of a set can be created by the `Set.element` method.

element-like Any data structure which can be converted into an *element* of a *Set* by the `Set.element` method. For example, an `Rn(3)` element-like is any *array-like* object with 3 real entries.

Example: `'DiscreteLp'` element-like means that `DiscreteLp.element` can create a *DiscreteLpVector* from the input.

extension Operator in a *discretization* mapping a concrete (finite-dimensional) object to an abstract (infinite-dimensional) one. Example: *LinearInterpolation*.

in-place evaluation Operator evaluation method which uses an existing data container to store the result. Usually more efficient than *out-of-place evaluation* since no new memory is allocated and no data is copied.

meshgrid Tuple of arrays defining a tensor grid by all possible combinations of entries, one from each array. In 2 dimensions, for example, the arrays `[1, 2]` and `[-1, 0, 1]` define the grid points `(1, -1)`, `(1, 0)`, `(1, 1)`, `(2, -1)`, `(2, 0)`, `(2, 1)`.

operator Mathematical notion for a mapping between arbitrary vector spaces. This includes the important special case of an operator taking a (discretized) function as an input and returning another function. For example, the Fourier Transform maps a function to its transformed version. Operators of this type are the most prominent use case in ODL. See *the in-depth guide on operators* for details on their implementation.

order Ordering of the axes in a multi-dimensional array with linear (one-dimensional) storage. For C ordering (`'C'`), the last axis has smallest stride (varies fastest), and the first axis has largest stride (varies slowest). Fortran ordering (`'F'`) is the exact opposite.

out-of-place evaluation Operator evaluation method which creates a new data container to store the result. Usually less efficient than *in-place evaluation* since new memory is allocated and data needs to be copied.

range Set of elements to which an operator maps, i.e. in which the result of an operator evaluation lies.

restriction Operator in a *discretization* mapping an abstract (infinite-dimensional) object to a concrete (finite-dimensional) one. Example: *PointCollocation*.

vectorization Ability of a function to be evaluated on a grid in a single call rather than looping over the grid points. Vectorized evaluation gives a huge performance boost compared to Python loops (at least if there is no JIT) since loops are implemented in optimized C code.

The vectorization concept in ODL differs slightly from the one in NumPy in that arguments have to be passed as a single tuple rather than a number of (positional) arguments. See `numpy.vectorization` for more details.

2.5 Mathematics behind ODL

This section explains the mathematical concepts on which ODL is built.

2.5.1 Discretizations

Mathematical background

In mathematics, the term *discretization* stands for the transition from abstract, continuous, often infinite-dimensional objects to concrete, discrete, finite-dimensional counterparts. We define discretizations as tuples encompassing all necessary aspects involved in this transition. Let \mathcal{X} be an arbitrary set, \mathbb{F}^n be the set of n -tuples where each component lies in \mathbb{F} . We define two mappings

$$\begin{aligned}\mathcal{R}_{\mathcal{X}} : \mathcal{X} &\rightarrow \mathbb{F}^n, \\ \mathcal{E}_{\mathcal{X}} : \mathbb{F}^n &\rightarrow \mathcal{X},\end{aligned}$$

which we call *restriction* and *extension*, respectively. Then, the discretization of \mathcal{X} with respect to \mathbb{F}^n and the above operators is defined as the tuple

$$\mathcal{D}(\mathcal{X}) = (\mathcal{X}, \mathbb{F}^n, \mathcal{R}_{\mathcal{X}}, \mathcal{E}_{\mathcal{X}}).$$

The following abstract diagram visualizes a discretization:

$$\begin{array}{ccc} & \mathcal{X} & \\ \mathcal{R}_{\mathcal{X}} \downarrow & & \uparrow \mathcal{E}_{\mathcal{X}} \\ & \mathbb{F}^n & \\ & \mathcal{Y} & \end{array}$$

Example

Let $\mathcal{X} = C([0, 1])$ be the space of real-valued continuous functions on the interval $[0, 1]$, and let $x_1 < \dots < x_n$ be ordered sampling points in $[0, 1]$.

Restriction operator:

We define the *grid collocation operator* as

$$\begin{aligned}\mathcal{C} : \mathcal{X} &\rightarrow \mathbb{R}^n, \\ \mathcal{C}(f) &:= (f(x_1), \dots, f(x_n)).\end{aligned}$$

The abstract object in this case is the input function f , and the operator evaluates this function at the given points, resulting in a vector in \mathbb{R}^n .

This operator is implemented as `PointCollocation`.

Extension operator:

Let discrete values $\bar{f} \in \mathbb{R}^n$ be given. Consider the linear interpolation of those values at a point $x \in [0, 1]$:

$$\begin{aligned}I(\bar{f}; x) &:= (1 - \lambda(x))f_i + \lambda(x)f_{i+1}, \\ \lambda(x) &= \frac{x - x_i}{x_{i+1} - x_i},\end{aligned}$$

where i is the index such that $x \in [x_i, x_{i+1})$.

Then we can define the linear interpolation operator as

$$\begin{aligned}\mathcal{L} : \mathbb{R}^n &\rightarrow C([0, 1]), \\ \mathcal{L}(\bar{f}) &:= I(\bar{f}; \cdot),\end{aligned}$$

where $I(\bar{f}; \cdot)$ stands for the function $x \mapsto I(\bar{f}; x)$.

Hence, this operator maps the finite array $\bar{f} \in \mathbb{R}^n$ to the abstract interpolating function $I(\bar{f}; \cdot)$.

This interpolation scheme is implemented in the `LinearInterpolation` operator.

Useful Wikipedia articles

- [Discretization](#)

2.5.2 Transformations

This section contains the mathematical descriptions of (integral) transforms implemented in ODL.

Fourier Transform

Background

Definition and basic properties The Fourier Transform (FT) of a function f belonging to the [Lebesgue Space](#) $L^1(\mathbb{R})$ is defined as

$$\widehat{f}(\xi) = \mathcal{F}(f)(\xi) = (2\pi)^{-\frac{1}{2}} \int_{\mathbb{R}} f(x) e^{-ix\xi} dx. \quad (2.1)$$

By unique continuation, the bounded FT operator can be [extended](#) to $L^p(\mathbb{R})$ for $p \in [1, 2]$, yielding a mapping

$$\mathcal{F} : L^p(\mathbb{R}) \longrightarrow L^q(\mathbb{R}), \quad q = \frac{p}{p-1},$$

where q is the conjugate exponent of p (for $p = 1$ one sets $q = \infty$). Finite exponents larger than 2 also allow the extension of the operator but require the notion of [Distributions](#) to characterize its range. See [\[SW1971\]](#) for further details.

The inverse of \mathcal{F} on its range is given by the formula

$$\widetilde{\phi}(x) = \mathcal{F}^{-1}(\phi)(x) = (2\pi)^{-\frac{1}{2}} \int_{\mathbb{R}} \phi(\xi) e^{ix\xi} d\xi. \quad (2.2)$$

For $p = 2$, the conjugate exponent is $q = 2$, and the FT is a unitary operator on $L^2(\mathbb{R})$ according to [Parseval's Identity](#)

$$\int_{\mathbb{R}} |f(x)|^2 dx = \int_{\mathbb{R}} |\widetilde{f}(\xi)|^2 d\xi,$$

which implies that its adjoint is its inverse, $\mathcal{F}^* = \mathcal{F}^{-1}$.

Further Properties

$$\begin{aligned} \mathcal{F}^{-\infty}(\phi) &= \mathcal{F}(\check{\phi}) = \mathcal{F}(\phi)(-\cdot) = \overline{\mathcal{F}(\phi)} = \mathcal{F}^3(\phi), & \check{\phi}(x) &= \phi(-x), \\ \mathcal{F}(f(\cdot - b))(\xi) &= e^{-ib\xi} \widehat{f}(\xi), \\ \mathcal{F}(f(a\cdot))(\xi) &= a^{-1} \widehat{f}(a^{-1}\xi), \\ \frac{d}{d\xi} \widehat{f}(\xi) &= \mathcal{F}(-ixf)(\xi) \\ \mathcal{F}(f')(\xi) &= i\xi \widehat{f}(\xi). \end{aligned} \quad (2.3)$$

The first identity implies in particular that for real-valued f , it is $\overline{\mathcal{F}(\phi)}(\xi) = \mathcal{F}(\phi)(-\xi)$, i.e. the FT is completely known already from the its values in a half-space only. This property is later exploited to reduce storage.

In d dimensions, the FT is defined as

$$\mathcal{F}(f)(\xi) = (2\pi)^{-\frac{d}{2}} \int_{\mathbb{R}^d} f(x) e^{-ix^T \xi} dx$$

with the usual inner product $x^T \xi = \sum_{k=1}^d x_k \xi_k$ in \mathbb{R}^d . The identities (2.3) also hold in this case with obvious modifications.

Discretized Fourier Transform

General case The approach taken in ODL for the discretization of the FT follows immediately from the way *Discretizations* are defined, but the original inspiration for it came from the book [Pre+2007], Section 13.9 “Computing Fourier Integrals Using the FFT”.

Discretization of the Fourier transform operator means evaluating the Fourier integral (2.1) on a discretized function

$$f(x) = \sum_{k=0}^{n-1} f_k \phi_k(x) \quad (2.4)$$

with coefficients $\bar{f} = (f_0, \dots, f_{n-1}) \in \mathbb{C}^n$ and functions $\phi_0, \dots, \phi_{n-1}$. This approach follows from the way , but can be We consider in particular functions generated from a single kernel ϕ via

$$\phi_k(x) = \phi\left(\frac{x - x_k}{s_k}\right),$$

where $x_0 < \dots < x_{n-1}$ are sampling points and $s_k > 0$ scaling factors. Using the shift and scaling properties in (2.3) yields

$$\hat{f}(\xi) = \sum_{k=0}^{n-1} f_k \widehat{\phi_k}(\xi) = \sum_{k=0}^{n-1} f_k s_k \widehat{\phi}(s_k \xi) e^{-ix_k \xi}. \quad (2.5)$$

There exist methods for the fast approximation of such sums for a general choice of frequency samples ξ_m , e.g. *NFFT*.

Regular grids For regular grids

$$x_k = x_0 + ks, \quad \xi_j = \xi_0 + j\sigma, \quad (2.6)$$

the evaluation of the integral can be written in the form which uses trigonometric sums as *computed in FFTW* or in *Numpy*:

$$\hat{f}_j = \sum_{k=0}^{n-1} f_k e^{-i2\pi jk/n}. \quad (2.7)$$

Hence, the Fourier integral evaluation can be built around established libraries with simple pre- and post-processing steps.

With regular grids, the discretized integral (2.5) evaluated at $\xi = \xi_j$, can be expanded to

$$\hat{f}(\xi_j) = s \widehat{\phi}(s \xi_j) e^{-ix_0 \xi_j} \sum_{k=0}^{n-1} f_k e^{-iks \xi_0} e^{-ijk s \sigma}$$

To reach the form (2.7), the factor depending on both indices j and k must agree with the corresponding factor in the FFT sum. This is achieved by setting

$$\sigma = \frac{2\pi}{ns}, \quad (2.8)$$

finally yielding the representation

$$\hat{f}_j = \hat{f}(\xi_j) = s\hat{\phi}(s\xi_j)e^{-ix_0\xi_j} \sum_{k=0}^{n-1} f_k e^{-iks\xi_0} e^{-i2\pi jk/n}. \quad (2.9)$$

Choice of ξ_0 There is a certain degree of freedom in the choice of the most negative frequency ξ_0 . Usually one wants to center the Fourier space grid around zero since most information is typically concentrated there. Point-symmetric grids are the standard choice, however sometimes one explicitly wants to include (for even n) or exclude (for odd n) the zero frequency from the grid, which is achieved by shifting the frequency x_{i_0} by $-\sigma/2$. This results in two possible choices

$$\begin{aligned} \xi_{0,n} &= -\frac{\pi}{s} + \frac{\pi}{sn} \quad (\text{no shift}), \\ \xi_{0,s} &= -\frac{\pi}{s} \quad (\text{shift}). \end{aligned}$$

For the shifted frequency, the pre-processing factor in the sum in (2.9) can be simplified to

$$e^{-iks\xi_0} = e^{ik\pi} = (-1)^k,$$

which is favorable for real-valued input \bar{f} since this first operation preserves this property. For half-complex transforms, shifting is required.

The factor $\hat{\phi}(s\xi_j)$ In (2.9), the FT of the kernel ϕ appears as post-processing factor. We give the explicit formulas for the two standard discretizations currently used in ODL, which are nearest neighbor interpolation

$$\phi_{nn}(x) = \begin{cases} 1, & \text{if } -1/2 \leq x < 1/2, \\ 0, & \text{else,} \end{cases}$$

and linear interpolation

$$\phi_{lin}(x) = \begin{cases} 1 - |x|, & \text{if } -1 \leq x \leq 1, \\ 0, & \text{else.} \end{cases}$$

Their Fourier transforms are given by

$$\begin{aligned} \widehat{\phi_{nn}}(\xi) &= (2\pi)^{-1/2} \text{sinc}(\xi/2), \\ \widehat{\phi_{lin}}(\xi) &= (2\pi)^{-1/2} \text{sinc}^2(\xi/2). \end{aligned}$$

Since their arguments $s\xi_j = s\xi_0 + 2\pi/n$ lie between $-\pi$ and π , these functions introduce only a slight taper towards higher frequencies given the fact that the first zeros lie at $\pm 2\pi$.

Inverse transform According to (2.2), the inverse of the continuous Fourier transform is given by the same formula as the forward transform (2.1), except for a switched sign in the complex exponential. Hence, this operator can rather be viewed as a variation of the forward FT, and it is implemented via a `sign` parameter in `FourierTransform`.

The inverse of the discretized formula (2.9) is instead gained directly using the identity

$$\begin{aligned} \sum_{j=0}^{N-1} e^{i2\pi \frac{(l-k)j}{N}} &= \sum_{j=0}^{N-1} \left(e^{i2\pi \frac{(l-k)}{N}} \right)^j = \begin{cases} N, & \text{if } l = k, \\ \frac{1 - e^{i2\pi(l-k)}}{1 - e^{i2\pi(l-k)/N}} = 0, & \text{else} \end{cases} \\ &= N \delta_{l,k}. \end{aligned} \quad (2.10)$$

By dividing (2.9) with the factor

$$\alpha_j = s\widehat{\psi}(s\xi_j) e^{-ix_0\xi_j}$$

before the sum, multiplying with the exponential factor $e^{i2\pi\frac{l_j}{N}}$ and summing over j , the coefficients f_k can be recovered:

$$\begin{aligned} \sum_{j=0}^{N-1} \hat{f}_j \frac{1}{\alpha_j} e^{i2\pi\frac{l_j}{N}} &= \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \bar{f}_k e^{-i2\pi\frac{jk}{N}} e^{i2\pi\frac{l_j}{N}} \\ &= \sum_{k=0}^{N-1} \bar{f}_k N \delta_{l,k} \\ &= N \bar{f}_l. \end{aligned}$$

Hence, the inversion formula for the discretized FT reads as

$$f_k = e^{iks\xi_0} \frac{1}{N} \sum_{j=0}^{N-1} \hat{f}_j \frac{1}{s\widehat{\psi}(s\xi_j)} e^{ix_0\xi_j} e^{i2\pi\frac{kj}{N}}, \quad (2.11)$$

which can be calculated in the same manner as the forward FT, basically by switching the roles of pre- and post-processing steps and flipping the sign in the complex exponentials.

Useful Wikipedia articles

- [Fourier Transform](#)
- [Lebesgue Space](#)
- [Distributions](#)
- [Parseval's Identity](#)

2.6 Release Notes

2.6.1 ODL 0.2.2 Release Notes (2016-03-11)

From this release on, ODL can be installed through `pip` directly from the Python package index.

2.6.2 ODL 0.2.1 Release Notes (2016-03-11)

Fix for the version number in `setup.py`.

2.6.3 ODL 0.2 Release Notes (2016-03-11)

This release adds the functionality of the **Fourier Transform** in arbitrary dimensions. The operator comes in two different flavors: the “bare”, trigonometric-sum-only [Discrete Fourier Transform](#) and the discretization of the continuous [Fourier Transform](#).

New Features

Fourier Transform (FT)

The FT is an *operator* mapping a function to its transformed version (shown for 1d):

$$\hat{f}(\xi) = \mathcal{F}(f)(\xi) = (2\pi)^{-\frac{1}{2}} \int_{\mathbb{R}} f(x) e^{-ix\xi} dx, \quad \xi \in \mathbb{R}.$$

This implementation acts on discretized functions and accounts for scaling and shift of the underlying grid as well as the type of discretization used. Supported backends are [Numpy's FFTPACK based transform](#) and [pyFFTW](#) (Python wrapper for [FFTW](#)). The implementation has full support for the wrapped backends, including

- Forward and backward transforms,
- Half-complex transforms, i.e. real-to-complex transforms where roughly only half of the coefficients need to be stored,
- Partial transforms along selected axes,
- Computation of efficient FFT plans (pyFFTW only).

Discrete Fourier Transform (DFT)

This operator merely calculates the trigonometric sum

$$\hat{f}_j = \sum_{k=0}^{n-1} f_k e^{-i2\pi jk/n}, \quad j = 0, \dots, n-1$$

without accounting for shift and scaling of the underlying grid. It supports the same features of the wrapped backends as the FT.

Further additions

- The `weighting` attribute in `FnBase` is now public and can be used to initialize a new space.
- The `FnBase` classes now have a `default_dtype` static method.
- A `discr_sequence_space` has been added as a simple implementation of finite sequences with multi-indexing.
- `DiscreteLp` and `FunctionSpace` elements now have `real` and `imag` with setters as well as a `conj()` method.
- `FunctionSpace` explicitly handles output data type and allows this attribute to be chosen during initialization.
- `FunctionSpace`, `FnBase` and `DiscreteLp` spaces support creation of a copy with different data type via the `astype()` method.
- New `conj_exponent()` utility to get the conjugate of a given exponent.

Improvements

- Handle some not-so-unlikely corner cases where vectorized functions don't behave as they should. The main issue was the way Python 2 treats comparisons of tuples against scalars (Python 3 raises an exception which is correctly handled by the subsequent code). In Python 2, the following happens:

```

>>> t = ()
>>> t > 0
True
>>> t = (-1,)
>>> t > 0
True

```

This is especially unfortunate if used as `t[t > 0]` in 1d functions, when `t` is a *meshgrid* sequence (of 1 element). In this case, `t > 0` evaluates to `True`, which is treated as 1 in the index expression, which in turn will raise an `IndexError` since the sequence has only length one. This situation is now properly caught.

- `x ** 0` evaluates to the `one()` space element if implemented.

Changes

- Move *fast_1d_tensor_mult* to the `numerics.py` module.

2.6.4 ODL 0.1 Release Notes (2016-03-08)

First official release.

2.7 References

2.8 odl

ODL is a functional analysis library with a focus on discretization.

ODL supports abstract sets, linear vector spaces defined on such and Operators/Functionals defined on these sets. It is intended to be used to write general code and facilitate code reuse.

Modules

2.8.1 diagnostics

Automated tests for ODL.

Modules

examples

Functions for generating standardized examples in spaces.

Functions

<i>samples</i> (*sets)	Generate some samples from the given sets.
<i>scalar_examples</i> (field)	Generate example scalars in <i>field</i> .
<i>vector_examples</i> (space)	Generate example vectors in <i>space</i> .

samples

`odl.diagnostics.examples.samples(*sets)`

Generate some samples from the given sets.

Currently supports vectors according to *vector_examples* and scalars according to *scalar_examples*.

Parameters**sets* : *Set* instance(s)

Return*samples* : generator

Generator that yields tuples of examples from the sets.

scalar_examples

`odl.diagnostics.examples.scalar_examples(field)`

Generate example scalars in field.

Parameters*field* : *Field*

The field to generate examples from

Return*examples* : generator

Yields elements in field

vector_examples

`odl.diagnostics.examples.vector_examples(space)`

Generate example vectors in space.

Parameters*space* : *LinearSpace*

The space to generate examples from

Return*examples* : generator

Yields tuples (string, *LinearSpaceVector*) where string is a short description of the vector

operator

Standardized tests for *Operator*'s.

Classes

OperatorTest(operator[, operator_norm]) Automated tests for *Operator* implementations.

OperatorTest

class `odl.diagnostics.operator.OperatorTest(operator, operator_norm=None)`

Bases: object

Automated tests for *Operator* implementations.

This class allows users to automatically test various features of an *Operator* such as linearity and the adjoint

definition.

Methods

<code>__eq__</code>	Return <code>self==value</code> .
<code>adjoint()</code>	Verify that <code>Operator.adjoint</code> works appropriately.
<code>derivative([step])</code>	Verify that <code>Operator.derivative</code> works appropriately.
<code>linear()</code>	Verify that the operator is actually linear.
<code>norm()</code>	Estimate the operator norm of the operator.
<code>run_tests()</code>	Run all tests on this operator.
<code>self_adjoint()</code>	Verify $(Ax, y) = (x, Ay)$

OperatorTest.adjoint

`OperatorTest.adjoint()`

Verify that `Operator.adjoint` works appropriately.

References

Wikipedia article on [Adjoint](#).

OperatorTest.derivative

`OperatorTest.derivative(step=0.0001)`

Verify that `Operator.derivative` works appropriately.

References

Wikipedia article on [Derivative](#). Wikipedia article on [Frechet derivative](#).

OperatorTest.linear

`OperatorTest.linear()`

Verify that the operator is actually linear.

OperatorTest.norm

`OperatorTest.norm()`

Estimate the operator norm of the operator.

The norm is estimated by calculating

`A(x).norm() / x.norm()`

for some nonzero `x`

Returns`norm` : float

Estimate of operator norm

References

Wikipedia article on [Operator norm](#).

OperatorTest.run_tests`OperatorTest.run_tests()`

Run all tests on this operator.

OperatorTest.self_adjoint`OperatorTest.self_adjoint()`Verify $(Ax, y) = (x, Ay)$ `__init__(operator, operator_norm=None)`

Create a new instance

Parameters`operator` : *Operator*

The operator to run tests on

operator_norm : float

The norm of the operator, used for error estimates can be estimated otherwise.

spaceStandardized tests for *LinearSpace*'s.**Classes**

SpaceTest(space[, eps]) Automated tests for *LinearSpace* instances.

SpaceTest**class** odl.diagnostics.space.**SpaceTest** (space, eps=1e-05)

Bases: object

Automated tests for *LinearSpace* instances.This class allows users to automatically test various features of an *LinearSpace* such as linearity and the various operators.**Methods**

<code>__eq__</code>	Return self==value.
<code>__lincomb()</code>	Check linear combination.
<code>contains()</code>	Verify <i>LinearSpace.__contains__</i> .
<code>dist()</code>	Verify <i>LinearSpace.dist</i> .
<code>element()</code>	Verify <i>LinearSpace.element</i>
<code>equals()</code>	Verify <i>LinearSpace.__eq__</i> .
<code>field()</code>	Verify <i>LinearSpace.field</i>
<code>inner()</code>	Verify <i>LinearSpace.inner</i> .
<code>linearity()</code>	Verify the linear space properties by examples.
<code>multiply()</code>	Verify <i>LinearSpace.multiply</i> .
<code>norm()</code>	Verify <i>LinearSpace.norm</i> .
<code>run_tests()</code>	Run all tests on this space.

Continued on next page

Table 2.5 – continued from previous page

<code>vector()</code>	Verify <code>LinearSpaceVector</code> .
<code>vector_assign()</code>	Verify <code>LinearSpaceVector.assign</code> .
<code>vector_copy()</code>	Verify <code>LinearSpaceVector.copy</code> .
<code>vector_equals()</code>	Verify <code>LinearSpaceVector.__eq__</code> .
<code>vector_set_zero()</code>	Verify <code>LinearSpaceVector.set_zero</code> .
<code>vector_space()</code>	Verify <code>LinearSpaceVector.space</code> .

SpaceTest.lincomb

`SpaceTest.lincomb()`

Check linear combination.

SpaceTest.contains

`SpaceTest.contains()`

Verify `LinearSpace.__contains__`.

SpaceTest.dist

`SpaceTest.dist()`

Verify `LinearSpace.dist`.

The dist satisfies properties

positivity $d(x, y) \geq 0$

coincidence $d(x, y) = 0$ iff $x = y$

symmetry $d(x, y) = d(y, x)$

triangle inequality $d(x, z) = d(x, y) + d(y, z)$

References

Wikipedia article on [metric](#)

SpaceTest.element

`SpaceTest.element()`

Verify `LinearSpace.element`

SpaceTest.equals

`SpaceTest.equals()`

Verify `LinearSpace.__eq__`.

SpaceTest.field

`SpaceTest.field()`

Verify `LinearSpace.field`

SpaceTest.inner`SpaceTest.inner()`Verify `LinearSpace.inner`.

The inner product satisfies properties such as

conjugate symmetry $(x, y) = (y, x)^*$ (* complex conjugate)linearity $(a * x, y) = a * (x, y)$ $(x + y, z) = (x, z) + (y, z)$ positivity $(x, x) \geq 0$ **References**Wikipedia article on [inner product](#).**SpaceTest.linearity**`SpaceTest.linearity()`

Verify the linear space properties by examples.

These properties include things such as associativity

 $x + y = y + x$ and identity of the `LinearSpace.zero` element $x + 0 = x$ **References**Wikipedia article on [Vector space](#).**SpaceTest.multiply**`SpaceTest.multiply()`Verify `LinearSpace.multiply`.

Multiplication satisfies

Zero element $0 * x = 0$ Commutativity $x * y = y * x$ Associativity $x * (y * z) = (x * y) * z$ Distributivity $a * (x + y) = a * x + a * y$ $x * (y + z) = x * y + x * z$ **SpaceTest.norm**`SpaceTest.norm()`Verify `LinearSpace.norm`.

The norm satisfies properties

linearity $||a * x|| = |a| * ||x||$ triangle inequality $||x + y|| \leq ||x|| + ||y||$ separation $||x|| = 0$ iff $x = 0$ positivity $||x|| \geq 0$

References

Wikipedia article on [norm](#).

SpaceTest.run_tests

`SpaceTest.run_tests()`

Run all tests on this space.

SpaceTest.vector

`SpaceTest.vector()`

Verify `LinearSpaceVector`.

SpaceTest.vector_assign

`SpaceTest.vector_assign()`

Verify `LinearSpaceVector.assign`.

SpaceTest.vector_copy

`SpaceTest.vector_copy()`

Verify `LinearSpaceVector.copy`.

SpaceTest.vector_equals

`SpaceTest.vector_equals()`

Verify `LinearSpaceVector.__eq__`.

SpaceTest.vector_set_zero

`SpaceTest.vector_set_zero()`

Verify `LinearSpaceVector.set_zero`.

SpaceTest.vector_space

`SpaceTest.vector_space()`

Verify `LinearSpaceVector.space`.

`__init__(space, eps=1e-05)`

Initialize a new instance.

Parameters`space` : `LinearSpace`

The space that should be tested

eps : float, optional

Precision of the tests.

2.8.2 discr

Discretizations in ODL.

Modules

discr_mappings

Mappings between abstract (continuous) and discrete sets.

Includes grid evaluation (collocation) and various interpolation operators.

Classes

<i>FunctionSetMapping</i> (map_type, fset, ..., linear)	Abstract base class for function set discretization mappings.
<i>LinearInterpolation</i> (fspace, partition, ...)	Linear interpolation interpolation as an <i>Operator</i> .
<i>NearestInterpolation</i> (fset, partition, ...)	Nearest neighbor interpolation as an <i>Operator</i> .
<i>PerAxisInterpolation</i> (fspace, partition, ...)	Interpolation scheme set for each axis individually.
<i>PointCollocation</i> (ip_fset, partition, dspace, ...)	Function evaluation at grid points.

FunctionSetMapping

class odl.discr.discr_mappings.**FunctionSetMapping** (*map_type, fset, partition, dspace, linear=False, **kwargs*)

Bases: *odl.operator.operator.Operator*

Abstract base class for function set discretization mappings.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>grid</i>	The sampling grid.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>order</i>	Axis ordering in the data storage.
<i>partition</i>	The underlying domain partition.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

FunctionSetMapping.adjoint

FunctionSetMapping.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSetMapping.domain

FunctionSetMapping.**domain**

Set of objects on which this operator can be evaluated.

FunctionSetMapping.grid

`FunctionSetMapping.grid`
The sampling grid.

FunctionSetMapping.inverse
`FunctionSetMapping.inverse`
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSetMapping.is_functional
`FunctionSetMapping.is_functional`
True if the this operator's range is a *Field*.

FunctionSetMapping.is_linear
`FunctionSetMapping.is_linear`
True if this operator is linear.

FunctionSetMapping.order
`FunctionSetMapping.order`
Axis ordering in the data storage.

FunctionSetMapping.partition
`FunctionSetMapping.partition`
The underlying domain partition.

FunctionSetMapping.range
`FunctionSetMapping.range`
Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	
<code>_call(x[, out])</code>	Implementation of the operator evaluation.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

FunctionSetMapping.__call__
`FunctionSetMapping.__call__(x, out=None, **kwargs)`
Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns `out` : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

FunctionSetMapping.__eq__

FunctionSetMapping.__eq__(*other*)

FunctionSetMapping._call

FunctionSetMapping._call(*x*, *out=None*, ***kwargs*)

Implementation of the operator evaluation.

This method is private backend for the evaluation of this operator. It needs to match certain signature conventions, and its implementation type is inferred from its signature.

The following signatures are allowed:

Python 2 and 3:

- `_call(self, x)` -> out-of-place evaluation
- `_call(self, vec, out)` -> in-place evaluation
- `_call(self, x, out=None)` -> both

Python 3 only:

- `_call(self, x, *, out=None)` (*out* as keyword-only argument) -> both

For disambiguation, the instance name (the first argument) **must** be ‘self’.

The name of the `out` argument **must** be ‘out’, the second argument may have any name.

Additional variable `**kwargs` and keyword-only arguments (Python 3 only) are also allowed.

Parameters`sx` : *Operator.domain element-like*

Element to which the operator is applied

out : *Operator.range element*, optional

Element to which the result is written

Returns`out` : *Operator.range element-like*

Result of the evaluation. If `out` was provided, the returned object is a reference to it.

Notes

The public call pattern `op()` using `op.__call__` provides a default implementation of the underlying in-place or out-of-place call even if you choose the respective other pattern.

FunctionSetMapping.derivative

`FunctionSetMapping.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*map_type, fset, partition, dspace, linear=False, **kwargs*)

Initialize a new instance.

Parameters`map_type` : {‘restriction’, ‘extension’}

The type of operator

fset : *FunctionSet*

The non-discretized (abstract) set of functions to be discretized

partition : *RectPartition*

Partition of (a subset of) `fset.domain` based on a *TensorGrid*

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its *NtuplesBase.size* must be equal to the total number of grid points.

linear : bool

Create a linear operator if `True`, otherwise a non-linear operator.

order : {‘C’, ‘F’}, optional

Ordering of the axes in the data storage. ‘C’ means the first axis varies slowest, the last axis fastest; vice versa for ‘F’. Default: ‘C’

LinearInterpolation

class odl.dscr.dscr_mappings.**LinearInterpolation** (*fspace*, *partition*, *dspace*, ***kwargs*)

Bases: *odl.dscr.dscr_mappings.FunctionSetMapping*

Linear interpolation interpolation as an *Operator*.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>grid</i>	The sampling grid.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>order</i>	Axis ordering in the data storage.
<i>partition</i>	The underlying domain partition.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

LinearInterpolation.adjoint

`LinearInterpolation.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

LinearInterpolation.domain

`LinearInterpolation.domain`

Set of objects on which this operator can be evaluated.

LinearInterpolation.grid

`LinearInterpolation.grid`

The sampling grid.

LinearInterpolation.inverse

`LinearInterpolation.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

LinearInterpolation.is_functional

`LinearInterpolation.is_functional`

True if the this operator's range is a *Field*.

LinearInterpolation.is_linear

`LinearInterpolation.is_linear`

True if this operator is linear.

LinearInterpolation.order`LinearInterpolation.order`

Axis ordering in the data storage.

LinearInterpolation.partition`LinearInterpolation.partition`

The underlying domain partition.

LinearInterpolation.range`LinearInterpolation.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	
<code>_call(x[, out])</code>	Create an interpolator from grid values <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

LinearInterpolation.__call__`LinearInterpolation.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.**See also:**`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

LinearInterpolation.__eq__

`LinearInterpolation.__eq__(other)`

LinearInterpolation._call

`LinearInterpolation._call(x, out=None)`

Create an interpolator from grid values `x`.

Parameters`x` : *FnBaseVector*

The array of values to be interpolated

out : *FunctionSpaceVector*, optional

Vector in which to store the interpolator

Returns`out` : *FunctionSpaceVector*

Linear interpolator for the grid of this operator. If `out` was provided, the returned object is a reference to it.

LinearInterpolation.derivative

`LinearInterpolation.derivative(point)`

Return the operator derivative at `point`.

Raises`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

__init__(*fspace, partition, dspace, **kwargs*)

Initialize a new instance.

Parameters`fspace` : *FunctionSpace*

The undiscretized (abstract) space of functions to be discretized. Its field must be the same as that of data space. The function domain must provide a *Set.contains_set* method.

partition : *RectPartition*

Partition of (a subset of) `fspace.domain` based on a *TensorGrid*

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its *NtuplesBase.size* must be equal to the total number of grid points, and its *FnBase.field* must be the same as that of the function space.

order : {'C', 'F'}, optional

Ordering of the axes in the data storage. ‘C’ means the first axis varies slowest, the last axis fastest; vice versa for ‘F’. Default: ‘C’

NearestInterpolation

class odl.dscr.dscr_mappings.**NearestInterpolation** (*fset, partition, dspace, **kwargs*)

Bases: *odl.dscr.dscr_mappings.FunctionSetMapping*

Nearest neighbor interpolation as an *Operator*.

Given points $x_1 < x_2 < \dots < x_N$, and values f_1, \dots, f_N , nearest neighbor interpolation at x is defined by:

$$I(x) = f_j \text{ with } j \text{ such that } |x - x_j| \text{ is minimal.}$$

The ambiguity at the midpoints is resolved by preferring one of the neighbors. For higher dimensions, this rule is applied per component.

The nearest neighbor interpolation operator is defined as the mapping from the values f_1, \dots, f_N to the function $I(x)$ (as a whole).

In higher dimensions, this principle is applied per axis, the only difference being the additional information about the ordering of the axes in the flat storage array (C- vs. Fortran ordering).

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>grid</i>	The sampling grid.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>order</i>	Axis ordering in the data storage.
<i>partition</i>	The underlying domain partition.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

NearestInterpolation.adjoint

`NearestInterpolation.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

NearestInterpolation.domain

`NearestInterpolation.domain`

Set of objects on which this operator can be evaluated.

NearestInterpolation.grid

`NearestInterpolation.grid`

The sampling grid.

NearestInterpolation.inverse`NearestInterpolation.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

NearestInterpolation.is_functional`NearestInterpolation.is_functional`True if the this operator's range is a *Field*.**NearestInterpolation.is_linear**`NearestInterpolation.is_linear`

True if this operator is linear.

NearestInterpolation.order`NearestInterpolation.order`

Axis ordering in the data storage.

NearestInterpolation.partition`NearestInterpolation.partition`

The underlying domain partition.

NearestInterpolation.range`NearestInterpolation.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	
<code>_call(x[, out])</code>	Create an interpolator from grid values <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

NearestInterpolation.__call__`NearestInterpolation.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

NearestInterpolation.__eq__

NearestInterpolation.__eq__(*other*)

NearestInterpolation._call

NearestInterpolation._call(*x*, *out=None*)

Create an interpolator from grid values *x*.

Parameters`x` : `NtuplesVector`

The array of values to be interpolated

out : `FunctionSetVector`, optional

Vector in which to store the interpolator

Returns`out` : `FunctionSetVector`

Nearest-neighbor interpolator for the grid of this operator. If `out` was provided, the returned object is a reference to it.

See also:

`LinearInterpolation`(bi-/tri-/...)linear interpolation

Notes

Important: if called on a point array, the points are assumed to be sorted in ascending order in each dimension for efficiency reasons.

Nearest neighbor interpolation is the only scheme which works with data of non-scalar type since it does not involve any arithmetic operations on the values.

Examples

We test nearest neighbor interpolation with a non-scalar data type in 2d:

```
>>> import numpy as np
>>> from odl import Rectangle, Strings, FunctionSet
>>> rect = Rectangle([0, 0], [1, 1])
>>> strings = Strings(1) # 1-char strings
>>> space = FunctionSet(rect, strings)
```

Partitioning the domain uniformly with no nodes on the boundary (will shift the grid points):

```
>>> from odl import uniform_partition_fromintv, Ntuples
>>> part = uniform_partition_fromintv(rect, [4, 2],
...                                   nodes_on_bdry=False)
>>> part.grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]), array([ 0.25,  0.75]))
```

```
>>> dspace = Ntuples(part.size, dtype='U1')
```

Now we initialize the operator and test it with some points:

```
>>> interp_op = NearestInterpolation(space, part, dspace)
>>> values = np.array([c for c in 'mystring'])
>>> function = interp_op(values)
>>> print(function([0.3, 0.6])) # closest to index (1, 1) -> 3
t
>>> out = np.empty(2, dtype='U1')
>>> pts = np.array([[0.3, 0.6],
...                [1.0, 1.0]])
>>> out = function(pts.T, out=out) # returns original out
>>> all(out == ['t', 'g'])
True
```

NearestInterpolation.derivative

`NearestInterpolation.derivative` (*point*)

Return the operator derivative at point.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*fset*, *partition*, *dspace*, ***kwargs*)

Initialize a new instance.

Parameters*fset* : *FunctionSet*

The undiscretized (abstract) set of functions to be discretized. The function domain must provide a *Set.contains_set* method.

partition : *RectPartition*

Partition of (a subset of) `ip_fset.domain` based on a spatial grid

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its *NtuplesBase.size* must be equal to the total number of grid points.

variant : { 'left', 'right' }, optional

Behavior variant at midpoint between neighbors

'left' : favor left neighbor (default)

'right' : favor right neighbor

order : { 'C', 'F' }, optional

Ordering of the axes in the data storage. 'C' means the first axis varies slowest, the last axis fastest; vice versa for 'F'. Default: 'C'

Notes

The distinction between 'left' and 'right' variants is currently made by changing `<=` to `<` at one place. This difference may not be noticable in some situations due to rounding errors.

PerAxisInterpolation

class `odl.discr.discr_mappings.PerAxisInterpolation` (*fspace*, *partition*, *dspace*, *schemes*, ***kwargs*)

Bases: `odl.discr.discr_mappings.FunctionSetMapping`

Interpolation scheme set for each axis individually.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>grid</i>	The sampling grid.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>nn_variants</i>	List of nearest neighbor variants, one for each axis.
<i>order</i>	Axis ordering in the data storage.
<i>partition</i>	The underlying domain partition.
<i>range</i>	Set in which the result of an evaluation of this operator lies.
<i>schemes</i>	List of interpolation schemes, one for each axis.

PerAxisInterpolation.adjoint

`PerAxisInterpolation.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

PerAxisInterpolation.domain

`PerAxisInterpolation.domain`

Set of objects on which this operator can be evaluated.

PerAxisInterpolation.grid

`PerAxisInterpolation.grid`

The sampling grid.

PerAxisInterpolation.inverse

`PerAxisInterpolation.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

PerAxisInterpolation.is_functional

`PerAxisInterpolation.is_functional`

True if the this operator's range is a *Field*.

PerAxisInterpolation.is_linear

`PerAxisInterpolation.is_linear`

True if this operator is linear.

PerAxisInterpolation.nn_variants

`PerAxisInterpolation.nn_variants`

List of nearest neighbor variants, one for each axis.

PerAxisInterpolation.order

`PerAxisInterpolation.order`

Axis ordering in the data storage.

PerAxisInterpolation.partition

`PerAxisInterpolation.partition`

The underlying domain partition.

PerAxisInterpolation.range

`PerAxisInterpolation.range`

Set in which the result of an evaluation of this operator lies.

PerAxisInterpolation.schemes

`PerAxisInterpolation.schemes`

List of interpolation schemes, one for each axis.

Continued on next page

Table 2.14 – continued from previous page

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	
<code>_call(x[, out])</code>	Create an interpolator from grid values <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

PerAxisInterpolation.__call__

`PerAxisInterpolation.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

PerAxisInterpolation.__eq__

`PerAxisInterpolation.__eq__ (other)`

PerAxisInterpolation._call

`PerAxisInterpolation._call (x, out=None)`

Create an interpolator from grid values `x`.

Parameters`x` : *FnBaseVector*

The array of values to be interpolated

out : *FunctionSpaceVector*, optional

Vector in which to store the interpolator

Returns`out` : *FunctionSpaceVector*

Per-axis interpolator for the grid of this operator. If `out` was provided, the returned object is a reference to it.

PerAxisInterpolation.derivative

`PerAxisInterpolation.derivative (point)`

Return the operator derivative at `point`.

Raises`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*fspace*, *partition*, *dspace*, *schemes*, ***kwargs*)

Initialize a new instance.

Parameters`fspace` : *FunctionSpace*

The undiscretized (abstract) space of functions to be discretized. Its field must be the same as that of data space. The function domain must provide a *Set.contains_set* method.

partition : *RectPartition*

Partition of (a subset of) `fspace.domain` based on a *TensorGrid*

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its *NtuplesBase.size* must be equal to the total number of grid points, and its *FnBase.field* must be the same as that of the function space.

schemes : `str` or sequence of `str`

Indicates which interpolation scheme to use for which axis. A single string is interpreted as a global scheme for all axes.

nn_variants : `str` or sequence of `str`, optional

Which variant ('left' or 'right') to use in nearest neighbor interpolation for which axis. A single string is interpreted as a global variant for all axes. This option has no effect for schemes other than nearest neighbor. Default: 'left'

order : {'C', 'F'}, optional

Ordering of the axes in the data storage. 'C' means the first axis varies slowest, the last axis fastest; vice versa for 'F'. Default: 'C'

PointCollocation

class odl.dscr.dscr_mappings.**PointCollocation**(*ip_fset, partition, dspace, **kwargs*)

Bases: odl.dscr.dscr_mappings.*FunctionSetMapping*

Function evaluation at grid points.

This operator evaluates a given function in a set of points. These points are given as the sampling grid of a partition of the function domain. The result of this evaluation is an array of function values at these points.

If, for example, a function is defined on the interval $[0, 1]$, and a partition divides the interval into N subintervals, the resulting array will have length N . The sampling points are defined by the partition, usually they are the midpoints of the subintervals.

In higher dimensions, the same principle is applied, with the only difference being the additional information about the ordering of the axes in the flat storage array (C- vs. Fortran ordering).

This operator is the default ‘restriction’ used by all core discretization classes.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>grid</i>	The sampling grid.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>order</i>	Axis ordering in the data storage.
<i>partition</i>	The underlying domain partition.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

PointCollocation.adjoint

PointCollocation.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

PointCollocation.domain

PointCollocation.**domain**

Set of objects on which this operator can be evaluated.

PointCollocation.grid

PointCollocation.**grid**

The sampling grid.

PointCollocation.inverse

PointCollocation.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

PointCollocation.is_functional**PointCollocation.is_functional**True if the this operator's range is a *Field*.**PointCollocation.is_linear****PointCollocation.is_linear**

True if this operator is linear.

PointCollocation.order**PointCollocation.order**

Axis ordering in the data storage.

PointCollocation.partition**PointCollocation.partition**

The underlying domain partition.

PointCollocation.range**PointCollocation.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	
<code>_call(func[, out])</code>	Evaluate <code>func</code> at the grid of this operator.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

PointCollocation.__call__**PointCollocation.__call__** (*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns***out* : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call__` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

PointCollocation.__eq__

PointCollocation.__eq__(*other*)

PointCollocation._call

PointCollocation.__call__(*func*, *out=None*)

Evaluate *func* at the grid of this operator.

Parameters*func* : *FunctionSetVector*

The function to be evaluated

out : *NtuplesBaseVector*, optional

Array to which the values are written. Its shape must be $(N,)$, where N is the total number of grid points. The data type must be the same as in the *dspace* of this mapping.

Returns*out* : *NtuplesBaseVector*, optional

The function values at the grid points. If *out* was provided, the returned object is a reference to it.

See also:

`odl.discr.grid.TensorGrid.meshgrid`, `numpy.meshgrid`

Notes

This operator expects its input functions to be written in a vectorization-conforming manner to ensure fast evaluation. See the [vectorization guide](#) for a detailed introduction.

Examples

Define a set of functions from the rectangle $[1, 3] \times [2, 5]$ to the real numbers:

```
>>> from odl import FunctionSpace, Rectangle
>>> rect = Rectangle([1, 3], [2, 5])
>>> funcset = FunctionSpace(rect)
```

Partition the rectangle by a tensor grid:

```
>>> from odl import TensorGrid, Rectangle, RectPartition, Rn
>>> rect = Rectangle([1, 3], [2, 5])
>>> grid = TensorGrid([1, 2], [3, 4, 5])
>>> partition = RectPartition(rect, grid)
>>> rn = Rn(grid.size)
```

Finally create the operator and test it on a function:

```
>>> coll_op = PointCollocation(funcset, partition, rn)
...
... # Properly vectorized function
>>> func_elem = funcset.element(lambda x: x[0] - x[1])
>>> coll_op(func_elem)
Rn(6).element([-2.0, -3.0, -4.0, -1.0, -2.0, -3.0])
>>> coll_op(lambda x: x[0] - x[1]) # Works directly
Rn(6).element([-2.0, -3.0, -4.0, -1.0, -2.0, -3.0])
>>> out = Rn(6).element()
>>> coll_op(func_elem, out=out) # In-place
Rn(6).element([-2.0, -3.0, -4.0, -1.0, -2.0, -3.0])
```

Fortran ordering:

```
>>> coll_op = PointCollocation(funcset, partition, rn, order='F')
>>> coll_op(func_elem)
Rn(6).element([-2.0, -1.0, -3.0, -2.0, -4.0, -3.0])
```

PointCollocation.derivative

`PointCollocation.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*ip_fset*, *partition*, *dspace*, ***kwargs*)

Initialize a new instance.

Parameters*fset* : *FunctionSet*

The non-discretized (abstract) set of functions to be discretized. The function domain must provide a *Set.contains_set* method.

partition : *RectPartition*

Partition of (a subset of) *ip_fset.domain* based on a *TensorGrid*

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its *NtuplesBase.size* must be equal to the total number of grid points.

order : {'C', 'F'}, optional

Ordering of the axes in the data storage. ‘C’ means the first axis varies slowest, the last axis fastest; vice versa for ‘F’. Default: ‘C’

discr_ops

Operators defined on *DiscreteLp*.

Classes

<i>Divergence</i> (space[, method])	Divergence operator for <i>DiscreteLp</i> spaces.
<i>Gradient</i> (space[, method])	Spatial gradient operator for <i>DiscreteLp</i> spaces.
<i>Laplacian</i> (space)	Spatial Laplacian operator for <i>DiscreteLp</i> spaces.
<i>PartialDerivative</i> (space[, axis, method, ...])	Calculate the discrete partial derivative along a given axis.

Divergence

class odl.discr.discr_ops.**Divergence**(space, method='forward')

Bases: *odl.operator.operator.Operator*

Divergence operator for *DiscreteLp* spaces.

Calls helper function *finite_diff* for each component of the input product space vector. For the adjoint of the *Divergence* operator to match the negative *Gradient* operator implicit zero is assumed.

Attributes

<i>adjoint</i>	Return the adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

Divergence.adjoint

Divergence.**adjoint**

Return the adjoint operator.

Assuming implicit zero padding, the adjoint operator is given by the negative of the *Gradient* operator.

Divergence.domain

Divergence.**domain**

Set of objects on which this operator can be evaluated.

Divergence.inverse

Divergence.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

Divergence.is_functional

Divergence.**is_functional**

True if the this operator's range is a *Field*.

Divergence.is_linear

Divergence.**is_linear**

True if this operator is linear.

Divergence.range

Divergence.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Calculate the divergence of <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

Divergence.__call__

Divergence.**__call__**(`x`, `out=None`, `**kwargs`)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```


Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

Divergence._call

Divergence._call(*x*, *out=None*)

Calculate the divergence of *x*.

Parameters*rx*: domain *element*

ProductSpaceVector to which the divergence operator is applied

out: range *element*, optional

Output vector to which the result is written

Returns*out*: range *element*

Result of the evaluation. If *out* is provided, the returned object is a reference to it.

Examples

```
>>> from odl import uniform_discr
>>> data = np.array([[0., 1., 2., 3., 4.],
...                 [1., 2., 3., 4., 5.],
...                 [2., 3., 4., 5., 6.]])
>>> space = uniform_discr([0, 0], [3, 5], data.shape)
>>> div = Divergence(space)
>>> f = div.domain.element([data, data])
>>> div_f = div(f)
>>> print(div_f)
[[2.0, 2.0, 2.0, 2.0, -3.0],
 [2.0, 2.0, 2.0, 2.0, -4.0],
 [-1.0, -2.0, -3.0, -4.0, -12.0]]
```

Verify adjoint:

```
>>> g = div.range.element(data ** 2)
>>> adj_div_g = div.adjoint(g)
>>> g.inner(div_f) / f.inner(adj_div_g)
1.0
```

Divergence.derivative

Divergence.derivative(*point*)

Return the operator derivative at *point*.

Raises*OpNotImplementedError*

If the operator is not linear, the derivative cannot be default implemented.

`__init__(space, method='forward')`

Initialize a *Divergence* operator instance.

Zero padding is assumed for the adjoint of the *Divergence* operator to match negative *Gradient* operator.

Parameters`space` : *DiscreteLp*

The space of elements which the operator is acting on

method : {'central', 'forward', 'backward'}, optional

Finite difference method to be used

Gradient

class `odl.discr.discr_ops.Gradient(space, method='forward')`

Bases: *odl.operator.operator.Operator*

Spatial gradient operator for *DiscreteLp* spaces.

Calls helper function *finite_diff* to calculate each component of the resulting product space vector. For the adjoint of the *Gradient* operator, zero padding is assumed to match the negative *Divergence* operator

Attributes

<i>adjoint</i>	Return the adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

Gradient.adjoint

`Gradient.adjoint`

Return the adjoint operator.

Assuming implicit zero padding, the adjoint operator is given by the negative of the *Divergence* operator.

The Divergence is constructed from a space as a product space operator $\text{space}^n \rightarrow \text{space}$, hence we need to provide the domain of this operator.

Gradient.domain

`Gradient.domain`

Set of objects on which this operator can be evaluated.

Gradient.inverse

`Gradient.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

Gradient.is_functional`Gradient.is_functional`True if the this operator's range is a *Field*.**Gradient.is_linear**`Gradient.is_linear`

True if this operator is linear.

Gradient.range`Gradient.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Calculate the spatial gradient of <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

Gradient.__call__`Gradient.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.**See also:**`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

Gradient.call

`Gradient._call(x, out=None)`

Calculate the spatial gradient of `x`.

Parameters`rx`: domain *element*

Input vector to which the *Gradient* operator is applied

out: range *element*, optional

Output vector to which the result is written

Returns`out`: range *element*

Result of the evaluation. If `out` is provided, the returned object is a reference to it.

Examples

```
>>> from odl import uniform_discr
>>> data = np.array([[ 0., 1., 2., 3., 4.],
...                  [ 0., 2., 4., 6., 8.]])
>>> discr = uniform_discr([0, 0], [2, 5], data.shape)
>>> f = discr.element(data)
>>> grad = Gradient(discr)
>>> grad_f = grad(f)
>>> print(grad_f[0])
[[0.0, 1.0, 2.0, 3.0, 4.0],
 [0.0, -2.0, -4.0, -6.0, -8.0]]
>>> print(grad_f[1])
[[1.0, 1.0, 1.0, 1.0, -4.0],
 [2.0, 2.0, 2.0, 2.0, -8.0]]
```

Verify adjoint:

```
>>> g = grad.range.element((data, data ** 2))
>>> adj_g = grad.adjoint(g)
>>> print(adj_g)
[[0.0, -2.0, -5.0, -8.0, -11.0],
 [0.0, -5.0, -14.0, -23.0, -32.0]]
>>> g.inner(grad_f) / f.inner(adj_g)
1.0
```

Gradient.derivative

`Gradient.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*space*, *method*='forward')

Initialize a *Gradient* operator instance.

Zero padding is assumed for the adjoint of the *Gradient* operator to match negative *Divergence* operator.

Parameters*space* : *DiscreteLp*

The space of elements which the operator is acting on.

method : {'central', 'forward', 'backward'}, optional

Finite difference method to be used

Laplacian

class odl.discr.discr_ops.**Laplacian** (*space*)

Bases: *odl.operator.operator.Operator*

Spatial Laplacian operator for *DiscreteLp* spaces.

Calls helper function *finite_diff* to calculate each component of the resulting product space vector.

Outside the domain zero padding is assumed.

Attributes

<i>adjoint</i>	Return the adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

Laplacian.adjoint

Laplacian.adjoint

Return the adjoint operator.

The laplacian is self-adjoint, so this returns `self`.

Laplacian.domain

Laplacian.domain

Set of objects on which this operator can be evaluated.

Laplacian.inverse

Laplacian.inverse

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

Laplacian.is_functional

Laplacian.**is_functional**

True if the this operator's range is a *Field*.

Laplacian.is_linear

Laplacian.**is_linear**

True if this operator is linear.

Laplacian.range

Laplacian.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Calculate the spatial Laplacian of <code>x</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

Laplacian.__call__

Laplacian.**__call__** (`x`, `out=None`, `**kwargs`)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

Laplacian._call

`Laplacian._call(x, out=None)`

Calculate the spatial Laplacian of `x`.

Parameters`x`: domain *element*

Input vector to which the *Laplacian* operator is applied

out: range *element*, optional

Output vector to which the result is written

Returns`out`: range *element*

Result of the evaluation. If `out` is provided, the returned object is a reference to it.

Examples

```
>>> from odl import uniform_discr
>>> data = np.array([[ 0., 0., 0.],
...                  [ 0., 1., 0.],
...                  [ 0., 0., 0.]])
>>> space = uniform_discr([0, 0], [3, 3], data.shape)
>>> f = space.element(data)
>>> lap = Laplacian(space)
>>> print(lap(f))
[[0.0, 1.0, 0.0],
 [1.0, -4.0, 1.0],
 [0.0, 1.0, 0.0]]
```

Laplacian.derivative

`Laplacian.derivative(point)`

Return the operator derivative at `point`.

Raises`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

__init__(*space*)

Initialize a *Laplacian* operator instance.

Parameters`space`: *DiscreteLp*

The space of elements which the operator is acting on

PartialDerivative

```
class odl.discr.discr_ops.PartialDerivative(space, axis=0, method='forward',
                                             padding_method=None, padding_value=0,
                                             edge_order=None)
```

Bases: `odl.operator.operator.Operator`

Calculate the discrete partial derivative along a given axis.

Calls helper function `finite_diff` to calculate finite difference. Preserves the shape of the underlying grid.

Attributes

<code>adjoint</code>	Return the adjoint operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

PartialDerivative.adjoint

`PartialDerivative.adjoint`

Return the adjoint operator.

PartialDerivative.domain

`PartialDerivative.domain`

Set of objects on which this operator can be evaluated.

PartialDerivative.inverse

`PartialDerivative.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

PartialDerivative.is_functional

`PartialDerivative.is_functional`

True if the this operator's range is a *Field*.

PartialDerivative.is_linear

`PartialDerivative.is_linear`

True if this operator is linear.

PartialDerivative.range

`PartialDerivative.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Apply gradient operator to <code>x</code> and store result in <code>out</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

PartialDerivative.__call__

`PartialDerivative.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

PartialDerivative._call`PartialDerivative._call(x, out=None)`Apply gradient operator to `x` and store result in `out`.**Parameters**`x`: domain *element*

Input vector to which the operator is applied to

out: range element, optional

Output vector to which the result is written

Returns`out`: range *element*Result of the evaluation. If `out` is provided, the returned object is a reference to it.**Examples**

```

>>> from odl import uniform_discr
>>> data = np.array([[ 0.,  1.,  2.,  3.,  4.],
...                  [ 0.,  2.,  4.,  6.,  8.]])
>>> discr = uniform_discr([0, 0], [2, 1], data.shape)
>>> par_deriv = PartialDerivative(discr)
>>> f = par_deriv.domain.element(data)
>>> par_div_f = par_deriv(f)
>>> print(par_div_f)
[[0.0, 1.0, 2.0, 3.0, 4.0],
 [0.0, 1.0, 2.0, 3.0, 4.0]]

```

PartialDerivative.derivative`PartialDerivative.derivative(point)`Return the operator derivative at `point`.**RaisesOpNotImplementedError**

If the operator is not linear, the derivative cannot be default implemented.

`__init__(space, axis=0, method='forward', padding_method=None, padding_value=0, edge_order=None)`

Initialize an operator instance.

Parameters`space`: *DiscreteLp*

The space of elements which the operator is acting on

axis: int, optional

The axis along which the partial derivative is evaluated

method: {'central', 'forward', 'backward'}, optionalFinite difference method which is used in the interior of the domain of `f`**padding_method**: {'constant', 'symmetric'}, optional'constant': Pads values outside the domain of `f` with a constant value given by `padding_value`

'symmetric': Pads with the reflection of the vector mirrored along the edge of the array

If `None` is given, one-sided forward or backward differences are used at the boundary**padding_value**: float, optional

If `padding_method` is 'constant' `f` assumes `padding_value` for indices outside the domain of `f`

edge_order : {1, 2}, optional

Edge-order accuracy at the boundaries if no padding is used. If `None` the edge-order accuracy at endpoints corresponds to the accuracy in the interior.

Functions

`finite_diff(f[, axis, dx, method, out])` Calculate the partial derivative of `f` along a given `axis`.

`finite_diff`

`odl.discr.discr_ops.finite_diff(f, axis=0, dx=1.0, method='forward', out=None, **kwargs)`

Calculate the partial derivative of `f` along a given `axis`.

In the interior of the domain of `f`, the partial derivative is computed using first-order accurate forward or backward difference or second-order accurate central differences.

With padding the same method and thus accuracy is used on endpoints as in the interior i.e. forward and backward differences use first-order accuracy on edges while central differences use second-order accuracy at edges.

Without padding one-sided forward or backward differences are used at the boundaries. The accuracy at the endpoints can then also be triggered by the edge order.

The returned array has the same shape as the input array `f`.

Per default forward difference with `dx=1` and no padding is used.

Parameters `f` : *array-like*

An N-dimensional array

axis : int, optional

The axis along which the partial derivative is evaluated

dx : float, optional

Scalar specifying the distance between sampling points along `axis`

method : {'central', 'forward', 'backward'}, optional

Finite difference method which is used in the interior of the domain of `f`.

padding_method : {'constant', 'symmetric'}, optional

'constant' : Pads values outside the domain of `f` with a constant value given by `padding_value`

'symmetric' : Pads with the reflection of the vector mirrored along the edge of the array

If `None` is given, one-sided forward or backward differences are used at the boundary.

padding_value : float, optional

If `padding_method` is 'constant' `f` assumes `padding_value` for indices outside the domain of `f`

edge_order : {1, 2}, optional

Edge-order accuracy at the boundaries if no padding is used. If `None` the edge-order accuracy at endpoints corresponds to the accuracy in the interior. Default: `None`

out : `numpy.ndarray`, optional

An N-dimensional array to which the output is written. Has to have the same shape as the input array `f`. Default: `None`

Returns**out** : `numpy.ndarray`

N-dimensional array of the same shape as `f`. If `out` is provided, the returned object is a reference to it.

Notes

Without padding the use of second-order accurate edges requires at least three elements.

Central differences with padding cannot be used with first-order accurate edges.

Forward and backward differences with padding use the first-order accuracy on edges (as in the interior).

An edge-order accuracy different from the interior can only be triggered without padding i.e. when one-sided differences are used at the edges.

Examples

```
>>> f = np.array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
>>> finite_diff(f)
array([ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Without arguments the above defaults to:

```
>>> finite_diff(f, axis=0, dx=1.0, method='forward', padding_method=None,
... edge_order=None)
array([ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
>>> finite_diff(f, dx=0.5)
array([ 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
>>> finite_diff(f, padding_method='constant')
array([ 1., 1., 1., 1., 1., 1., 1., 1., 1., -9.])
```

Central differences and different edge orders:

```
>>> finite_diff(1/2*f**2, method='central')
array([-0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> finite_diff(1/2*f**2, method='central', edge_order=1)
array([ 0.5, 1., 2., 3., 4., 5., 6., 7., 8., 8.5])
```

In-place evaluation:

```
>>> out = f.copy()
>>> out is finite_diff(f, out=out)
True
```

discretization

Base classes for discretization.

Classes

<code>Discretization(ospace, dspace[, restr, ext])</code>	Abstract class for discretizations of linear vector spaces.
<code>DiscretizationVector(space, data)</code>	Representation of a <code>Discretization</code> element.
<code>RawDiscretization(ospace, dspace[, restr, ext])</code>	Abstract raw discretization class.
<code>RawDiscretizationVector(space, ntuple)</code>	Representation of a <code>RawDiscretization</code> element.

Discretization

class `odl.discr.discretization.Discretization` (*ospace, dspace, restr=None, ext=None*)

Bases: `odl.discr.discretization.RawDiscretization`, `odl.space.base_ntuples.FnBase`

Abstract class for discretizations of linear vector spaces.

This variant of `RawDiscretization` adds linear structure to all its members. The `RawDiscretization.ospace` is a `LinearSpace`, the `RawDiscretization.dspace` for the data representation is an implementation of \mathbb{F}^n , where \mathbb{F} is some `Field`, and both `RawDiscretization.restriction` and `RawDiscretization.extension` are linear `Operator`'s.

Attributes

<code>domain</code>	The domain of the continuous space.
<code>dspace</code>	The data space.
<code>dspace_type</code>	Data space type of this discretization.
<code>dtype</code>	The data type of each entry.
<code>element_type</code>	<code>DiscretizationVector</code>
<code>extension</code>	The operator mapping an n-tuple to a <code>ospace</code> element.
<code>field</code>	The field of this vector space.
<code>is_cn</code>	Return <code>True</code> if the space represents \mathbb{C}^n , i.e.
<code>is_rn</code>	Return <code>True</code> if the space represents \mathbb{R}^n , i.e.
<code>is_weighted</code>	Return <code>True</code> if the <code>dspace</code> is weighted.
<code>restriction</code>	The operator mapping a <code>ospace</code> element to an n-tuple.
<code>shape</code>	The shape of this space.
<code>size</code>	The number of entries per tuple.
<code>ospace</code>	The undiscretized space.
<code>weighting</code>	This space's weighting scheme.

Discretization.domain

`Discretization.domain`

The domain of the continuous space.

Discretization.dspace

`Discretization.dspace`

The data space.

Discretization.dspace_type

`Discretization.dspace_type`

Data space type of this discretization.

Discretization.dtype`Discretization.dtype`

The data type of each entry.

Discretization.element_type`Discretization.element_type`

DiscretizationVector

Discretization.extension`Discretization.extension`

The operator mapping an n-tuple to a *uspace* element.

Discretization.field`Discretization.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns`field` : *Field*

The underlying field.

Discretization.is_cn`Discretization.is_cn`

Return True if the space represents \mathbb{C}^n , i.e. complex tuples.

Discretization.is_rn`Discretization.is_rn`

Return True if the space represents \mathbb{R}^n , i.e. real tuples.

Discretization.is_weighted`Discretization.is_weighted`

Return True if the dspace is weighted.

Discretization.restriction`Discretization.restriction`

The operator mapping a *uspace* element to an n-tuple.

Discretization.shape`Discretization.shape`

The shape of this space.

Discretization.size`Discretization.size`

The number of entries per tuple.

Discretization.uspace`Discretization.uspace`

The undiscretized space.

Discretization.weighting`Discretization.weighting`

This space's weighting scheme.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>_dist(x1, x2)</code>	Raw distance between two vectors.
<code>_divide(x1, x2, out)</code>	Raw pointwise multiplication of two vectors.
<code>_inner(x1, x2)</code>	Raw inner product of two vectors.
<code>_lincomb(a, x1, b, x2, out)</code>	Raw linear combination.
<code>_multiply(x1, x2, out)</code>	Raw pointwise multiplication of two vectors.
<code>_norm(x)</code>	Raw norm of a vector.
<code>astype(dtype)</code>	Return a copy of this space with new dtype.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp])</code>	Create an element from inp or from scratch.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create a vector of ones.
<code>zero()</code>	Create a vector of zeros.

Discretization.__contains__`Discretization.__contains__(other)`

Return other in self.

Returns`contains` : boolTrue if other is an `NtuplesBaseVector` instance and `other.space` is equal to this space, False otherwise.**Examples**

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```


Discretization.__eq__

Discretization.__eq__(other)

Return self == other.

Returnsequals : bool

True if other is a *RawDiscretization* instance and all attributes *uspace*, *dspace*, *RawDiscretization.restriction* and *RawDiscretization.extension* of other and this discretization are equal, False otherwise.

Discretization._dist

Discretization._dist(x1, x2)

Raw distance between two vectors.

Discretization._divide

Discretization._divide(x1, x2, out)

Raw pointwise multiplication of two vectors.

Discretization._inner

Discretization._inner(x1, x2)

Raw inner product of two vectors.

Discretization._lincomb

Discretization._lincomb(a, x1, b, x2, out)

Raw linear combination.

Discretization._multiply

Discretization._multiply(x1, x2, out)

Raw pointwise multiplication of two vectors.

Discretization._norm

Discretization._norm(x)

Raw norm of a vector.

Discretization.astype

Discretization.astype(dtype)

Return a copy of this space with new dtype.

Parametersdtype :

Data type of the returned space. Can be given in any way *numpy.dtype* understands, e.g. as string ('complex64') or data type (*complex*).

Returnsnewspace : *FnBase*

The version of this space with given data type

Discretization.contains_all

`Discretization.contains_all` (*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

Discretization.contains_set

`Discretization.contains_set` (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Discretization.dist

`Discretization.dist` (*x1*, *x2*)

Calculate the distance between two vectors.

Parameters*x1*, *x2* : *LinearSpaceVector*

Vectors whose distance to compute

Returns*dist* : float

Distance between vectors

Discretization.divide

`Discretization.divide` (*x1*, *x2*, *out=None*)

Calculate the pointwise division of *x1* and *x2*

Parameters*x1* : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns*out* : *LinearSpaceVector*

Ratio of the vectors. If *out* was provided, the returned object is a reference to it.

Discretization.element

`Discretization.element` (*inp=None*)

Create an element from *inp* or from scratch.

Parameters*inp* : object, optional

The input data to create an element from. Must be recognizable by the *LinearSpace.element* method of either *dspace* or *uspace*.

Return*element* : *RawDiscretizationVector*

The discretized element, calculated as `dspace.element(inp)` or `restriction(uspace.element(inp))`, tried in this order.

Discretization.inner

`Discretization.inner(x1, x2)`

Calculate the inner product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Factors in the inner product

Returns`out` : *LinearSpace.field* element

Product of the vectors. If `out` was provided, the returned object is a reference to it.

Discretization.lincomb

`Discretization.lincomb(a, x1, b=None, x2=None, out=None)`

Linear combination of vectors.

Calculates

`out = a * x1`

or, if `b` and `y` are given,

`out = a*x1 + b*x2`

with error checking of types.

Parameters`a` : Scalar in the field of this space

Scalar to multiply `x1` with.

`x1` : *LinearSpaceVector*

The first of the summands

`b` : Scalar, optional

Scalar to multiply `x2` with.

`x2` : *LinearSpaceVector*, optional

The second of the summands

`out` : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns`out` : *LinearSpaceVector*

Result of the linear combination. If `out` was provided, the returned object is a reference to it.

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

`space.lincomb(x, 2, x, 3.14, out=x)`

is (mathematically) equivalent to

`x = x * (1 + 2 + 3.14)`

Discretization.multiply`Discretization.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

Discretization.norm`Discretization.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

Discretization.one`Discretization.one()`

Create a vector of ones.

Discretization.zero`Discretization.zero()`

Create a vector of zeros.

__init__(*uspace, dspace, restr=None, ext=None*)

Abstract initialization method.

Intended to be called by subclasses for proper type checking and setting of attributes.

Parameters`uspace` : *LinearSpace*

The (abstract) space to be discretized

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its *FnBase.field* attribute must be the same as `uspace.field`.

restr : *Operator*, linear, optional

Operator mapping a *RawDiscretization.uspace* element to a *RawDiscretization.dspace* element. Must satisfy `restr.domain == uspace`, `restr.range == dspace`

ext : *Operator*, linear, optional

Operator mapping a *RawDiscretization.dspace* element to a *RawDiscretization.uspace* element. Must satisfy `ext.domain == dspace`, `ext.range == uspace`.

DiscretizationVector

class odl.discr.discretization.**DiscretizationVector** (*space, data*)

Bases: *odl.discr.discretization.RawDiscretizationVector*,
odl.space.base_ntuples.FnBaseVector

Representation of a *Discretization* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (.
<i>dtype</i>	type of data storage.
<i>extension</i>	The extension operator associated with this vector.
<i>itemsize</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
<i>ntuple</i>	Structure for data storage.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	size of data storage.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>NtuplesBaseUFuncs</i> , access to numpy style ufuncs.

DiscretizationVector.T

DiscretizationVector.**T**

The transpose of a vector, the functional given by (., self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

DiscretizationVector.dtype

DiscretizationVector.**dtype**

type of data storage.

DiscretizationVector.extension`DiscretizationVector.extension`

The extension operator associated with this vector.

Return`extension_op` : *FunctionSetMapping*

Operator representing a continuous extension of this vector.

See also:

RawDiscretization.extension For full description

Examples

```
>>> import odl
>>> import numpy as np
```

Create continuous extension of 1d function with nearest neighbour

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True)
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.,  1.]
```

Create continuous extension of 1d function wiht linear interpolation

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True, interp='linear')
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.48,  0.52])
```

DiscretizationVector.itemsize`DiscretizationVector.itemsize`

The size in bytes on one element of this type.

DiscretizationVector.nbytes`DiscretizationVector.nbytes`

The number of bytes this vector uses in memory.

DiscretizationVector.ndim`DiscretizationVector.ndim`

Number of dimensions, always 1.

DiscretizationVector.ntuple`DiscretizationVector.ntuple`

Structure for data storage.

DiscretizationVector.shape`DiscretizationVector.shape`

Number of entries per axis, equals (size,) for linear storage.

DiscretizationVector.size

`DiscretizationVector.size`
size of data storage.

DiscretizationVector.space

`DiscretizationVector.space`
Space to which this vector.

DiscretizationVector.ufunc

`DiscretizationVector.ufunc`
NtuplesBaseUFuncs, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

<code>__eq__(other)</code>	Return <code>vec == other</code> .
<code>__getitem__(indices)</code>	Return <code>self[indices]</code> .
<code>__setitem__(indices, values)</code>	Implement <code>self[indices] = values</code> .
<code>asarray([out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to <code>self</code> .
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>restriction(ufunc)</code>	Restrict a continuous function and assign to this vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, show, fig])</code>	Display the function graphically.

DiscretizationVector.__eq__

`DiscretizationVector.__eq__(other)`
Return `vec == other`.

Returnsequals : bool

True if all entries of `other` are equal to this vector's entries, False otherwise.

DiscretizationVector.__getitem__

`DiscretizationVector.__getitem__(indices)`
Return `self[indices]`.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : *NtuplesBaseVector*

The value(s) at the index (indices)

DiscretizationVector.__setitem__

DiscretizationVector.**__setitem__** (*indices, values*)

Implement `self[indices] = values`.

Parameters*indices* : int or slice

The position(s) that should be set

values : scalar, *array-like* or *NtuplesBaseVector*

The value(s) that are to be assigned.

If *index* is an int, value must be single value.

If *index* is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

DiscretizationVector.asarray

DiscretizationVector.**asarray** (*out=None*)

Extract the data of this array as a numpy array.

Parameters*out* : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

DiscretizationVector.assign

DiscretizationVector.**assign** (*other*)

Assign the values of *other* to self.

DiscretizationVector.copy

DiscretizationVector.**copy** ()

Create an identical (deep) copy of this vector.

DiscretizationVector.dist

DiscretizationVector.**dist** (*other*)

Distance to *other*.

LinearSpace.dist

DiscretizationVector.divide

DiscretizationVector.**divide** (*x, y*)

Divide by *other* inplace.

LinearSpace.divide

DiscretizationVector.inner

DiscretizationVector.**inner** (*other*)

Inner product with *other*.

LinearSpace.inner

DiscretizationVector.lincomb

DiscretizationVector.**lincomb**(a, x1, b=None, x2=None)

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

DiscretizationVector.multiply

DiscretizationVector.**multiply**(x, y)

Multiply by other inplace.

LinearSpace.multiply

DiscretizationVector.norm

DiscretizationVector.**norm**()

Norm of vector

LinearSpace.norm

DiscretizationVector.restriction

DiscretizationVector.**restriction**(ufunc)

Restrict a continuous function and assign to this vector

Parameters**ufunc**: self.space.uspace element

The continuous function that should be restricted.

See also:

RawDiscretization.restriction For full description

Examples

```
>>> import odl
>>> import numpy as np
```

Create discretization

```
>>> X = odl.uniform_discr(0, 1, 5)
>>> x = X.element()
```

Assign x according to continuous vector

```
>>> x.restriction(lambda x: x)
>>> print(x) # Print values at gridpoints (which are centered)
[0.1, 0.3, 0.5, 0.7, 0.9]
```

DiscretizationVector.set_zero

DiscretizationVector.**set_zero**()

Set this vector to zero.

LinearSpace.zero

DiscretizationVector.show

`DiscretizationVector.show` (*title=None, method='scatter', show=False, fig=None, **kwargs*)

Display the function graphically.

Parameter`title` : str, optional

Set the title of the figure

method : str, optional

Id methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.Figure

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.

kwargs : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig` : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

See also:

[`odl.util.graphics.show_discrete_data`](#) Underlying implementation

`__init__` (*space, data*)

Initialize a new instance.

RawDiscretization

class `odl.discr.discretization.RawDiscretization` (*uspace, dspace, restr=None, ext=None*)

Bases: [`odl.space.base_ntuples.NtuplesBase`](#)

Abstract raw discretization class.

A discretization in ODL is a way to encode the transition from an arbitrary set to a set of n-tuples explicitly representable in a computer. The most common use case is the discretization of an infinite-dimensional vector space of functions by means of storing coefficients in a finite basis.

The minimal information required to create a discretization is the set to be discretized (“undiscretized space”) and a backend for storage and processing of the n-tuples (“data space” or “discretized space”).

As additional information, two mappings can be provided. The first one is an explicit way to map an (abstract) element from the source set to an n-tuple. This mapping is called **restriction** in ODL. The second one encodes the converse way of mapping an n-tuple to an element of the original set. This mapping is called **extension**.

Attributes

<i>domain</i>	The domain of the continuous space.
<i>dspace</i>	The data space.
<i>dspace_type</i>	Data space type of this discretization.
<i>dtype</i>	The data type of each entry.
<i>element_type</i>	<i>RawDiscretizationVector</i>
<i>extension</i>	The operator mapping an n-tuple to a <i>uspace</i> element.
<i>restriction</i>	The operator mapping a <i>uspace</i> element to an n-tuple.
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.
<i>uspace</i>	The undiscretized space.

RawDiscretization.domain`RawDiscretization.domain`

The domain of the continuous space.

RawDiscretization.dspace`RawDiscretization.dspace`

The data space.

RawDiscretization.dspace_type`RawDiscretization.dspace_type`

Data space type of this discretization.

RawDiscretization.dtype`RawDiscretization.dtype`

The data type of each entry.

RawDiscretization.element_type`RawDiscretization.element_type`*RawDiscretizationVector***RawDiscretization.extension**`RawDiscretization.extension`The operator mapping an n-tuple to a *uspace* element.**RawDiscretization.restriction**`RawDiscretization.restriction`The operator mapping a *uspace* element to an n-tuple.**RawDiscretization.shape**`RawDiscretization.shape`

The shape of this space.

RawDiscretization.size`RawDiscretization.size`

The number of entries per tuple.

RawDiscretization.uspace`RawDiscretization.uspace`

The undiscretized space.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>element([inp])</code>	Create an element from inp or from scratch.

RawDiscretization.__contains__`RawDiscretization.__contains__(other)`

Return other in self.

Returns`contains` : bool

True if other is an `NtuplesBaseVector` instance and `other.space` is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

RawDiscretization.__eq__`RawDiscretization.__eq__(other)`

Return self == other.

Returns`equals` : bool

True if other is a `RawDiscretization` instance and all attributes `uspace`, `dspace`, `RawDiscretization.restriction` and `RawDiscretization.extension` of other and this discretization are equal, False otherwise.

RawDiscretization.contains_all`RawDiscretization.contains_all(other)`

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

RawDiscretization.contains_set

`RawDiscretization.contains_set (other)`

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

RawDiscretization.element

`RawDiscretization.element (inp=None)`

Create an element from `inp` or from scratch.

Parameters`inp` : object, optional

The input data to create an element from. Must be recognizable by the `LinearSpace.element` method of either `dspace` or `uspace`.

Returns`element` : `RawDiscretizationVector`

The discretized element, calculated as `dspace.element (inp)` or `restriction (uspace.element (inp))`, tried in this order.

__init__ (`uspace`, `dspace`, `restr=None`, `ext=None`)

Abstract initialization method.

Intended to be called by subclasses for proper type checking and setting of attributes.

Parameters`uspace` : `Set`

The undiscretized (abstract) set to be discretized

dspace : `NtuplesBase`

Data space providing containers for the values of a discretized object

restr : `Operator`, optional

Operator mapping a `uspace` element to a `dspace` element. Must satisfy `restr.domain == uspace`, `restr.range == dspace`.

ext : `Operator`, optional

Operator mapping a `dspace` element to a `uspace` element. Must satisfy `ext.domain == dspace`, `ext.range == uspace`.

RawDiscretizationVector

class `odl.discr.discretization.RawDiscretizationVector (space, ntuple)`

Bases: `odl.space.base_ntuples.NtuplesBaseVector`

Representation of a `RawDiscretization` element.

Basically only a wrapper class for `dspace`'s vector class.

Attributes

<code>dtype</code>	type of data storage.
<code>extension</code>	The extension operator associated with this vector.
<code>itemsizes</code>	The size in bytes on one element of this type.
<code>nbytes</code>	The number of bytes this vector uses in memory.
<code>ndim</code>	Number of dimensions, always 1.
Continued on next page	

Table 2.34 – continued from previous page

<i>ntuple</i>	Structure for data storage.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	size of data storage.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>NtuplesBaseUFuncs</i> , access to numpy style ufuncs.

RawDiscretizationVector.dtype

`RawDiscretizationVector.dtype`
type of data storage.

RawDiscretizationVector.extension

`RawDiscretizationVector.extension`

The extension operator associated with this vector.

Return`extension_op` : *FunctionSetMapping*

Operator representing a continuous extension of this vector.

See also:

[RawDiscretization.extension](#) For full description

Examples

```
>>> import odl
>>> import numpy as np
```

Create continuous extension of 1d function with nearest neighbour

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True)
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.,  1.])
```

Create continuous extension of 1d function wiht linear interpolation

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True, interp='linear')
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.48,  0.52])
```

RawDiscretizationVector.itemsize

`RawDiscretizationVector.itemsize`
The size in bytes on one element of this type.

RawDiscretizationVector.nbytes

`RawDiscretizationVector.nbytes`
The number of bytes this vector uses in memory.

RawDiscretizationVector.ndim

`RawDiscretizationVector.ndim`
Number of dimensions, always 1.

RawDiscretizationVector.ntupleRawDiscretizationVector.**ntuple**

Structure for data storage.

RawDiscretizationVector.shapeRawDiscretizationVector.**shape**

Number of entries per axis, equals (size,) for linear storage.

RawDiscretizationVector.sizeRawDiscretizationVector.**size**

size of data storage.

RawDiscretizationVector.spaceRawDiscretizationVector.**space**

Space to which this vector.

RawDiscretizationVector.ufuncRawDiscretizationVector.**ufunc***NtuplesBaseUFuncs*, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

<code>__eq__(other)</code>	Return <code>vec == other</code> .
<code>__getitem__(indices)</code>	Return <code>self[indices]</code> .
<code>__setitem__(indices, values)</code>	Implement <code>self[indices] = values</code> .
<code>asarray([out])</code>	Extract the data of this array as a numpy array.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>restriction(ufunc)</code>	Restrict a continuous function and assign to this vector
<code>show([title, method, show, fig])</code>	Display the function graphically.

RawDiscretizationVector.__eq__RawDiscretizationVector.**__eq__**(*other*)Return `vec == other`.**Return** `sequals` : boolTrue if all entries of `other` are equal to this vector's entries, False otherwise.**RawDiscretizationVector.__getitem__**RawDiscretizationVector.**__getitem__**(*indices*)Return `self[indices]`.**Parameters** `indices` : int or slice

The position(s) that should be accessed

Returns `values` : *NtuplesBaseVector*The value(s) at the index (`indices`)

RawDiscretizationVector.__setitem__

`RawDiscretizationVector.__setitem__(indices, values)`

Implement `self[indices] = values`.

Parameters`indices` : int or slice

The position(s) that should be set

values : scalar, *array-like* or *NtuplesBaseVector*

The value(s) that are to be assigned.

If `index` is an int, value must be single value.

If `index` is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

RawDiscretizationVector.asarray

`RawDiscretizationVector.asarray(out=None)`

Extract the data of this array as a numpy array.

Parameters`out` : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

RawDiscretizationVector.copy

`RawDiscretizationVector.copy()`

Create an identical (deep) copy of this vector.

RawDiscretizationVector.restriction

`RawDiscretizationVector.restriction(ufunc)`

Restrict a continuous function and assign to this vector

Parameters`ufunc` : `self.space.uspace` element

The continuous function that should be restricted.

See also:

[*RawDiscretization.restriction*](#) For full description

Examples

```
>>> import odl
>>> import numpy as np
```

Create discretization

```
>>> X = odl.uniform_discr(0, 1, 5)
>>> x = X.element()
```

Assign x according to continuous vector

```
>>> x.restriction(lambda x: x)
>>> print(x) # Print values at gridpoints (which are centered)
[0.1, 0.3, 0.5, 0.7, 0.9]
```


RawDiscretizationVector.show

`RawDiscretizationVector.show` (*title=None*, *method='scatter'*, *show=False*, *fig=None*,
***kwargs*)

Display the function graphically.

Parameter*title* : str, optional

Set the title of the figure

method : str, optional

1d methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.Figure

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that *fig* is the return value from an earlier call to this function.

kwargs : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns*fig* : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

See also:

[`odl.util.graphics.show_discrete_data`](#) Underlying implementation

__init__ (*space, ntuple*)
Initialize a new instance.

Functions

[`dspace_type\(space, impl\[, dtype\]\)`](#) Select the correct corresponding n-tuples space.

`dspace_type`

`odl.discr.discretization.dspace_type` (*space, impl, dtype=None*)
Select the correct corresponding n-tuples space.

Parameters*space* :

Template space from which to infer an adequate data space. If it has a [`LinearSpace.field`](#) attribute, *dtype* must be consistent with it.

impl : {‘numpy’, ‘cuda’}

Implementation backend for the data space

dtype : type, optional

Data type which the space is supposed to use. If `None`, the space type is purely determined from `space` and `impl`. If given, it must be compatible with the field of `space`.

Returns`type` :

Space type selected after the space's field, the backend and the data type

grid

Sparse implementations of n-dimensional sampling grids.

Sampling grids are collections of points in an n-dimensional coordinate space with a certain structure which is exploited to minimize storage.

Classes

<code>RegularGrid(min_pt, max_pt, shape)</code>	An n-dimensional tensor grid with equidistant coordinates.
<code>TensorGrid(*coord_vectors)</code>	An n-dimensional tensor grid.

RegularGrid

class `odl.discr.grid.RegularGrid(min_pt, max_pt, shape)`

Bases: `odl.discr.grid.TensorGrid`

An n-dimensional tensor grid with equidistant coordinates.

This is a sparse representation of an n-dimensional grid defined as the tensor product of n coordinate vectors with equidistant nodes. The grid points are calculated according to the rule:

$$x_j = \text{min_pt} + j * (\text{max_pt} - \text{min_pt}) / (\text{shape} - 1)$$

with elementwise addition and multiplication.

Attributes

<code>center</code>	The center of the grid.
<code>coord_vectors</code>	The coordinate vectors of the grid.
<code>max_pt</code>	Vector containing the maximal grid coordinates per axis.
<code>meshgrid</code>	A grid suitable for function evaluation.
<code>min_pt</code>	Vector containing the minimal grid coordinates per axis.
<code>ndim</code>	The number of dimensions of the grid.
<code>shape</code>	The number of grid points per axis.
<code>size</code>	The total number of grid points.
<code>stride</code>	The step per axis between two neighboring grid points.

RegularGrid.center

`RegularGrid.center`

The center of the grid. Not necessarily a grid point.

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg.center
array([-1.,  1.])
```

RegularGrid.coord_vectors

RegularGrid.coord_vectors

The coordinate vectors of the grid.

Returnscoord_vectors : tuple of numpy.ndarray's

See also:

meshgrid Same result but with nd arrays

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.coord_vectors
>>> x
array([ 0.,  1.])
>>> y
array([-1.,  0.,  2.])
```

RegularGrid.max_pt

RegularGrid.max_pt

Vector containing the maximal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.max_pt
array([ 5.,  2.])
```

RegularGrid.meshgrid

RegularGrid.meshgrid

A grid suitable for function evaluation.

Returnsmeshgrid : tuple of numpy.ndarray

Function evaluation grid with ndim axes

See also:

numpy.meshgrid Coordinate matrices from coordinate vectors. We use indexing='ij' and copy=True

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.meshgrid
>>> x
array([[ 0.],
       [ 1.]])
>>> y
array([[ -1.,  0.,  2.]])
```

Easy function evaluation via broadcasting:

```
>>> x**2 - y**2
array([[ -1.,  0., -4.],
       [ 0.,  1., -3.]])
```

RegularGrid.min_pt

RegularGrid.**min_pt**

Vector containing the minimal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min_pt
array([ 1., -2.]])
```

RegularGrid.ndim

RegularGrid.**ndim**

The number of dimensions of the grid.

RegularGrid.shape

RegularGrid.**shape**

The number of grid points per axis.

RegularGrid.size

RegularGrid.**size**

The total number of grid points.

RegularGrid.stride

RegularGrid.**stride**

The step per axis between two neighboring grid points.

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg.stride
array([ 1.,  2.]])
```

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>__getitem__(slc)</code>	self[slc] implementation.
<code>append(other)</code>	Insert grid other at the end.
<code>approx_contains(other, atol)</code>	Test if other belongs to this grid up to a tolerance.
<code>approx_equals(other, atol)</code>	Test if this grid is equal to another grid.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>convex_hull()</code>	Return the smallest <i>IntervalProd</i> containing this grid.
<code>corner_grid()</code>	Return a grid with only the corner points.
<code>corners([order])</code>	The corner points of the grid in a single array.
<code>element()</code>	An arbitrary element, the minimum coordinates.
<code>extent()</code>	Return the edge lengths of this grid's minimal bounding box.
<code>insert(index, other)</code>	Insert another regular grid before the given index.
<code>is_subgrid(other[, atol])</code>	Test if this grid is contained in another grid.
<code>max(**kwargs)</code>	Return <i>max_pt</i> .
<code>min(**kwargs)</code>	Return <i>min_pt</i> .
<code>points([order])</code>	All grid points in a single array.
<code>squeeze()</code>	Return the grid with removed degenerate dimensions.

RegularGrid.__contains__

RegularGrid.__contains__(other)
Return other in self.

RegularGrid.__eq__

RegularGrid.__eq__(other)
Return self == other.

RegularGrid.__getitem__

RegularGrid.__getitem__(slc)
self[slc] implementation.

Parametersslc : int or slice

Negative indices and None (new axis) are not supported.

Examples

```
>>> g = RegularGrid([-1.5, -3, -1], [-0.5, 7, 4], (2, 3, 6))
>>> g[0, 0, 0]
array([-1.5, -3. , -1. ])
>>> g[:, 0, 0]
RegularGrid([-1.5, -3.0, -1.0], [-0.5, -3.0, -1.0], (2, 1, 1))
```

Ellipsis can be used, too:

```
>>> g[..., ::3]
RegularGrid([-1.5, -3.0, -1.0], [-0.5, 7.0, 2.0], (2, 3, 2))
```

RegularGrid.append`RegularGrid.append(other)`

Insert grid `other` at the end.

Parameters`other` : *IntervalProd*, float or *array-like*

The set to be inserted. A float or array `a` is treated as an `IntervalProd(a, a)`.

See also:

insert

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 2])
>>> g2 = TensorGrid([1], [-6, 15])
>>> g1.append(g2)
TensorGrid([0.0, 1.0], [-1.0, 2.0], [1.0], [-6.0, 15.0])
```

RegularGrid.approx_contains`RegularGrid.approx_contains(other, atol)`

Test if `other` belongs to this grid up to a tolerance.

Parameters`other` : *array-like* or float

The object to test for membership in this grid

atol : float

Allow deviations up to this number in absolute value per vector entry.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.approx_contains([0, 0], atol=0.0)
True
>>> [0, 0] in g # equivalent
True
>>> g.approx_contains([0.1, -0.1], atol=0.0)
False
>>> g.approx_contains([0.1, -0.1], atol=0.15)
True
```

RegularGrid.approx_equals`RegularGrid.approx_equals(other, atol)`

Test if this grid is equal to another grid.

Parameters`other` : object

Object to be tested

atol : float

Allow deviations up to this number in absolute value per vector entry.

Return`equals` : bool

True if *other* is a *TensorGrid* instance with all coordinate vectors equal (up to the given tolerance), to the ones of this grid, otherwise False.

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 0, 2])
>>> g2 = TensorGrid([-0.1, 1.1], [-1, 0.1, 2])
>>> g1.approx_equals(g2, atol=0)
False
>>> g1.approx_equals(g2, atol=0.15)
True
```

RegularGrid.contains_all

`RegularGrid.contains_all(other)`

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

RegularGrid.contains_set

`RegularGrid.contains_set(other)`

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

RegularGrid.convex_hull

`RegularGrid.convex_hull()`

Return the smallest *IntervalProd* containing this grid.

The convex hull of a set is the union of all line segments between points in the set. For a tensor grid, it is the interval product given by the extremal coordinates.

Returns: *IntervalProd*

Interval product defined by the minimum and maximum of the grid

Examples

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g.convex_hull()
IntervalProd([-1.0, 2.0, 5.0, 2.0], [3.0, 4.0, 5.0, 7.0])
```

RegularGrid.corner_grid

`RegularGrid.corner_grid()`

Return a grid with only the corner points.

Returns: *TensorGrid*

Grid with size 2 in non-degenerate dimensions and 1 in degenerate ones

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.corner_grid()
TensorGrid([0.0, 1.0], [-1.0, 2.0])
```

RegularGrid.corners

RegularGrid.**corners** (*order='C'*)

The corner points of the grid in a single array.

Parameters*order* : {'C', 'F'}

Axis ordering in the resulting point array

Returns*corners* : numpy.ndarray

The size of the array is $2^m \times \text{ndim}$, where m is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.corners()
array([[ 0., -1.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  2.]])
>>> g.corners(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

RegularGrid.element

RegularGrid.**element** ()

An arbitrary element, the minimum coordinates.

RegularGrid.extent

RegularGrid.**extent** ()

Return the edge lengths of this grid's minimal bounding box.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.extent()
array([ 4.,  4.])
```

RegularGrid.insert

RegularGrid.**insert** (*index, other*)

Insert another regular grid before the given index.

The given grid (m dimensions) is inserted into the current one (n dimensions) before the given index, resulting in a new *RegularGrid* with $n + m$ dimensions. Note that no changes are made in-place.

Parameters`index` : int

Index of the dimension before which `other` is to be inserted. Negative indices count backwards from `self.ndim`.

other : *TensorGrid*

Grid to be inserted. If a *RegularGrid* is given, the output will be a *RegularGrid*.

Returns`newgrid` : *TensorGrid* or *RegularGrid*

The enlarged grid. If the inserted grid is a *RegularGrid*, so is the return value.

Examples

```
>>> rg1 = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg2 = RegularGrid(-3, 7, 6)
>>> rg1.insert(1, rg2)
RegularGrid([-1.5, -3.0, -1.0], [-0.5, 7.0, 3.0], (2, 6, 3))
```

If `other` is a *TensorGrid*, so is the result:

```
>>> tg = TensorGrid([0, 1, 2])
>>> rg1.insert(2, tg)
TensorGrid([-1.5, -0.5], [-1.0, 1.0, 3.0], [0.0, 1.0, 2.0])
```

RegularGrid.is_subgrid

RegularGrid.**is_subgrid**(*other*, *atol*=0.0)

Test if this grid is contained in another grid.

Parameters`other` :

Check if this object is a subgrid

atol : float

Allow deviations up to this number in absolute value per coordinate vector entry.

Examples

```
>>> rg = RegularGrid([-2, -2], [0, 4], (3, 4))
>>> rg.coord_vectors
(array([-2., -1., 0.]), array([-2., 0., 2., 4.]))
>>> rg_sub = RegularGrid([-1, 2], [0, 4], (2, 2))
>>> rg_sub.coord_vectors
(array([-1., 0.]), array([ 2., 4.]))
>>> rg_sub.is_subgrid(rg)
True
```

Fuzzy check is also possible. Note that the tolerance still applies to the coordinate vectors.

```
>>> rg_sub = RegularGrid([-1.015, 2], [0, 3.99], (2, 2))
>>> rg_sub.is_subgrid(rg, atol=0.01)
False
>>> rg_sub.is_subgrid(rg, atol=0.02)
True
```

RegularGrid.max`RegularGrid.max(**kwargs)`Return `max_pt`.**Parameterskwargs**For duck-typing with `numpy.amax`**See also:**`min`, `odl.set.domain.IntervalProd.max`**Examples**

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.max()
array([ 5.,  2.]
```

Also works with `numpy`

```
>>> import numpy
>>> numpy.max(g)
array([ 5.,  2.]
```

RegularGrid.min`RegularGrid.min(**kwargs)`Return `min_pt`.**Parameterskwargs**For duck-typing with `numpy.amin`**See also:**`max`, `odl.set.domain.IntervalProd.min`**Examples**

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min()
array([ 1., -2.]
```

Also works with `numpy`

```
>>> import numpy
>>> numpy.min(g)
array([ 1., -2.]
```

RegularGrid.points`RegularGrid.points(order='C')`

All grid points in a single array.

Parameters`order` : {'C', 'F'}

Axis ordering in the resulting point array

Returns`points` : `numpy.ndarray`

The shape of the array is `size x ndim`, i.e. the points are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.points()
array([[ 0., -1.],
       [ 0.,  0.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  0.],
       [ 1.,  2.]])
>>> g.points(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

RegularGrid.squeeze

`RegularGrid.squeeze()`

Return the grid with removed degenerate dimensions.

Returnssqueezed : *RegularGrid*

The squeezed grid

Examples

```
>>> g = RegularGrid([0, 0, 0], [1, 0, 1], (5, 1, 5))
>>> g.squeeze()
RegularGrid([0.0, 0.0], [1.0, 1.0], (5, 5))
```

__init__ (*min_pt, max_pt, shape*)

Initialize a new instance.

Parameters*min_pt* : *array-like* or float

Grid point with minimum coordinates, can be a single float for 1D grids

max_pt : *array-like* or float

Grid point with maximum coordinates, can be a single float for 1D grids

shape : *array-like* or int

The number of grid points per axis, can be an integer for 1D grids

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg
RegularGrid([-1.5, -1.0], [-0.5, 3.0], (2, 3))
>>> rg.coord_vectors
(array([-1.5, -0.5]), array([-1.,  1.,  3.]])
```

```
>>> rg.ndim, rg.size
(2, 6)
```

TensorGrid

class odl.discr.grid.**TensorGrid**(**coord_vectors*)

Bases: *odl.set.sets.Set*

An n-dimensional tensor grid.

A tensor grid is the set of points defined by all possible combination of coordinates taken from fixed coordinate vectors.

In 2 dimensions, for example, given two coordinate vectors:

```
coord_vec1 = [0, 1]
coord_vec2 = [-1, 0, 2]
```

the corresponding tensor grid is the set of all 2d points whose first component is from `coord_vec1` and the second component from `coord_vec2`. The total number of such points is $2 * 3 = 6$:

```
points = [[0, -1], [0, 0], [0, 2],
          [1, -1], [1, 0], [1, 2]]
```

Note that this is the standard ‘C’ ordering where the first axis (`coord_vec1`) varies slowest. Ordering is only relevant when the point array is actually created; the grid itself is independent of this ordering.

The storage need for a tensor grid is only the sum of the lengths of the coordinate vectors, while the total number of points is the product of these lengths. This class makes use of this sparse storage.

Attributes

<i>coord_vectors</i>	The coordinate vectors of the grid.
<i>max_pt</i>	Vector containing the maximal grid coordinates per axis.
<i>meshgrid</i>	A grid suitable for function evaluation.
<i>min_pt</i>	Vector containing the minimal grid coordinates per axis.
<i>ndim</i>	The number of dimensions of the grid.
<i>shape</i>	The number of grid points per axis.
<i>size</i>	The total number of grid points.

TensorGrid.coord_vectors

TensorGrid.**coord_vectors**

The coordinate vectors of the grid.

Returns`coord_vectors` : tuple of `numpy.ndarray`’s

See also:

meshgrid Same result but with nd arrays

Examples

```

>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.coord_vectors
>>> x
array([ 0.,  1.])
>>> y
array([-1.,  0.,  2.])

```

TensorGrid.max_pt

TensorGrid.**max_pt**

Vector containing the maximal grid coordinates per axis.

Examples

```

>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.max_pt
array([ 5.,  2.])

```

TensorGrid.meshgrid

TensorGrid.**meshgrid**

A grid suitable for function evaluation.

Returns meshgrid: tuple of `numpy.ndarray`

Function evaluation grid with `ndim` axes

See also:

numpy.meshgrid Coordinate matrices from coordinate vectors. We use `indexing='ij'` and `copy=True`

Examples

```

>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.meshgrid
>>> x
array([[ 0.],
       [ 1.]])
>>> y
array([[ -1.],
       [  0.],
       [  2.]])

```

Easy function evaluation via broadcasting:

```

>>> x**2 - y**2
array([[ -1.,  0., -4.],
       [  0.,  1., -3.]])

```

TensorGrid.min_pt

TensorGrid.**min_pt**

Vector containing the minimal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min_pt
array([ 1., -2.])
```

TensorGrid.ndim

TensorGrid.**ndim**

The number of dimensions of the grid.

TensorGrid.shape

TensorGrid.**shape**

The number of grid points per axis.

TensorGrid.size

TensorGrid.**size**

The total number of grid points.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>__getitem__(slc)</code>	Return self[slc].
<code>append(other)</code>	Insert grid other at the end.
<code>approx_contains(other, atol)</code>	Test if other belongs to this grid up to a tolerance.
<code>approx_equals(other, atol)</code>	Test if this grid is equal to another grid.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>convex_hull()</code>	Return the smallest <i>IntervalProd</i> containing this grid.
<code>corner_grid()</code>	Return a grid with only the corner points.
<code>corners([order])</code>	The corner points of the grid in a single array.
<code>element()</code>	An arbitrary element, the minimum coordinates.
<code>extent()</code>	Return the edge lengths of this grid's minimal bounding box.
<code>insert(index, other)</code>	Return a copy with other inserted before index.
<code>is_subgrid(other[, atol])</code>	Test if this grid is contained in another grid.
<code>max(**kwargs)</code>	Return <i>max_pt</i> .
<code>min(**kwargs)</code>	Return <i>min_pt</i> .
<code>points([order])</code>	All grid points in a single array.
<code>squeeze()</code>	Return the grid with removed degenerate (length 1) dimensions.

TensorGrid.__contains__

TensorGrid.**__contains__**(other)

Return other in self.

TensorGrid.__eq__

TensorGrid.**__eq__**(other)

Return self == other.

TensorGrid.__getitem__TensorGrid.**__getitem__**(*slc*)

Return self[slc].

Parameters*slc* : int or slice

Negative indices and None (new axis) are not supported.

Examples

```

>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g[0, 0, 0, 0]
array([-1.,  2.,  5.,  2.])
>>> g[:, 0, 0, 0]
TensorGrid([-1.0, 0.0, 3.0], [2.0], [5.0], [2.0])
>>> g[0, ..., 1:]
TensorGrid([-1.0], [2.0, 4.0], [5.0], [4.0, 7.0])
>>> g[:, 2, ..., ::2]
TensorGrid([-1.0, 3.0], [2.0, 4.0], [5.0], [2.0, 7.0])

```

TensorGrid.appendTensorGrid.**append**(*other*)Insert grid *other* at the end.**Parameters***other* : *IntervalProd*, float or *array-like*The set to be inserted. A float or array *a* is treated as an *IntervalProd*(*a*, *a*).**See also:***insert***Examples**

```

>>> g1 = TensorGrid([0, 1], [-1, 2])
>>> g2 = TensorGrid([1], [-6, 15])
>>> g1.append(g2)
TensorGrid([0.0, 1.0], [-1.0, 2.0], [1.0], [-6.0, 15.0])

```

TensorGrid.approx_containsTensorGrid.**approx_contains**(*other*, *atol*)Test if *other* belongs to this grid up to a tolerance.**Parameters***other* : *array-like* or float

The object to test for membership in this grid

atol : float

Allow deviations up to this number in absolute value per vector entry.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.approx_contains([0, 0], atol=0.0)
True
>>> [0, 0] in g # equivalent
True
>>> g.approx_contains([0.1, -0.1], atol=0.0)
False
>>> g.approx_contains([0.1, -0.1], atol=0.15)
True
```

TensorGrid.approx_equals

TensorGrid.**approx_equals** (*other*, *atol*)

Test if this grid is equal to another grid.

Parameters*other* : object

Object to be tested

atol : float

Allow deviations up to this number in absolute value per vector entry.

Return*equals* : bool

True if *other* is a *TensorGrid* instance with all coordinate vectors equal (up to the given tolerance), to the ones of this grid, otherwise False.

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 0, 2])
>>> g2 = TensorGrid([-0.1, 1.1], [-1, 0.1, 2])
>>> g1.approx_equals(g2, atol=0)
False
>>> g1.approx_equals(g2, atol=0.15)
True
```

TensorGrid.contains_all

TensorGrid.**contains_all** (*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

TensorGrid.contains_set

TensorGrid.**contains_set** (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

TensorGrid.convex_hull

TensorGrid.**convex_hull** ()

Return the smallest *IntervalProd* containing this grid.

The convex hull of a set is the union of all line segments between points in the set. For a tensor grid, it is the interval product given by the extremal coordinates.

Return*shull* : *IntervalProd*

Interval product defined by the minimum and maximum of the grid

Examples

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g.convex_hull()
IntervalProd([-1.0, 2.0, 5.0, 2.0], [3.0, 4.0, 5.0, 7.0])
```

TensorGrid.corner_grid

TensorGrid.**corner_grid**()

Return a grid with only the corner points.

Returns *grid* : *TensorGrid*

Grid with size 2 in non-degenerate dimensions and 1 in degenerate ones

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.corner_grid()
TensorGrid([0.0, 1.0], [-1.0, 2.0])
```

TensorGrid.corners

TensorGrid.**corners** (*order*='C')

The corner points of the grid in a single array.

Parameters *order* : {'C', 'F'}

Axis ordering in the resulting point array

Returns *corners* : *numpy.ndarray*

The size of the array is $2^m \times \text{ndim}$, where m is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.corners()
array([[ 0., -1.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  2.]])
>>> g.corners(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

TensorGrid.element

TensorGrid.**element**()

An arbitrary element, the minimum coordinates.

TensorGrid.extent`TensorGrid.extent()`

Return the edge lengths of this grid's minimal bounding box.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.extent()
array([ 4.,  4.])
```

TensorGrid.insert`TensorGrid.insert(index, other)`

Return a copy with `other` inserted before `index`.

The given grid (`m` dimensions) is inserted into the current one (`n` dimensions) before the given index, resulting in a new *TensorGrid* with `n + m` dimensions. Note that no changes are made in-place.

Parameters`index` : int

The index of the dimension before which `other` is to be inserted. Negative indices count backwards from `self.ndim`.

other : *TensorGrid*, float or *array-like*

The grid to be inserted

Returns`newgrid` : *TensorGrid*

The enlarged grid

See also:

append

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 2])
>>> g2 = TensorGrid([1], [-6, 15])
>>> g1.insert(1, g2)
TensorGrid([0.0, 1.0], [1.0], [-6.0, 15.0], [-1.0, 2.0])
```

TensorGrid.is_subgrid`TensorGrid.is_subgrid(other, atol=0.0)`

Test if this grid is contained in another grid.

Parameters`other` : *TensorGrid*

The other grid which is supposed to contain this grid

atol : float

Allow deviations up to this number in absolute value per coordinate vector entry.

Examples

```

>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g_sub = TensorGrid([0], [-1, 2])
>>> g_sub.is_subgrid(g)
True
>>> g_sub = TensorGrid([0.1], [-1.05, 2.1])
>>> g_sub.is_subgrid(g)
False
>>> g_sub.is_subgrid(g, atol=0.15)
True

```

TensorGrid.max

TensorGrid.**max**(*kwargs)

Return *max_pt*.

Parameterskwargs

For duck-typing with `numpy.amax`

See also:

min, *odl.set.domain.IntervalProd.max*

Examples

```

>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.max()
array([ 5.,  2.])

```

Also works with `numpy`

```

>>> import numpy
>>> numpy.max(g)
array([ 5.,  2.])

```

TensorGrid.min

TensorGrid.**min**(*kwargs)

Return *min_pt*.

Parameterskwargs

For duck-typing with `numpy.amin`

See also:

max, *odl.set.domain.IntervalProd.min*

Examples

```

>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min()
array([ 1., -2.])

```

Also works with `numpy`

```
>>> import numpy
>>> numpy.min(g)
array([ 1., -2.])
```

TensorGrid.points

TensorGrid.**points** (*order='C'*)

All grid points in a single array.

Parameters*order* : {'C', 'F'}

Axis ordering in the resulting point array

Returns*points* : numpy.ndarray

The shape of the array is *size* x *ndim*, i.e. the points are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.points()
array([[ 0., -1.],
       [ 0.,  0.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  0.],
       [ 1.,  2.]])
>>> g.points(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

TensorGrid.squeeze

TensorGrid.**squeeze** ()

Return the grid with removed degenerate (length 1) dimensions.

Returns*squeezed* : *TensorGrid*

The squeezed grid

Examples

```
>>> g = TensorGrid([0, 1], [-1], [-1, 0, 2])
>>> g.squeeze()
TensorGrid([0.0, 1.0], [-1.0, 0.0, 2.0])
```

__init__ (**coord_vectors*)

Initialize a TensorGrid instance.

Parameters*vec1,...,vecN* : *array-like*

The coordinate vectors defining the grid points. They must be sorted in ascending order and may not contain duplicates. Empty vectors are not allowed.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g
TensorGrid([1.0, 2.0, 5.0], [-2.0, 1.5, 2.0])
>>> print(g)
grid [1.0, 2.0, 5.0] x [-2.0, 1.5, 2.0]
>>> g.ndim # number of axes
2
>>> g.shape # points per axis
(3, 3)
>>> g.size # total number of points
9
```

Grid points can be extracted with index notation (NOTE: This is slow, do not loop over the grid using indices!):

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g[0, 0, 0, 0]
array([-1., 2., 5., 2.])
```

Slices and ellipsis are also supported:

```
>>> g[:, 0, 0, 0]
TensorGrid([-1.0, 0.0, 3.0], [2.0], [5.0], [2.0])
>>> g[0, ..., 1:]
TensorGrid([-1.0], [2.0, 4.0], [5.0], [4.0, 7.0])
```

Functions

<code>sparse_meshgrid(*x)</code>	Make a sparse <i>meshgrid</i> by adding empty dimensions.
<code>uniform_sampling(begin, end, num_nodes)</code>	Sample an implicitly defined interval product uniformly.
<code>uniform_sampling_fromintv(intv_prod, num_nodes)</code>	Sample an interval product uniformly.

sparse_meshgrid

`odl.discr.grid.sparse_meshgrid(*x)`

Make a sparse *meshgrid* by adding empty dimensions.

Parameters`x1,...,xN`: *array-like*

Input arrays to turn into sparse meshgrid vectors

Returns`meshgrid`: tuple of `numpy.ndarray`

Sparse coordinate vectors representing an N-dimensional grid

See also:

`numpy.meshgrid` dense or sparse meshgrids

Examples

```
>>> x, y = [0, 1], [2, 3, 4]
>>> mesh = sparse_meshgrid(x, y)
>>> sum(xi for xi in mesh).ravel() # first axis slowest
array([2, 3, 4, 3, 4, 5])
```

uniform_sampling

`odl.discr.grid.uniform_sampling(begin, end, num_nodes)`

Sample an implicitly defined interval product uniformly.

Parameters`begin` : *array-like* or float

The lower ends of the intervals in the product

`end` : *array-like* or float

The upper ends of the intervals in the product

`num_nodes` : int or tuple of int

Number of nodes per axis. For dimension ≥ 2 , a tuple is required. All entries must be positive. Entries corresponding to degenerate axes must be equal to 1.

See also:

[`uniform_sampling_fromintv`](#) sample a given interval product

[`odl.discr.partition.uniform_partition`](#) divide implicitly defined interval product into equally sized subsets

Examples

```
>>> grid = uniform_sampling([-1.5, 2], [-0.5, 3], (3, 3))
>>> grid.coord_vectors
(array([-1.5, -1. , -0.5]), array([ 2. ,  2.5,  3. ]))
```

uniform_sampling_fromintv

`odl.discr.grid.uniform_sampling_fromintv(intv_prod, num_nodes)`

Sample an interval product uniformly.

The resulting grid will include `begin` and `end` of `intv_prod` as grid points. If you want a subdivision into equally sized cells with grid points in the middle, use [`uniform_partition`](#) instead.

Parameters`intv_prod` : *IntervalProd*

Set to be sampled

`num_nodes` : int or tuple of int

Number of nodes per axis. For dimension ≥ 2 , a tuple is required. All entries must be positive. Entries corresponding to degenerate axes must be equal to 1.

Return`sampling` : *RegularGrid*

Uniform sampling grid for the interval product

See also:

`uniform_sampling` Sample an implicitly created `IntervalProd`

Examples

```
>>> from odl import IntervalProd
>>> rbox = IntervalProd([-1.5, 2], [-0.5, 3])
>>> grid = uniform_sampling_fromintv(rbox, (3, 3))
>>> grid.coord_vectors
(array([-1.5, -1. , -0.5]), array([ 2. ,  2.5,  3. ]))
```

lp_discr

L^p type discretizations of function spaces.

Classes

<code>DiscreteLp(fspace, partition, dspace[, ...])</code>	Discretization of a Lebesgue L^p space.
<code>DiscreteLpVector(space, data)</code>	Representation of a <code>DiscreteLp</code> element.

DiscreteLp

class odl.dscr.lp_discr.**DiscreteLp** (*fspace, partition, dspace, exponent=2.0, interp='nearest', **kwargs*)

Bases: `odl.dscr.discretization.Discretization`

Discretization of a Lebesgue L^p space.

Attributes

<code>cell_sides</code>	Side lengths of a cell in an underlying <i>uniform</i> partition.
<code>cell_volume</code>	Cell volume of an underlying regular partition.
<code>domain</code>	The domain of the continuous space.
<code>dspace</code>	The data space.
<code>dspace_type</code>	Data space type of this discretization.
<code>dtype</code>	The data type of each entry.
<code>element_type</code>	<code>DiscreteLpVector</code>
<code>exponent</code>	The exponent p in L^p .
<code>extension</code>	The operator mapping an n -tuple to a <i>uspace</i> element.
<code>field</code>	The field of this vector space.
<code>grid</code>	Sampling grid of the discretization mappings.
<code>interp</code>	Interpolation type of this discretization.
<code>is_cn</code>	Return True if the space represents C^n , i.e.
<code>is_rn</code>	Return True if the space represents R^n , i.e.
<code>is_weighted</code>	Return True if the <code>dspace</code> is weighted.
<code>meshgrid</code>	All sampling points in the partition as a sparse meshgrid.
<code>ndim</code>	Number of dimensions.
<code>order</code>	Axis ordering for array flattening.

Continued on next page

Table 2.44 – continued from previous page

<i>partition</i>	The <i>RectPartition</i> of the domain.
<i>restriction</i>	The operator mapping a <i>uspace</i> element to an n-tuple.
<i>shape</i>	Shape of the underlying partition.
<i>size</i>	Total number of underlying partition cells.
<i>uspace</i>	The undiscretized space.
<i>weighting</i>	This space’s weighting scheme.

DiscreteLp.cell_sides**DiscreteLp.cell_sides**Side lengths of a cell in an underlying *uniform* partition.**DiscreteLp.cell_volume****DiscreteLp.cell_volume**

Cell volume of an underlying regular partition.

DiscreteLp.domain**DiscreteLp.domain**

The domain of the continuous space.

DiscreteLp.dspace**DiscreteLp.dspace**

The data space.

DiscreteLp.dspace_type**DiscreteLp.dspace_type**

Data space type of this discretization.

DiscreteLp.dtype**DiscreteLp.dtype**

The data type of each entry.

DiscreteLp.element_type**DiscreteLp.element_type***DiscreteLpVector***DiscreteLp.exponent****DiscreteLp.exponent**The exponent p in L^p .**DiscreteLp.extension****DiscreteLp.extension**The operator mapping an n-tuple to a *uspace* element.

DiscreteLp.field`DiscreteLp.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

DiscreteLp.grid`DiscreteLp.grid`

Sampling grid of the discretization mappings.

DiscreteLp.interp`DiscreteLp.interp`

Interpolation type of this discretization.

DiscreteLp.is_cn`DiscreteLp.is_cn`

Return True if the space represents C^n , i.e. complex tuples.

DiscreteLp.is_rn`DiscreteLp.is_rn`

Return True if the space represents R^n , i.e. real tuples.

DiscreteLp.is_weighted`DiscreteLp.is_weighted`

Return True if the dspace is weighted.

DiscreteLp.meshgrid`DiscreteLp.meshgrid`

All sampling points in the partition as a sparse meshgrid.

DiscreteLp.ndim`DiscreteLp.ndim`

Number of dimensions.

DiscreteLp.order`DiscreteLp.order`

Axis ordering for array flattening.

DiscreteLp.partition`DiscreteLp.partition`

The *RectPartition* of the domain.

DiscreteLp.restriction`DiscreteLp.restriction`The operator mapping a *uspace* element to an n-tuple.**DiscreteLp.shape**`DiscreteLp.shape`

Shape of the underlying partition.

DiscreteLp.size`DiscreteLp.size`

Total number of underlying partition cells.

DiscreteLp.uspace`DiscreteLp.uspace`

The undiscretized space.

DiscreteLp.weighting`DiscreteLp.weighting`

This space's weighting scheme.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>_dist(x, y)</code>	Return self.dist(x, y).
<code>_divide(x1, x2, out)</code>	Raw pointwise multiplication of two vectors.
<code>_inner(x, y)</code>	Return self.inner(x, y).
<code>_lincomb(a, x1, b, x2, out)</code>	Raw linear combination.
<code>_multiply(x1, x2, out)</code>	Raw pointwise multiplication of two vectors.
<code>_norm(x)</code>	Return self.norm(x).
<code>astype(dtype)</code>	Return a copy of this space with new dtype.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp])</code>	Create an element from inp or from scratch.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create a vector of ones.
<code>points()</code>	All sampling points in the partition.
<code>zero()</code>	Create a vector of zeros.

DiscreteLp.__contains__`DiscreteLp.__contains__(other)`

Return other in self.

Returnscontains : bool

True if other is an *NtuplesBaseVector* instance and other.space is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

DiscreteLp.__eq__

DiscreteLp.__eq__(other)

Return self == other.

Returnsequals : bool

True if other is a *RawDiscretization* instance and all attributes *uspace*, *dspace*, *RawDiscretization.restriction* and *RawDiscretization.extension* of other and this discretization are equal, False otherwise.

DiscreteLp._dist

DiscreteLp._dist(x, y)

Return self.dist(x, y).

DiscreteLp._divide

DiscreteLp._divide(x1, x2, out)

Raw pointwise multiplication of two vectors.

DiscreteLp._inner

DiscreteLp._inner(x, y)

Return self.inner(x, y).

DiscreteLp._lincomb

DiscreteLp._lincomb(a, x1, b, x2, out)

Raw linear combination.

DiscreteLp._multiply

DiscreteLp._multiply(x1, x2, out)

Raw pointwise multiplication of two vectors.

DiscreteLp._norm

DiscreteLp._norm(x)

Return self.norm(x).

DiscreteLp.astype

`DiscreteLp.astype(dtype)`

Return a copy of this space with new dtype.

Parametersdtype :

Data type of the returned space. Can be given in any way `numpy.dtype` understands, e.g. as string ('complex64') or data type (`complex`).

Returnsnewspace : *FnBase*

The version of this space with given data type

DiscreteLp.contains_all

`DiscreteLp.contains_all(other)`

Test if all points in `other` are contained in this set.

This is a default implementation and should be overridden by subclasses.

DiscreteLp.contains_set

`DiscreteLp.contains_set(other)`

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

DiscreteLp.dist

`DiscreteLp.dist(x1, x2)`

Calculate the distance between two vectors.

Parametersx1, x2 : *LinearSpaceVector*

Vectors whose distance to compute

Returnsdist : `float`

Distance between vectors

DiscreteLp.divide

`DiscreteLp.divide(x1, x2, out=None)`

Calculate the pointwise division of `x1` and `x2`

Parametersx1 : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returnsout : *LinearSpaceVector*

Ratio of the vectors. If `out` was provided, the returned object is a reference to it.

DiscreteLp.element

`DiscreteLp.element` (*inp=None*)

Create an element from *inp* or from scratch.

Parameters*inp* : object, optional

The input data to create an element from. Must be recognizable by the `LinearSpace.element` method of either `RawDiscretization.dspace` or `RawDiscretization.uspace`.

Return*element* : `DiscreteLpVector`

The discretized element, calculated as `dspace.element(inp)` or `restriction(uspace.element(inp))`, tried in this order.

DiscreteLp.inner

`DiscreteLp.inner` (*x1, x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1, x2* : `LinearSpaceVector`

Factors in the inner product

Return*out* : `LinearSpace.field` element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

DiscreteLp.lincomb

`DiscreteLp.lincomb` (*a, x1, b=None, x2=None, out=None*)

Linear combination of vectors.

Calculates

$out = a * x1$

or, if *b* and *y* are given,

$out = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : `LinearSpaceVector`

The first of the summands

b : Scalar, optional

Scalar to multiply *x2* with.

x2 : `LinearSpaceVector`, optional

The second of the summands

out : `LinearSpaceVector`, optional

The Vector that the result should be written to.

Return*out* : `LinearSpaceVector`

Result of the linear combination. If *out* was provided, the returned object is a reference to it.

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

DiscreteLp.multiply

`DiscreteLp.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

DiscreteLp.norm

`DiscreteLp.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

DiscreteLp.one

`DiscreteLp.one()`

Create a vector of ones.

DiscreteLp.points

`DiscreteLp.points()`

All sampling points in the partition.

DiscreteLp.zero

`DiscreteLp.zero()`

Create a vector of zeros.

__init__(*fspace, partition, dspace, exponent=2.0, interp='nearest', **kwargs*)

Initialize a new instance.

Parameters`fspace` : *FunctionSpace*

The continuous space to be discretized

partition : *RectPartition*

Partition of (a subset of) `fspace.domain` based on a *TensorGrid*

dspace : *FnBase*

Space of elements used for data storage. It must have the same *FnBase.field* as *fspace*

exponent : positive float, optional

The parameter p in L^p . If the exponent is not equal to the default 2.0, the space has no inner product.

interp : str or sequence of str, optional

The interpolation type to be used for discretization. A sequence is interpreted as interpolation scheme per axis.

‘nearest’ : use nearest-neighbor interpolation (default)

‘linear’ : use linear interpolation

order : {‘C’, ‘F’}, optional

Ordering of the axes in the data storage. ‘C’ means the first axis varies slowest, the last axis fastest; vice versa for ‘F’. Default: ‘C’

DiscreteLpVector

class odl.discr.lp_discr.**DiscreteLpVector**(*space, data*)

Bases: *odl.discr.discretization.DiscretizationVector*

Representation of a *DiscreteLp* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (.
<i>cell_sides</i>	Side lengths of a cell in an underlying <i>uniform</i> partition.
<i>cell_volume</i>	Cell volume of an underlying regular grid.
<i>dtype</i>	type of data storage.
<i>extension</i>	The extension operator associated with this vector.
<i>imag</i>	Imaginary part of this element.
<i>itemsize</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions.
<i>ntuple</i>	Structure for data storage.
<i>order</i>	Axis ordering for array flattening.
<i>real</i>	Real part of this element.
<i>shape</i>	Multi-dimensional shape of this discrete function.
<i>size</i>	size of data storage.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>DiscreteLpUFuncs</i> , access to numpy style ufuncs.

DiscreteLpVector.T

DiscreteLpVector.T

The transpose of a vector, the functional given by (., self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

DiscreteLpVector.cell_sides

`DiscreteLpVector.cell_sides`

Side lengths of a cell in an underlying *uniform* partition.

DiscreteLpVector.cell_volume

`DiscreteLpVector.cell_volume`

Cell volume of an underlying regular grid.

DiscreteLpVector.dtype

`DiscreteLpVector.dtype`

type of data storage.

DiscreteLpVector.extension

`DiscreteLpVector.extension`

The extension operator associated with this vector.

Return`extension_op`: *FunctionSetMapping*

Operator representing a continuous extension of this vector.

See also:

RawDiscretization.extensionFor full description

Examples

```
>>> import odl
>>> import numpy as np
```

Create continuous extension of 1d function with nearest neighbour

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True)
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.,  1.])
```


Create continuous extension of 1d function wiht linear interpolation

```
>>> X = odl.uniform_discr(0, 1, 3, nodes_on_bdry=True, interp='linear')
>>> x = X.element([0, 1, 0])
>>> x.extension(np.array([0.24, 0.26]))
array([ 0.48,  0.52])
```

DiscreteLpVector.imag

`DiscreteLpVector.imag`

Imaginary part of this element.

DiscreteLpVector.itemsize

`DiscreteLpVector.itemsize`

The size in bytes on one element of this type.

DiscreteLpVector.nbytes

`DiscreteLpVector.nbytes`

The number of bytes this vector uses in memory.

DiscreteLpVector.ndim

`DiscreteLpVector.ndim`

Number of dimensions.

DiscreteLpVector.ntuple

`DiscreteLpVector.ntuple`

Structure for data storage.

DiscreteLpVector.order

`DiscreteLpVector.order`

Axis ordering for array flattening.

DiscreteLpVector.real

`DiscreteLpVector.real`

Real part of this element.

DiscreteLpVector.shape

`DiscreteLpVector.shape`

Multi-dimensional shape of this discrete function.

DiscreteLpVector.size

`DiscreteLpVector.size`

size of data storage.

DiscreteLpVector.space

`DiscreteLpVector.space`

Space to which this vector.

DiscreteLpVector.ufuncDiscreteLpVector.**ufunc***DiscreteLpUFuncs*, access to numpy style ufuncs.**Notes**

These are optimized to use the underlying ntuple space and incur no overhead unless these do.

Examples

```
>>> X = uniform_discr(0, 1, 2)
>>> x = X.element([1, -2])
>>> x.ufunc.absolute()
uniform_discr(0.0, 1.0, 2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
uniform_discr(0.0, 1.0, 2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
uniform_discr(0.0, 1.0, 2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = X.element([3, 4])
>>> out = X.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
uniform_discr(0.0, 1.0, 2).element([4.0, 2.0])
>>> result is out
True
```

Methods

<code>__eq__(other)</code>	Return <code>vec == other</code> .
<code>__getitem__(indices)</code>	Return <code>self[indices]</code> .
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to self.
<code>conj([out])</code>	The complex conjugate of this element.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .

Continued on next page

Table 2.47 – continued from previous page

<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>restriction(ufunc)</code>	Restrict a continuous function and assign to this vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, indices, show, fig])</code>	Display the function graphically.

DiscreteLpVector.__eq__

`DiscreteLpVector.__eq__ (other)`

Return `vec == other`.

Returnsequals : bool

True if all entries of `other` are equal to this vector's entries, False otherwise.

DiscreteLpVector.__getitem__

`DiscreteLpVector.__getitem__ (indices)`

Return `self[indices]`.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : *NtuplesBaseVector*

The value(s) at the index (indices)

DiscreteLpVector.__setitem__

`DiscreteLpVector.__setitem__ (indices, values)`

Set values of this vector.

Parametersindices : int or slice

The position(s) that should be set

values : scalar, *array-like* or *NtuplesVector*

The value(s) that are to be assigned. If `indices` is an int, `values` must be a single value. If `indices` is a slice, `values` must be broadcastable to the size of the slice (same size, shape (1,) or single value). For `indices=slice(None)`, i.e. in the call `vec[:] = values`, a multi-dimensional array of correct shape is allowed as `values`.

DiscreteLpVector.asarray

`DiscreteLpVector.asarray (out=None)`

Extract the data of this array as a numpy array.

Parametersout : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype and shape.

DiscreteLpVector.assign

`DiscreteLpVector.assign (other)`

Assign the values of `other` to self.

DiscreteLpVector.conj`DiscreteLpVector.conj(out=None)`

The complex conjugate of this element.

Parameters`out` : *DiscreteLpVector*, optional

Element to which the complex conjugate is written. Must be an element of this vector's space.

Returns`out` : *DiscreteLpVector*The complex conjugate vector. If `out` is provided, the returned object is a reference to it.**Examples**

```
>>> discr = uniform_discr(0, 1, 4, dtype='complex')
>>> x = discr.element([5+1j, 3, 2-2j, 1j])
>>> y = x.conj(); print(y)
[(5-1j), (3-0j), (2+2j), -1j]
```

The `out` parameter allows you to avoid a copy:

```
>>> z = discr.element()
>>> z_out = x.conj(out=z); print(z)
[(5-1j), (3-0j), (2+2j), -1j]
>>> z_out is z
True
```

It can also be used for in-place conjugation:

```
>>> x_out = x.conj(out=x); print(x)
[(5-1j), (3-0j), (2+2j), -1j]
>>> x_out is x
True
```

DiscreteLpVector.copy`DiscreteLpVector.copy()`

Create an identical (deep) copy of this vector.

DiscreteLpVector.dist`DiscreteLpVector.dist(other)`

Distance to other.

*LinearSpace.dist***DiscreteLpVector.divide**`DiscreteLpVector.divide(x, y)`

Divide by other inplace.

LinearSpace.divide

DiscreteLpVector.innerDiscreteLpVector.**inner** (*other*)

Inner product with other.

*LinearSpace.inner***DiscreteLpVector.lincomb**DiscreteLpVector.**lincomb** (*a*, *x1*, *b=None*, *x2=None*)

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.*LinearSpace.lincomb***DiscreteLpVector.multiply**DiscreteLpVector.**multiply** (*x*, *y*)

Multiply by other inplace.

*LinearSpace.multiply***DiscreteLpVector.norm**DiscreteLpVector.**norm** ()

Norm of vector

*LinearSpace.norm***DiscreteLpVector.restriction**DiscreteLpVector.**restriction** (*ufunc*)

Restrict a continuous function and assign to this vector

Parameters*ufunc*: self.space.uspace element

The continuous function that should be restricted.

See also:**RawDiscretization.restriction**For full description**Examples**

```
>>> import odl
>>> import numpy as np
```

Create discretization

```
>>> X = odl.uniform_discr(0, 1, 5)
>>> x = X.element()
```

Assign x according to continuous vector

```
>>> x.restriction(lambda x: x)
>>> print(x) # Print values at gridpoints (which are centered)
[0.1, 0.3, 0.5, 0.7, 0.9]
```

DiscreteLpVector.set_zero`DiscreteLpVector.set_zero()`

Set this vector to zero.

LinearSpace.zero

DiscreteLpVector.show`DiscreteLpVector.show(title=None, method='', indices=None, show=False, fig=None, **kwargs)`

Display the function graphically.

Parameter`title`: str, optional

Set the title of the figure

method: str, optional

1d methods:

‘plot’: graph plot

‘scatter’: scattered 2d points (2nd axis <-> value)

2d methods:

‘imshow’: image plot with coloring according to value, including a colorbar.

‘scatter’: cloud of scattered 3d points (3rd axis <-> value)

indices: index expression, optional

Display a slice of the array instead of the full array. The index expression is most easily created with the `numpy.s_` constructor, i.e. supply `np.s_[:, 1, :]` to display the first slice along the second axis. For data with 3 or more dimensions, the 2d slice in the first two axes at the “middle” along the remaining axes is shown (semantically `[:, :, shape[2:] // 2]`).

show: bool, optional

If the plot should be showed now or deferred until later.

fig: `matplotlib.figure.Figure`

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.

kwargs: {‘figsize’, ‘saveto’, ‘clim’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig`: `matplotlib.figure.Figure`

The resulting figure. It is also shown to the user.

See also:

`odl.util.graphics.show_discrete_data` Underlying implementation

__init__(*space, data*)

Initialize a new instance.

Functions

<code>discr_sequence_space(shape[, exponent, impl])</code>	Return an object mimicing the sequence space $l^p(\mathbb{R}^d)$.
<code>uniform_discr(min_corner, max_corner, nsamples)</code>	Discretize an L_p function space by uniform sampling.
<code>uniform_discr_frompartition(partition[, ...])</code>	Discretize an L_p function space given a uniform partition.
<code>uniform_discr_fromspace(fspace, nsamples[, ...])</code>	Discretize an L_p function space by uniform partition.

discr_sequence_space

`odl.discr.lp_discr.discr_sequence_space(shape, exponent=2.0, impl='numpy', **kwargs)`

Return an object mimicing the sequence space $l^p(\mathbb{R}^d)$.

The returned object is a *DiscreteLp* without restriction and extension operators. It uses a grid with stride 1 and no weighting.

Parameters`shape` : sequence of int

Multi-dimensional size of the elements in this space

exponent : positive float, optional

The parameter p in L^p . If the exponent is not equal to the default 2.0, the space has no inner product.

impl : { 'numpy', 'cuda' }

Implementation of the data storage arrays

dtype : dtype

Data type for the discretized space

Default for 'numpy': 'float64'

Default for 'cuda': 'float32'

order : { 'C', 'F' }, optional

Ordering of the axes in the data storage. 'C' means the first axis varies slowest, the last axis fastest; vice versa for 'F'. Default: 'C'

Returns`seqspc` : *DiscreteLp*

The sequence-space-like discrete L_p

Examples

```
>>> seq_spc = discr_sequence_space((3, 3))
>>> seq_spc.one().norm() == 3.0
True
>>> seq_spc = discr_sequence_space((3, 3), exponent=1)
>>> seq_spc.one().norm() == 9.0
True
```

uniform_discr

`odl.discr.lp_discr.uniform_discr(min_corner, max_corner, nsamples, exponent=2.0, interp='nearest', impl='numpy', **kwargs)`

Discretize an Lp function space by uniform sampling.

Parameters`min_corner` : float or tuple of float

Minimum corner of the result.

`nsamples` : float or tuple of float

Minimum corner of the result.

`nsamples` : int or tuple of int

Number of samples per axis. For dimension ≥ 2 , a tuple is required.

exponent : positive float, optional

The parameter p in L^p . If the exponent is not equal to the default 2.0, the space has no inner product.

interp : str or sequence of str, optional

Interpolation type to be used for discretization. A sequence is interpreted as interpolation scheme per axis.

‘nearest’ : use nearest-neighbor interpolation

‘linear’ : use linear interpolation

impl : {‘numpy’, ‘cuda’}, optional

Implementation of the data storage arrays

nodes_on_bdry : bool or sequence, optional

If a sequence is provided, it determines per axis whether to place the last grid point on the boundary (True) or shift it by half a cell size into the interior (False). In each axis, an entry may consist in a single bool or a 2-tuple of bool. In the latter case, the first tuple entry decides for the left, the second for the right boundary. The length of the sequence must be `array.ndim`.

A single boolean is interpreted as a global choice for all boundaries. Default: `False`

dtype : dtype, optional

Data type for the discretized space

Default for ‘numpy’: ‘float64’ / ‘complex128’

Default for ‘cuda’: ‘float32’

order : {‘C’, ‘F’}, optional

Ordering of the axes in the data storage. ‘C’ means the first axis varies slowest, the last axis fastest; vice versa for ‘F’. Default: ‘C’

weighting : {‘const’, ‘none’}, optional

Weighting of the discretized space functions.

‘const’ : weight is a constant, the cell volume (default)

‘none’ : no weighting

Returns`discr` : *DiscreteLp*

The uniformly discretized function space

See also:

`uniform_discr_frompartition` uniform Lp discretization using a given uniform partition of a function domain

`uniform_discr_fromspace` uniform discretization from an existing function space

Examples

Create real space:

```
>>> uniform_discr([0, 0], [1, 1], [10, 10])
uniform_discr([0.0, 0.0], [1.0, 1.0], [10, 10])
```

Can create complex space by giving a dtype

```
>>> uniform_discr([0, 0], [1, 1], [10, 10], dtype='complex')
uniform_discr([0.0, 0.0], [1.0, 1.0], [10, 10], dtype='complex')
```

`uniform_discr_frompartition`

```
odl.discr.lp_discr.uniform_discr_frompartition(partition,          exponent=2.0,      in-
                                                interp='nearest',    impl='numpy',
                                                **kwargs)
```

Discretize an Lp function space given a uniform partition.

Parameters`partition` : *RectPartition*

Regular (uniform) partition to be used for discretization

exponent : positive float, optional

The parameter p in L^p . If the exponent is not equal to the default 2.0, the space has no inner product.

interp : str or sequence of str, optional

Interpolation type to be used for discretization. A sequence is interpreted as interpolation scheme per axis.

‘nearest’ : use nearest-neighbor interpolation

‘linear’ : use linear interpolation

impl : { ‘numpy’, ‘cuda’ }, optional

Implementation of the data storage arrays

Returns`discr` : *DiscreteLp*

The uniformly discretized function space

Other Parameters`order` : { ‘C’, ‘F’ }, optional

Axis ordering in the data storage. Default: ‘C’

dtype : dtype

Data type for the discretized space

Default for 'numpy': 'float64' / 'complex128'

Default for 'cuda': 'float32'

weighting : {'const', 'none'}, optional

Weighting of the discretized space functions.

'const' : weight is a constant, the cell volume (default)

'none' : no weighting

See also:

[`uniform_discr`](#)implicit uniform Lp discretization

[`uniform_discr_fromspace`](#)uniform Lp discretization from an existing function space

[`odl.discr.partition.uniform_partition`](#)partition of the function domain

Examples

```
>>> from odl import uniform_partition
>>> part = uniform_partition(0, 1, 10)
>>> uniform_discr_frompartition(part)
uniform_discr(0.0, 1.0, 10)
```

`uniform_discr_fromspace`

`odl.discr.lp_discr.uniform_discr_fromspace` (*fspace*, *nsamples*, *exponent*=2.0, *interp*='nearest', *impl*='numpy', ***kwargs*)

Discretize an Lp function space by uniform partition.

Parameters*fspace* : [*FunctionSpace*](#)

Continuous function space. Its domain must be an [*IntervalProd*](#) instance.

nsamples : int or tuple of int

Number of samples per axis. For dimension ≥ 2 , a tuple is required.

exponent : positive float, optional

The parameter p in L^p . If the exponent is not equal to the default 2.0, the space has no inner product.

interp : str or sequence of str, optional

Interpolation type to be used for discretization. A sequence is interpreted as interpolation scheme per axis.

'nearest' : use nearest-neighbor interpolation

'linear' : use linear interpolation

impl : {'numpy', 'cuda'}, optional

Implementation of the data storage arrays

Returns*discr* : [*DiscreteLp*](#)

The uniformly discretized function space

Other Parameters*nodes_on_bdry* : bool or boolean *array-like*, optional

If `True`, place the outermost grid points at the boundary. For `False`, they are shifted by half a cell size to the ‘inner’. If an array-like is given, it must have shape `(ndim, 2)`, where `ndim` is the number of dimensions. It defines per axis whether the leftmost (first column) and rightmost (second column) nodes lie on the boundary. Default: `False`

order : {‘C’, ‘F’}, optional

Axis ordering in the data storage. Default: ‘C’

dtype : dtype, optional

Data type for the discretized space. If not specified, the `FunctionSpace.out_dtype` of `fspace` is used.

weighting : {‘const’, ‘none’}, optional

Weighting of the discretized space functions.

‘const’ : weight is a constant, the cell volume (default)

‘none’ : no weighting

See also:

`uniform_discr` implicit uniform Lp discretization

`uniform_discr_frompartition` uniform Lp discretization using a given uniform partition of a function domain

`odl.discr.partition.uniform_partition` partition of the function domain

Examples

```
>>> from odl import Interval, FunctionSpace
>>> intv = Interval(0, 1)
>>> space = FunctionSpace(intv)
>>> uniform_discr_fromspace(space, 10)
uniform_discr(0.0, 1.0, 10)
```

partition

Partitons of interval products based on tensor grids.

A partition of a set is a finite collection of nonempty, pairwise disjoint subsets whose union is the original set. The partitions considered here are based on hypercubes, i.e. the tensor products of partitions of intervals.

Classes

`RectPartition(intv_prod, grid)` Rectangular partition by hypercubes based on `TensorGrid`.

RectPartition

class `odl.discr.partition.RectPartition(intv_prod, grid)`
 Bases: `object`

Rectangular partition by hypercubes based on *TensorGrid*.

In 1d, a partition of an interval is implicitly defined by a collection of points $x[0], \dots, x[N-1]$ (a grid) which are chosen to lie in the center of the subintervals. The i -th subinterval is thus given by:

$$I[i] = [(x[i-1]+x[i])/2, (x[i]+x[i+1])/2]$$

Attributes

<i>begin</i>	Minimum coordinates of the partitioned set.
<i>boundary_cell_fractions</i>	Return a tuple of contained fractions of boundary cells.
<i>cell_boundary_vecs</i>	Return the cell boundaries as coordinate vectors.
<i>cell_sides</i>	Side lengths of all ‘inner’ cells of a regular partition.
<i>cell_sizes_vecs</i>	Return the cell sizes as coordinate vectors.
<i>cell_volume</i>	Volume of the ‘inner’ cells, regardless of begin and end.
<i>end</i>	Maximum coordinates of the partitioned set.
<i>grid</i>	The <i>TensorGrid</i> defining this partition.
<i>is_regular</i>	Return True if <code>self.grid</code> is a <i>RegularGrid</i> .
<i>meshgrid</i>	Return the sparse meshgrid of sampling points.
<i>ndim</i>	Number of dimensions.
<i>set</i>	The partitioned set, an <i>IntervalProd</i> .
<i>shape</i>	Number of cells per axis, equal to <code>self.grid.shape</code> .
<i>size</i>	Total number of cells, equal to <code>self.grid.size</code> .

RectPartition.begin

`RectPartition.begin`

Minimum coordinates of the partitioned set.

RectPartition.boundary_cell_fractions

`RectPartition.boundary_cell_fractions`

Return a tuple of contained fractions of boundary cells.

Since the outermost grid points can have any distance to the boundary of the partitioned set, the “natural” outermost cell around these points can either be cropped or extended. This property is a tuple of (float, float) tuples, one entry per dimension, where the fractions of the left- and rightmost cells inside the set are stored. If a grid point lies exactly on the boundary, the value is 1/2 since the cell is cut in half. Otherwise, any value larger than 1/2 is possible.

Returnson_bdry : tuple of 2-tuple of float

Each 2-tuple contains the fraction of the leftmost (first entry) and rightmost (second entry) cell in the partitioned set in the corresponding dimension.

See also:

cell_boundary_vecs

Examples

We create a partition of the rectangle $[0, 1.5] \times [-2, 2]$ with the grid points $[0, 1] \times [-1, 0, 2]$. The “natural” cells at the boundary would be:

$[-0.5, 0.5]$ and $[0.5, 1.5]$ in the first axis

$[-1.5, -0.5]$ and $[1, 3]$ in the second axis

Thus, in the first axis, the fractions contained in $[0, 1.5]$ are 0.5 and 1, and in the second axis, $[-2, 2]$ contains the fractions 1.5 and 0.5.

```
>>> rect = IntervalProd([0, -2], [1.5, 2])
>>> grid = TensorGrid([0, 1], [-1, 0, 2])
>>> part = RectPartition(rect, grid)
>>> part.boundary_cell_fractions
((0.5, 1.0), (1.5, 0.5))
```

RectPartition.cell_boundary_vecs

RectPartition.cell_boundary_vecs

Return the cell boundaries as coordinate vectors.

Examples

```
>>> rect = IntervalProd([0, -1], [1, 2])
>>> grid = TensorGrid([0, 1], [-1, 0, 2])
>>> part = RectPartition(rect, grid)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.5,  1. ]), array([-1. , -0.5,  1. ,  2. ]))
```

RectPartition.cell_sides

RectPartition.cell_sides

Side lengths of all ‘inner’ cells of a regular partition.

Only defined if `self.grid` is a *RegularGrid*.

Examples

We create a partition of the rectangle $[0, 1] \times [-1, 2]$ into 3×3 cells, where the grid points lie on the boundary. This means that the grid points are $[0, 0.5, 1] \times [-1, 0.5, 2]$, i.e. the inner cell has side lengths 0.5×1.5 :

```
>>> rect = IntervalProd([0, -1], [1, 2])
>>> grid = RegularGrid([0, -1], [1, 2], (3, 3))
>>> part = RectPartition(rect, grid)
>>> part.cell_sides
array([ 0.5,  1.5])
```

RectPartition.cell_sizes_vecs

RectPartition.cell_sizes_vecs

Return the cell sizes as coordinate vectors.

Returns: `sizes`: tuple of `numpy.ndarray`

The cell sizes per axis. The length of the vectors is the same as the corresponding `grid.coord_vectors`. For axes with 1 grid point, cell size is set to 0.0.

Examples

We create a partition of the rectangle $[0, 1] \times [-1, 2]$ into 2×3 cells with the grid points $[0, 1] \times [-1, 0, 2]$. This implies that the cell boundaries are given as $[0, 0.5, 1] \times [-1, -0.5, 1, 2]$, hence the cell size vectors are $[0.5, 0.5] \times [0.5, 1.5, 1]$:

```
>>> rect = IntervalProd([0, -1], [1, 2])
>>> grid = TensorGrid([0, 1], [-1, 0, 2])
>>> part = RectPartition(rect, grid)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.5,  1. ]), array([-1. , -0.5,  1. ,  2. ]))
>>> part.cell_sizes_vecs
(array([ 0.5,  0.5]), array([ 0.5,  1.5,  1. ]))
```

RectPartition.cell_volume

`RectPartition.cell_volume`

Volume of the ‘inner’ cells, regardless of begin and end.

Only defined if `self.grid` is a *RegularGrid*.

Examples

We create a partition of the rectangle $[0, 1] \times [-1, 2]$ into 3×3 cells, where the grid points lie on the boundary. This means that the grid points are $[0, 0.5, 1] \times [-1, 0.5, 2]$, i.e. the inner cell has side lengths 0.5×1.5 :

```
>>> rect = IntervalProd([0, -1], [1, 2])
>>> grid = RegularGrid([0, -1], [1, 2], (3, 3))
>>> part = RectPartition(rect, grid)
>>> part.cell_sides
array([ 0.5,  1.5])
>>> part.cell_volume
0.75
```

RectPartition.end

`RectPartition.end`

Maximum coordinates of the partitioned set.

RectPartition.grid

`RectPartition.grid`

The *TensorGrid* defining this partition.

RectPartition.is_regular

`RectPartition.is_regular`

Return True if `self.grid` is a *RegularGrid*.

RectPartition.meshgrid

`RectPartition.meshgrid`

Return the sparse meshgrid of sampling points.

RectPartition.ndim

`RectPartition.ndim`
 Number of dimensions.

RectPartition.set

`RectPartition.set`
 The partitioned set, an *IntervalProd*.

RectPartition.shape

`RectPartition.shape`
 Number of cells per axis, equal to `self.grid.shape`.

RectPartition.size

`RectPartition.size`
 Total number of cells, equal to `self.grid.size`.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>approx_equals(other, atol)</code>	Return True in case of approximate equality.
<code>extent()</code>	Return a vector containing the total extent (max - min).
<code>insert(index, other)</code>	Return a copy with <code>other</code> inserted before <code>index</code> .
<code>max()</code>	Return the maximum point of the partitioned set.
<code>min()</code>	Return the minimum point of the partitioned set.
<code>points()</code>	Return the sampling grid points.

RectPartition.__eq__

`RectPartition.__eq__(other)`
 Return `self == other`.

RectPartition.approx_equals

`RectPartition.approx_equals(other, atol)`
 Return True in case of approximate equality.

Returns`approx_eq` : bool

True if `other` is a *RectPartition* instance with `self.set == other.set` up to `atol` and `self.grid == other.grid` up to `atol`.

RectPartition.extent

`RectPartition.extent()`
 Return a vector containing the total extent (max - min).

RectPartition.insert

`RectPartition.insert(index, other)`
 Return a copy with `other` inserted before `index`.

Parameters`index` : int

Index of the dimension before which `other` is to be inserted. Negative indices count backwards from `self.ndim`.

other : *RectPartition*

Partition to be inserted

Returns`newpart` : *RectPartition*

Partition with the inserted `other` partition

RectPartition.max

`RectPartition.max()`

Return the maximum point of the partitioned set.

See also:

odl.set.domain.IntervalProd.max

RectPartition.min

`RectPartition.min()`

Return the minimum point of the partitioned set.

See also:

odl.set.domain.IntervalProd.min

RectPartition.points

`RectPartition.points()`

Return the sampling grid points.

__init__ (*intv_prod*, *grid*)

Initialize a new instance.

Parameters`intv_prod` : *IntervalProd*

Set to be partitioned

grid : *TensorGrid*

Spatial points supporting the partition. They must be contained in `intv_prod`.

Functions

<i>uniform_partition</i> (begin, end, num_nodes[, ...])	Return a partition of [begin, end] with equally sized cells.
<i>uniform_partition_fromgrid</i> (grid[, begin, end])	Return a partition of an interval product based on a given grid.
<i>uniform_partition_fromintv</i> (intv_prod, num_nodes)	Return a partition of an interval product into equally sized cells.

uniform_partition

`odl.discr.partition.uniform_partition(begin, end, num_nodes, nodes_on_bdry=False)`

Return a partition of [begin, end] with equally sized cells.

Parameters`begin, end` : *array-like*

Vectors defining the begin end end points of an *IntervalProd* (a rectangular box)

num_nodes : int or sequence of int

Number of nodes per axis. For 1d intervals, a single integer can be specified.

nodes_on_bdry : bool or sequence, optional

If a sequence is provided, it determines per axis whether to place the last grid point on the boundary (True) or shift it by half a cell size into the interior (False). In each axis, an entry may consist in a single bool or a 2-tuple of bool. In the latter case, the first tuple entry decides for the left, the second for the right boundary. The length of the sequence must be `array.ndim`.

A single boolean is interpreted as a global choice for all boundaries.

See also:

[`uniform_partition_fromintv`](#) partition an existing set

[`uniform_partition_fromgrid`](#) use an existing grid as basis

Examples

By default, no grid points are placed on the boundary:

```
>>> part = uniform_partition(0, 1, 4)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.25,  0.5 ,  0.75,  1. ]),)
>>> part.grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]),)
```

This can be changed with the `nodes_on_bdry` parameter:

```
>>> part = uniform_partition(0, 1, 3, nodes_on_bdry=True)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.25,  0.75,  1. ]),)
>>> part.grid.coord_vectors
(array([ 0. ,  0.5,  1. ]),)
```

We can specify this per axis, too. In this case we choose both in the first axis and only the rightmost in the second:

```
>>> part = uniform_partition([0, 0], [1, 1], (3, 3),
...                           nodes_on_bdry=(True, (False, True)))
...
>>> part.cell_boundary_vecs[0] # first axis, as above
array([ 0. ,  0.25,  0.75,  1. ])
>>> part.grid.coord_vectors[0]
array([ 0. ,  0.5,  1. ])
>>> part.cell_boundary_vecs[1] # second, asymmetric axis
array([ 0. ,  0.4,  0.8,  1. ])
>>> part.grid.coord_vectors[1]
array([ 0.2,  0.6,  1. ])
```

`uniform_partition_fromgrid`

`odl.discr.partition.uniform_partition_fromgrid(grid, begin=None, end=None)`

Return a partition of an interval product based on a given grid.

This method is complementary to [`uniform_partition_fromintv`](#) in that it infers the set to be partitioned from a given grid and optional parameters for the begin and the end of the set.

Parametersgrid : *TensorGrid*

Grid on which the partition is based

begin, end : *array-like* or dictionary

Spatial points defining the begin and end of an interval product to be partitioned. The points can be specified in two ways:

array-like: These values are used directly as begin and/or end.

dictionary: Index-value pairs specifying an axis and a spatial coordinate to be used in that axis. In axes which are not a key in the dictionary, the coordinate for the vector is calculated as:

```
begin = x[0] - (x[1] - x[0]) / 2
end = x[-1] + (x[-1] - x[-2]) / 2
```

See Examples below.

In general, begin may not be larger than `grid.min_pt`, and end not smaller than `grid.max_pt` in any component. None is equivalent to an empty dictionary, i.e. the values are calculated in each dimension.

See also:

uniform_partition_fromintv

Examples

Have begin and end of the bounding box automatically calculated:

```
>>> grid = RegularGrid(0, 1, 3)
>>> grid.coord_vectors
(array([ 0. ,  0.5,  1. ]),)
>>> part = uniform_partition_fromgrid(grid)
>>> part.cell_boundary_vecs
(array([-0.25,  0.25,  0.75,  1.25]),)
```

Begin and end can be given explicitly as array-like:

```
>>> part = uniform_partition_fromgrid(grid, begin=0, end=1)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.25,  0.75,  1. ]),)
```

Using dictionaries, selective axes can be explicitly set. The keys refer to axes, the values to the coordinates to use:

```
>>> grid = RegularGrid([0, 0], [1, 1], (3, 3))
>>> part = uniform_partition_fromgrid(grid, begin={0: -1}, end={-1: 3})
>>> part.cell_boundary_vecs[0]
array([-1. ,  0.25,  0.75,  1.25])
>>> part.cell_boundary_vecs[1]
array([-0.25,  0.25,  0.75,  3.  ])
```

uniform_partition_fromintv

```
odl.discr.partition.uniform_partition_fromintv(intv_prod, num_nodes,
                                              nodes_on_bdry=False)
```

Return a partition of an interval product into equally sized cells.

Parameters
intv_prod : *IntervalProd*

Interval product to be partitioned

num_nodes : int or sequence of int

Number of nodes per axis. For 1d intervals, a single integer can be specified.

nodes_on_bdry : bool or sequence, optional

If a sequence is provided, it determines per axis whether to place the last grid point on the boundary (True) or shift it by half a cell size into the interior (False). In each axis, an entry may consist in a single bool or a 2-tuple of bool. In the latter case, the first tuple entry decides for the left, the second for the right boundary. The length of the sequence must be `array.ndim`.

A single boolean is interpreted as a global choice for all boundaries.

See also:

uniform_partition_fromgrid

Examples

By default, no grid points are placed on the boundary:

```
>>> interval = IntervalProd(0, 1)
>>> part = uniform_partition_fromintv(interval, 4)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.25,  0.5 ,  0.75,  1. ]),)
>>> part.grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]),)
```

This can be changed with the `nodes_on_bdry` parameter:

```
>>> part = uniform_partition_fromintv(interval, 3, nodes_on_bdry=True)
>>> part.cell_boundary_vecs
(array([ 0. ,  0.25,  0.75,  1. ]),)
>>> part.grid.coord_vectors
(array([ 0. ,  0.5,  1. ]),)
```

We can specify this per axis, too. In this case we choose both in the first axis and only the rightmost in the second:

```
>>> rect = IntervalProd([0, 0], [1, 1])
>>> part = uniform_partition_fromintv(
...     rect, (3, 3), nodes_on_bdry=(True, (False, True)))
...
>>> part.cell_boundary_vecs[0] # first axis, as above
array([ 0. ,  0.25,  0.75,  1. ])
>>> part.grid.coord_vectors[0]
array([ 0. ,  0.5,  1. ])
>>> part.cell_boundary_vecs[1] # second, asymmetric axis
array([ 0. ,  0.4,  0.8,  1. ])
```

```
>>> part.grid.coord_vectors[1]
array([ 0.2,  0.6,  1. ])
```

2.8.3 operator

Mathematical operators in ODL.

Modules

default_ops

Default operators defined on any (reasonable) space.

Classes

<i>ConstantOperator</i> (vector[, dom])	Operator that always returns the same value
<i>IdentityOperator</i> (space)	Operator mapping each element to itself.
<i>InnerProductOperator</i> (vector)	Operator taking the inner product with a fixed vector.
<i>LinCombOperator</i> (space, a, b)	Operator mapping two space elements to a linear combination.
<i>MultiplyOperator</i> (y[, domain])	Operator multiplying two elements.
<i>ResidualOperator</i> (op, vec)	Operator that calculates the residual $\text{op}(x) - \text{vec}$.
<i>ScalingOperator</i> (space, scalar)	Operator of multiplication with a scalar.
<i>ZeroOperator</i> (space)	Operator mapping each element to the zero element.

ConstantOperator

class odl.operator.default_ops.**ConstantOperator**(vector, dom=None)

Bases: odl.operator.operator.Operator

Operator that always returns the same value

ConstantOperator(vector)(x) <==> vector

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ConstantOperator.adjoint

ConstantOperator.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

ConstantOperator.domain

`ConstantOperator.domain`

Set of objects on which this operator can be evaluated.

ConstantOperator.inverse

`ConstantOperator.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ConstantOperator.is_functional

`ConstantOperator.is_functional`

True if the this operator's range is a *Field*.

ConstantOperator.is_linear

`ConstantOperator.is_linear`

True if this operator is linear.

ConstantOperator.range

`ConstantOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>__call__(x[, out])</code>	Return the constant vector or assign it to <code>out</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ConstantOperator.__call__

`ConstantOperator.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `__call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `__call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ConstantOperator.__call

ConstantOperator.__call(*x*, *out=None*)

Return the constant vector or assign it to `out`.

Parameters`x` : domain *element*

An element of the domain

out : range element

Vector that gets assigned to the constant vector

Returns`out` : range element

Result of the assignment. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = ConstantOperator(x)
>>> op(x, out=r3.element())
Rn(3).element([1.0, 2.0, 3.0])
```

ConstantOperator.derivativeConstantOperator.**derivative** (*point*)Return the operator derivative at *point*.**RaisesOpNotImplementedError**

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*vector*, *dom=None*)

Initialize an instance.

Parameters*vector* : *LinearSpaceVector*

The vector constant to be returned

dom : *LinearSpace*, default

The domain of the operator.

IdentityOperator**class** odl.operator.default_ops.**IdentityOperator** (*space*)Bases: odl.operator.default_ops.*ScalingOperator*

Operator mapping each element to itself.

Attributes

<i>adjoint</i>	Adjoint, given as scaling with the conjugate of the scalar.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the inverse operator.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

IdentityOperator.adjointIdentityOperator.**adjoint**

Adjoint, given as scaling with the conjugate of the scalar.

IdentityOperator.domainIdentityOperator.**domain**

Set of objects on which this operator can be evaluated.

IdentityOperator.inverseIdentityOperator.**inverse**

Return the inverse operator.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
```

```
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

IdentityOperator.is_functional**IdentityOperator.is_functional**True if the this operator's range is a *Field*.**IdentityOperator.is_linear****IdentityOperator.is_linear**

True if this operator is linear.

IdentityOperator.range**IdentityOperator.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Scale input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

IdentityOperator.__call__**IdentityOperator.__call__** (*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns***out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:`_call`Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

IdentityOperator._call

IdentityOperator._call(*x*, *out=None*)

Scale input and write to output.

Parameters*x*: domain *element*

input vector to be scaled

out: range *element*, optional

Output vector to which the result is written

Returns*out*: range *element*

Result of the scaling. If *out* was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out) # In place, Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
>>> op(vec) # Out of place
Rn(3).element([2.0, 4.0, 6.0])
```

IdentityOperator.derivative

IdentityOperator.derivative(*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__(space)`

Initialize an IdentityOperator instance.

Parameters`space` : LinearSpace

The space of elements which the operator is acting on

InnerProductOperator

class `odl.operator.default_ops.InnerProductOperator` (*vector*)

Bases: `odl.operator.operator.Operator`

Operator taking the inner product with a fixed vector.

`InnerProductOperator(vec)(x) <==> x.inner(vec)`

This is only applicable in inner product spaces.

Attributes

<code>T</code>	The vector of this operator.
<code>adjoint</code>	The adjoint operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

InnerProductOperator.T

`InnerProductOperator.T`

The vector of this operator.

Returns`vector` : *LinearSpaceVector*

Vector used in this operator

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> x.T
InnerProductOperator(Rn(3).element([1.0, 2.0, 3.0]))
>>> x.T.T
Rn(3).element([1.0, 2.0, 3.0])
```

InnerProductOperator.adjoint

`InnerProductOperator.adjoint`

The adjoint operator.

Returns`adjoint` : *MultiplyOperator*

The operator of multiplication with vector.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = InnerProductOperator(x)
>>> op.adjoint(2.0)
Rn(3).element([2.0, 4.0, 6.0])
```

InnerProductOperator.domain

InnerProductOperator.**domain**

Set of objects on which this operator can be evaluated.

InnerProductOperator.inverse

InnerProductOperator.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

InnerProductOperator.is_functional

InnerProductOperator.**is_functional**

True if the this operator's range is a *Field*.

InnerProductOperator.is_linear

InnerProductOperator.**is_linear**

True if this operator is linear.

InnerProductOperator.range

InnerProductOperator.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x)</code>	Multiply the input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

InnerProductOperator.__call__

InnerProductOperator.**__call__**(*x*, *out=None*, ***kwargs*)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters*x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in *_call*

Returns *out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:

_call Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

InnerProductOperator._call

InnerProductOperator._call(x)

Multiply the input and write to output.

Parameters *x* : vector.space *element*

An element in the space of the vector

Returns *out* : field *element*

Result of the inner product calculation

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = InnerProductOperator(x)
>>> op(r3.element([1, 2, 3]))
14.0
```

InnerProductOperator.derivative

`InnerProductOperator.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*vector*)

Initialize a `InnerProductOperator` instance.

Parameters*vector* : *LinearSpaceVector*

The vector to take the inner product with

LinCombOperator

`class odl.operator.default_ops.LinCombOperator` (*space, a, b*)

Bases: `odl.operator.operator.Operator`

Operator mapping two space elements to a linear combination.

This operator calculates:

`out = a*x[0] + b*x[1]`

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

LinCombOperator.adjoint

`LinCombOperator.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

LinCombOperator.domain

`LinCombOperator.domain`

Set of objects on which this operator can be evaluated.

LinCombOperator.inverse

`LinCombOperator.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

LinCombOperator.is_functional**LinCombOperator.is_functional**True if the this operator's range is a *Field*.**LinCombOperator.is_linear****LinCombOperator.is_linear**

True if this operator is linear.

LinCombOperator.range**LinCombOperator.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Linearly combine the input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

LinCombOperator.__call__**LinCombOperator.__call__** (*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns***out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

LinCombOperator._call

`LinCombOperator._call(x, out=None)`

Linearly combine the input and write to output.

Parameters`rx`: domain *element*

An element of the operator domain (2-tuple of space elements) whose linear combination is calculated

out: `'range element`

Vector to which the result is written

Returns`out`: *range element*

Result of the linear combination. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn, ProductSpace
>>> r3 = Rn(3)
>>> r3xr3 = ProductSpace(r3, r3)
>>> xy = r3xr3.element([[1, 2, 3], [1, 2, 3]])
>>> z = r3.element()
>>> op = LinCombOperator(r3, 1.0, 1.0)
>>> op(xy, out=z) # Returns z
Rn(3).element([2.0, 4.0, 6.0])
>>> z
Rn(3).element([2.0, 4.0, 6.0])
```

LinCombOperator.derivative

`LinCombOperator.derivative(point)`

Return the operator derivative at `point`.

Raises`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

__init__(*space, a, b*)

Initialize a `LinCombOperator` instance.

Parameters`space`: *LinearSpace*

The space of elements which the operator is acting on

a, b: scalar

Scalars to multiply $x[0]$ and $x[1]$ with, respectively

MultiplyOperator

class odl.operator.default_ops.**MultiplyOperator**(y, domain=None)

Bases: *odl.operator.operator.Operator*

Operator multiplying two elements.

`MultiplyOperator(y)(x) <==> x * y`

Here, x is a *LinearSpaceVector* or *Field* element and y is a *LinearSpaceVector*. Hence, this operator can be defined either on a *LinearSpace* or on a *Field*. In the first case it is the pointwise multiplication, in the second the scalar multiplication.

Attributes

<i>adjoint</i>	The adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

MultiplyOperator.adjoint

`MultiplyOperator.adjoint`

The adjoint operator.

Returns`adjoint`: {*InnerProductOperator*, *MultiplyOperator*}

If the domain of this operator is the scalar field of a *LinearSpace* the adjoint is the inner product with y , else it is the multiplication with y

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
```

Multiply by vector

```
>>> op = MultiplyOperator(x)
>>> out = r3.element()
>>> op.adjoint(x)
Rn(3).element([1.0, 4.0, 9.0])
```

Multiply by scalar

```
>>> op2 = MultiplyOperator(x, domain=r3.field)
>>> op2.adjoint(x)
14.0
```

MultiplyOperator.domain

`MultiplyOperator.domain`

Set of objects on which this operator can be evaluated.

`MultiplyOperator.inverse`

`MultiplyOperator.inverse`

Return the operator inverse.

`RaisesOpNotImplementedError`

Since the inverse cannot be default implemented.

`MultiplyOperator.is_functional`

`MultiplyOperator.is_functional`

True if the this operator's range is a *Field*.

`MultiplyOperator.is_linear`

`MultiplyOperator.is_linear`

True if this operator is linear.

`MultiplyOperator.range`

`MultiplyOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Multiply the input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

`MultiplyOperator.__call__`

`MultiplyOperator.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

[`__call`](#)Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

MultiplyOperator.`__call`

MultiplyOperator.`__call`(*x*, *out=None*)

Multiply the input and write to output.

Parameters*xs*: domain *element*

An element in the operator domain (2-tuple of space elements) whose elementwise product is calculated

out: range *element*, optional

Vector to which the result is written

Returns*out*: range *element*

Result of the multiplication. If *out* was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
```

Multiply by vector

```
>>> op = MultiplyOperator(x)
>>> out = r3.element()
>>> op(x, out)
Rn(3).element([1.0, 4.0, 9.0])
```

Multiply by scalar

```
>>> op2 = MultiplyOperator(x, domain=r3.field)
>>> out = r3.element()
>>> op2(3, out)
Rn(3).element([3.0, 6.0, 9.0])
```

MultiplyOperator.derivative

`MultiplyOperator.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*y*, *domain=None*)

Initialize a `MultiplyOperator` instance.

Parameters *y* : *LinearSpaceVector*

The value to multiply by

domain : *LinearSpace* or *Field*, optional

The set to take values in. Default: *x.space*

ResidualOperator

class `odl.operator.default_ops.ResidualOperator` (*op*, *vec*)

Bases: `odl.operator.operator.Operator`

Operator that calculates the residual $op(x) - vec$.

`ResidualOperator(op, vector)(x) <==> op(x) - vec`

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ResidualOperator.adjoint

`ResidualOperator.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

ResidualOperator.domain

`ResidualOperator.domain`

Set of objects on which this operator can be evaluated.

ResidualOperator.inverse`ResidualOperator.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ResidualOperator.is_functional`ResidualOperator.is_functional`True if the this operator's range is a *Field*.**ResidualOperator.is_linear**`ResidualOperator.is_linear`

True if this operator is linear.

ResidualOperator.range`ResidualOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Evaluate the residual at <code>x</code> .
<code>derivative(point)</code>	The derivative the residual operator.

ResidualOperator.__call__`ResidualOperator.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ResidualOperator.call

`ResidualOperator._call` (*x*, *out=None*)

Evaluate the residual at *x*.

Parameters*x*: domain *element*

Any element of the domain

out: range *element*

Vector that gets assigned to the constant vector

Returns*out*: range *element*

Result of the evaluation. If *out* was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = IdentityOperator(r3)
>>> res = ResidualOperator(op, vec)
>>> x = r3.element([4, 5, 6])
>>> res(x, out=r3.element())
Rn(3).element([3.0, 3.0, 3.0])
```

ResidualOperator.derivative

`ResidualOperator.derivative` (*point*)

The derivative the residual operator.

It is equal to the derivative of the “inner” operator:

`ResidualOperator(op, vec).derivative(x) <==> op.derivative(x)`

Parameters*x*: domain *element*

Any element in the domain where the derivative should be taken

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> op = IdentityOperator(r3)
>>> res = ResidualOperator(op, r3.element([1, 2, 3]))
>>> x = r3.element([4, 5, 6])
>>> res.derivative(x)(x)
Rn(3).element([4.0, 5.0, 6.0])
```

__init__(*op*, *vec*)
Initialize a new instance.

Parameters*sop* : *Operator*

Operator to be used in the residual expression. Its *Operator.range* must be a *LinearSpace*.

vec : *Operator.range element-like*

Vector to be subtracted from the operator result

ScalingOperator

class odl.operator.default_ops.**ScalingOperator**(*space*, *scalar*)

Bases: *odl.operator.operator.Operator*

Operator of multiplication with a scalar.

Attributes

<i>adjoint</i>	Adjoint, given as scaling with the conjugate of the scalar.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the inverse operator.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ScalingOperator.adjoint

ScalingOperator.**adjoint**

Adjoint, given as scaling with the conjugate of the scalar.

ScalingOperator.domain

ScalingOperator.**domain**

Set of objects on which this operator can be evaluated.

ScalingOperator.inverse

ScalingOperator.**inverse**

Return the inverse operator.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

ScalingOperator.is_functional

ScalingOperator.**is_functional**

True if the this operator's range is a *Field*.

ScalingOperator.is_linear

ScalingOperator.**is_linear**

True if this operator is linear.

ScalingOperator.range

ScalingOperator.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Scale input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ScalingOperator.__call__

ScalingOperator.**__call__**(*x*, *out=None*, ***kwargs*)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters*x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns*out* : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ScalingOperator.__call

`ScalingOperator.__call(x, out=None)`

Scale input and write to output.

Parameters`x`: domain *element*

input vector to be scaled

out: range *element*, optional

Output vector to which the result is written

Returns`out`: range *element*

Result of the scaling. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out) # In place, Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
>>> op(vec) # Out of place
Rn(3).element([2.0, 4.0, 6.0])
```


ScalingOperator.derivativeScalingOperator.**derivative** (*point*)Return the operator derivative at *point*.**RaisesOpNotImplementedError**

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*space*, *scalar*)

Initialize a ScalingOperator instance.

Parameters*space* : *LinearSpace*

The space of elements which the operator is acting on

scalar : *LinearSpace.field element*

An element of the field of the space which vectors are scaled with

ZeroOperator**class** odl.operator.default_ops.**ZeroOperator** (*space*)Bases: odl.operator.default_ops.*ScalingOperator*

Operator mapping each element to the zero element.

Attributes

<i>adjoint</i>	Adjoint, given as scaling with the conjugate of the scalar.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the inverse operator.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ZeroOperator.adjointZeroOperator.**adjoint**

Adjoint, given as scaling with the conjugate of the scalar.

ZeroOperator.domainZeroOperator.**domain**

Set of objects on which this operator can be evaluated.

ZeroOperator.inverseZeroOperator.**inverse**

Return the inverse operator.

Examples

```

>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)

```

```
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

ZeroOperator.is_functional**ZeroOperator.is_functional**True if the this operator's range is a *Field*.**ZeroOperator.is_linear****ZeroOperator.is_linear**

True if this operator is linear.

ZeroOperator.range**ZeroOperator.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Scale input and write to output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ZeroOperator.__call__**ZeroOperator.__call__**(*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns***out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ZeroOperator._call

ZeroOperator._call(*x*, *out=None*)

Scale input and write to output.

Parameters*xs*: domain *element*

input vector to be scaled

out: range *element*, optional

Output vector to which the result is written

Returns*out*: range *element*

Result of the scaling. If *out* was provided, the returned object is a reference to it.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out) # In place, Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
>>> op(vec) # Out of place
Rn(3).element([2.0, 4.0, 6.0])
```

ZeroOperator.derivative

ZeroOperator.derivative(*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__(space)`

Initialize a `ZeroOperator` instance.

Parameters`space` : *LinearSpace*

The space of elements which the operator is acting on

operator

Abstract mathematical operators.

Classes

<i>FunctionalLeftVectorMult</i> (op, vector)	Expression type for the functional left vector multiplication.
<i>OpDomainError</i>	Exception for domain errors.
<i>OpNotImplementedError</i>	Exception for not implemented errors in <i>LinearSpace</i> 's.
<i>OpRangeError</i>	Exception for domain errors.
<i>OpTypeError</i>	Exception for operator type errors.
<i>Operator</i> (domain, range[, linear])	Abstract mathematical operator.
<i>OperatorComp</i> (left, right[, tmp])	Expression type for the composition of operators.
<i>OperatorLeftScalarMult</i> (op, scalar)	Expression type for the operator left scalar multiplication.
<i>OperatorLeftVectorMult</i> (op, vector)	Expression type for the operator left vector multiplication.
<i>OperatorPointwiseProduct</i> (op1, op2)	Expression type for the pointwise operator multiplication.
<i>OperatorRightScalarMult</i> (op, scalar[, tmp])	Expression type for the operator right scalar multiplication.
<i>OperatorRightVectorMult</i> (op, vector)	Expression type for the operator right vector multiplication.
<i>OperatorSum</i> (op1, op2[, tmp_ran, tmp_dom])	Expression type for the sum of operators.

FunctionalLeftVectorMult

class odl.operator.operator.**FunctionalLeftVectorMult** (op, vector)

Bases: *odl.operator.operator.Operator*

Expression type for the functional left vector multiplication.

A functional is a *Operator* whose *Operator.range* is a *Field*. It is multiplied from left with a vector, resulting in an operator mapping from the *Operator.domain* to the vector's *LinearSpaceVector.space*.

`FunctionalLeftVectorMult(op, vector)(x) <==> vector * op(x)`

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

FunctionalLeftVectorMult.adjoint

`FunctionalLeftVectorMult.adjoint`

The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

```
FunctionalLeftVectorMult(op, vector).adjoint == OperatorComp(op.adjoint,
vector.T)
```

$$(x * A)^T = A^T * x^T$$

RaisesOpNotImplementedError

If the underlying operator is non-linear.

FunctionalLeftVectorMult.domain

`FunctionalLeftVectorMult.domain`

Set of objects on which this operator can be evaluated.

FunctionalLeftVectorMult.inverse

`FunctionalLeftVectorMult.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionalLeftVectorMult.is_functional

`FunctionalLeftVectorMult.is_functional`

True if the this operator's range is a *Field*.

FunctionalLeftVectorMult.is_linear

`FunctionalLeftVectorMult.is_linear`

True if this operator is linear.

FunctionalLeftVectorMult.range

`FunctionalLeftVectorMult.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the derivative at x.

FunctionalLeftVectorMult.__call__

`FunctionalLeftVectorMult.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain* element-like

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

FunctionalLeftVectorMult._call

FunctionalLeftVectorMult._call(x, out=None)

Implement self(x[, out]).

FunctionalLeftVectorMult.derivative

FunctionalLeftVectorMult.derivative(x)

Return the derivative at x.

Left scalar multiplication and derivative are commutative:

```
FunctionalLeftVectorMult(op, vector).derivative(x) <==>
FunctionalLeftVectorMult(op.derivative(x), vector)
```

See also:

FunctionalLeftVectorMult the result

`__init__(op, vector)`
Initialize a new instance.

Parameters`op : Operator`

The range of `op` must be a *Field*.

vector : LinearSpaceVector

The vector to multiply by. Its space's *LinearSpace.field* must be the same as `op.range`

OpDomainError

exception `odl.operator.operator.OpDomainError`

Exception for domain errors.

Domain errors are raised by *Operator* subclasses when trying to call them with input not in the domain (*Operator.domain*).

OpNotImplementedError

exception `odl.operator.operator.OpNotImplementedError`

Exception for not implemented errors in *LinearSpace*'s.

These are raised when a method in *LinearSpace* that has not been defined in a specific space is called.

OpRangeError

exception `odl.operator.operator.OpRangeError`

Exception for domain errors.

Domain errors are raised by *Operator* subclasses when the returned value does not lie in the range (*Operator.range*).

OpTypeError

exception `odl.operator.operator.OpTypeError`

Exception for operator type errors.

Domain errors are raised by *Operator* subclasses when trying to call them with input not in the domain (*Operator.domain*) or with the wrong range (*Operator.range*).

Operator

class `odl.operator.operator.Operator(domain, range, linear=False)`

Bases: `object`

Abstract mathematical operator.

An operator is a mapping

$$\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Y}$$

between sets \mathcal{X} (domain) and \mathcal{Y} (range). The evaluation of \mathcal{A} at an element $x \in \mathcal{X}$ is denoted by $\mathcal{A}(x)$ and produces an element in \mathcal{Y} :

$$y = \mathcal{A}(x) \in \mathcal{Y}.$$

Programmatically, these properties are reflected in the `Operator` class described in the following.

Abstract attributes and methods

`Operator` is an **abstract** class, i.e. it can only be subclassed, not used directly.

Any subclass of `Operator` must have the following attributes:

domain[`Set`] The set of elements this operator can be applied to

range[`Set`] The set this operator maps to

It is **highly** recommended to call `super().__init__(dom, ran)` (Note: add `from builtins import super` in Python 2) in the `__init__()` method of any subclass, where `dom` and `ran` are the arguments specifying domain and range of the new operator. In that case, the attributes `Operator.domain` and `Operator.range` are automatically provided by the parent class `Operator`.

In addition, any subclass **must** implement the private method `Operator._call()`. Its signature determines how it is interpreted:

In-place-only evaluation: `_call(self, x, out[, **kwargs])`

In-place evaluation means that the operator is applied, and the result is written to an existing element `out` provided, i.e.

```
_call(self, x, out) <==> out <-- operator(x)
```

Parameters:

x[`Operator.domain element`] An object in the operator domain to which the operator is applied

out[`Operator.range element`] An object in the operator range to which the result of the operator evaluation is written.

Returns:

None (return value is ignored)

Out-of-place-only evaluation: `_call(self, x[, **kwargs])`

Out-of-place evaluation means that the operator is applied, and the result is written to a **new** element which is returned. In this case, a subclass has to implement the method

```
_call(self, x) <==> operator(x)
```

Parameters:

x[`Operator.domain element`] An object in the operator domain to which the operator is applied

Returns:

out[`Operator.range element-like`] An object in the operator range holding the result of the operator evaluation

Dual-use evaluation: `_call(self, x, out=None[, **kwargs])`

Evaluate in place if `out` is given, otherwise out of place.

Parameters:

x[`Operator.domain element`] An object in the operator domain to which the operator is applied

out[*Operator.range element*, optional] An object in the operator range to which the result of the operator evaluation is written

Returns:

None (return value is ignored)

Notes

- If *Operator._call* is implemented in-place-only or out-of-place-only and the *Operator.range* is a *LinearSpace*, a default implementation of the respective other is provided.
- Operator._call* is allowed to have keyword-only arguments (Python 3 only).
- The term “element-like” means that an object must be convertible to an element by the `domain.element()` method.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

Operator.adjoint

Operator.adjoint

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

Operator.domain

Operator.domain

Set of objects on which this operator can be evaluated.

Operator.inverse

Operator.inverse

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

Operator.is_functional

Operator.is_functional

True if the this operator’s range is a *Field*.

Operator.is_linear

`Operator.is_linear`

True if this operator is linear.

Operator.range

`Operator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implementation of the operator evaluation.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

Operator.__call__

`Operator.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

Operator._call

`Operator._call(x, out=None, **kwargs)`

Implementation of the operator evaluation.

This method is private backend for the evaluation of this operator. It needs to match certain signature conventions, and its implementation type is inferred from its signature.

The following signatures are allowed:

Python 2 and 3:

- `_call(self, x)` -> out-of-place evaluation
- `_call(self, vec, out)` -> in-place evaluation
- `_call(self, x, out=None)` -> both

Python 3 only:

- `_call(self, x, *, out=None)` (out as keyword-only argument) -> both

For disambiguation, the instance name (the first argument) **must** be 'self'.

The name of the `out` argument **must** be 'out', the second argument may have any name.

Additional variable `**kwargs` and keyword-only arguments (Python 3 only) are also allowed.

Parameters`x` : *Operator.domain element-like*

Element to which the operator is applied

out : *Operator.range element*, optional

Element to which the result is written

Returns`out` : *Operator.range element-like*

Result of the evaluation. If `out` was provided, the returned object is a reference to it.

Notes

The public call pattern `op()` using `op.__call__` provides a default implementation of the underlying in-place or out-of-place call even if you choose the respective other pattern.

Operator.derivative

`Operator.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*domain, range, linear=False*)

Initialize a new instance.

Parameters**domain** : *Set*

The domain of this operator, i.e., the set of elements to which this operator can be applied

range : *Set*

The range of this operator, i.e., the set this operator maps to

linear : bool

If True, the operator is considered as linear. In this case, domain and range have to be instances of *LinearSpace*, or *Field*.

OperatorComp

class odl.operator.operator.**OperatorComp**(left, right, tmp=None)

Bases: odl.operator.operator.Operator

Expression type for the composition of operators.

OperatorComp(left, right) <==> (x --> left(right(x)))

The composition is only well-defined if left.domain == right.range.

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	The operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorComp.adjoint

OperatorComp.**adjoint**

The operator adjoint.

The adjoint of the operator composition is the composition of the operator adjoints in reverse order:

OperatorComp(left, right).adjoint == OperatorComp(right.adjoint, left.adjoint)

RaisesOpNotImplementedError

If any of the underlying operators are non-linear.

OperatorComp.domain

OperatorComp.**domain**

Set of objects on which this operator can be evaluated.

OperatorComp.inverse

OperatorComp.**inverse**

The operator inverse.

The inverse of the operator composition is the composition of the inverses in reverse order:

```
OperatorComp(left, right).inverse == OperatorComp(right.inverse,
left.inverse)
```

OperatorComp.is_functional

OperatorComp.**is_functional**

True if the this operator's range is a *Field*.

OperatorComp.is_linear

OperatorComp.**is_linear**

True if this operator is linear.

OperatorComp.range

OperatorComp.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the operator derivative.

OperatorComp.__call__

OperatorComp.**__call__**(*x*, *out=None*, ***kwargs*)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters*x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns*out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorComp._call

`OperatorComp._call(x, out=None)`
Implement `self(x[, out])`.

OperatorComp.derivative

`OperatorComp.derivative(x)`
Return the operator derivative.

The derivative of the operator composition follows the chain rule:

```
OperatorComp(left, right).derivative(x) == OperatorComp(left.derivative(right(x)),
right.derivative(x))
```

Parameters`x` : *Operator.domain element-like*

Evaluation point of the derivative. Needs to be usable as input for the `right` operator.

__init__(*left, right, tmp=None*)

Initialize a new *OperatorComp* instance.

Parameters`left` : *Operator*

The left (“outer”) operator

right : *Operator*

The right (“inner”) operator. Its range must coincide with the domain of `left`.

tmp : *element* of the range of `right`, optional

Used to avoid the creation of a temporary when applying the operator.

OperatorLeftScalarMult

class `odl.operator.operator.OperatorLeftScalarMult(op, scalar)`

Bases: *odl.operator.operator.Operator*

Expression type for the operator left scalar multiplication.

```
OperatorLeftScalarMult(op, scalar) <==> (x --> scalar * op(x))
```

The scalar multiplication is well-defined only if `op.range` is a *LinearSpace*.

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	The inverse operator.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorLeftScalarMult.adjoint

`OperatorLeftScalarMult.adjoint`

The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

```
OperatorLeftScalarMult(op, scalar).adjoint == OperatorLeftScalarMult(op.adjoint,
scalar)
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorLeftScalarMult.domain

`OperatorLeftScalarMult.domain`

Set of objects on which this operator can be evaluated.

OperatorLeftScalarMult.inverse

`OperatorLeftScalarMult.inverse`

The inverse operator.

The inverse of `scalar * op` is given by `op.inverse * 1/scalar` if `scalar != 0`. If `scalar == 0`, the inverse is not defined.

```
OperatorLeftScalarMult(op, scalar).inverse <==> OperatorRightScalarMult(op.inverse,
1.0/scalar)
```

OperatorLeftScalarMult.is_functional

`OperatorLeftScalarMult.is_functional`

True if the this operator's range is a *Field*.

OperatorLeftScalarMult.is_linear

`OperatorLeftScalarMult.is_linear`

True if this operator is linear.

OperatorLeftScalarMult.range

`OperatorLeftScalarMult.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the derivative at <code>x</code> .

OperatorLeftScalarMult.__call__

`OperatorLeftScalarMult.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```


OperatorLeftScalarMult._call

`OperatorLeftScalarMult._call(x, out=None)`
 Implement `self(x[, out])`.

OperatorLeftScalarMult.derivative

`OperatorLeftScalarMult.derivative(x)`

Return the derivative at `x`.

Left scalar multiplication and derivative are commutative:

```
OperatorLeftScalarMult(op, scalar).derivative(x) <==>
OperatorLeftScalarMult(op.derivative(x), scalar)
```

Parameters`x` : *Operator.domain element-like*

Evaluation point of the derivative

See also:

OperatorLeftScalarMult the result

`__init__(op, scalar)`

Initialize a new *OperatorLeftScalarMult* instance.

Parameters`op` : *Operator*

The range of `op` must be a *LinearSpace* or *Field*.

scalar : `op.range.field element`

A real or complex number, depending on the field of the range.

OperatorLeftVectorMult

class `odl.operator.operator.OperatorLeftVectorMult(op, vector)`

Bases: *odl.operator.operator.Operator*

Expression type for the operator left vector multiplication.

```
OperatorLeftVectorMult(op, vector)(x) <==> vector * op(x)
```

The scalar multiplication is well-defined only if `op.range` is a `vector.space.field`.

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorLeftVectorMult.adjoint

`OperatorLeftVectorMult.adjoint`

The operator adjoint.

The adjoint of the operator vector multiplication is the vector multiplication of the operator adjoint:

```
OperatorLeftVectorMult(op, vector).adjoint == OperatorRightVectorMult(op.adjoint,
vector)
```

```
(x * A)^T = A^T * x
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorLeftVectorMult.domain

```
OperatorLeftVectorMult.domain
```

Set of objects on which this operator can be evaluated.

OperatorLeftVectorMult.inverse

```
OperatorLeftVectorMult.inverse
```

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorLeftVectorMult.is_functional

```
OperatorLeftVectorMult.is_functional
```

True if the this operator's range is a *Field*.

OperatorLeftVectorMult.is_linear

```
OperatorLeftVectorMult.is_linear
```

True if this operator is linear.

OperatorLeftVectorMult.range

```
OperatorLeftVectorMult.range
```

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the derivative at <code>x</code> .

OperatorLeftVectorMult.__call__

```
OperatorLeftVectorMult.__call__(x, out=None, **kwargs)
```

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorLeftVectorMult._call

`OperatorLeftVectorMult._call` (*x*, *out=None*)
Implement `self(x[, out])`.

OperatorLeftVectorMult.derivative

`OperatorLeftVectorMult.derivative` (*x*)
Return the derivative at *x*.

Left scalar multiplication and derivative are commutative:

```
OperatorLeftVectorMult(op, vector).derivative(x) <==>
OperatorLeftVectorMult(op.derivative(x), vector)
```

See also:

`OperatorLeftVectorMult` the result

`__init__` (*op*, *vector*)

Initialize a new `OperatorLeftVectorMult` instance.

Parameters *op* : `Operator`

The range of *op* must be a `LinearSpace`.

vector : *LinearSpaceVector* in `op.range`

The vector to multiply by

OperatorPointwiseProduct

class `odl.operator.operator.OperatorPointwiseProduct (op1, op2)`

Bases: `odl.operator.operator.Operator`

Expression type for the pointwise operator multiplication.

`OperatorPointwiseProduct (op1, op2) <==> (x --> op1 (x) * op2 (x))`

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorPointwiseProduct.adjoint

`OperatorPointwiseProduct.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

OperatorPointwiseProduct.domain

`OperatorPointwiseProduct.domain`

Set of objects on which this operator can be evaluated.

OperatorPointwiseProduct.inverse

`OperatorPointwiseProduct.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorPointwiseProduct.is_functional

`OperatorPointwiseProduct.is_functional`

True if the this operator's range is a *Field*.

OperatorPointwiseProduct.is_linear

`OperatorPointwiseProduct.is_linear`

True if this operator is linear.

OperatorPointwiseProduct.rangeOperatorPointwiseProduct.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

OperatorPointwiseProduct.__call__OperatorPointwiseProduct.**__call__**(*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns*out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorPointwiseProduct._call

OperatorPointwiseProduct.**_call**(*x*, *out=None*)
Implement self(*x*[, *out*]).

OperatorPointwiseProduct.derivative

OperatorPointwiseProduct.**derivative**(*point*)
Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__(*op1*, *op2*)
Initialize a new instance.

Parameters*op1* : *Operator*

The first factor

op2 : *Operator*

The second factor. Must have the same domain and range as *op1*.

OperatorRightScalarMult

class odl.operator.operator.**OperatorRightScalarMult**(*op*, *scalar*, *tmp=None*)

Bases: *odl.operator.operator.Operator*

Expression type for the operator right scalar multiplication.

OperatorRightScalarMult(*op*, *scalar*) <==> (*x* --> *op*(*scalar* * *x*))

The scalar multiplication is well-defined only if *op*.domain is a *LinearSpace*.

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	The inverse operator.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorRightScalarMult.adjoint

OperatorRightScalarMult.**adjoint**
The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

OperatorLeftScalarMult(*op*, *scalar*).adjoint == OperatorLeftScalarMult(*op*.adjoint,

scalar)

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorRightScalarMult.domain

`OperatorRightScalarMult.domain`

Set of objects on which this operator can be evaluated.

OperatorRightScalarMult.inverse

`OperatorRightScalarMult.inverse`

The inverse operator.

The inverse of `op * scalar` is given by `1/scalar * op.inverse` if `scalar != 0`. If `scalar == 0`, the inverse is not defined.

`OperatorRightScalarMult(op, scalar).inverse <==> OperatorLeftScalarMult(op.inverse, 1.0/scalar)`

OperatorRightScalarMult.is_functional

`OperatorRightScalarMult.is_functional`

True if the this operator's range is a *Field*.

OperatorRightScalarMult.is_linear

`OperatorRightScalarMult.is_linear`

True if this operator is linear.

OperatorRightScalarMult.range

`OperatorRightScalarMult.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the derivative at <code>x</code> .

OperatorRightScalarMult.__call__

`OperatorRightScalarMult.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorRightScalarMult._call

`OperatorRightScalarMult._call(x, out=None)`
Implement `self(x[, out])`.

OperatorRightScalarMult.derivative

`OperatorRightScalarMult.derivative(x)`
Return the derivative at `x`.

The derivative of the right scalar operator multiplication follows the chain rule:

```
OperatorRightScalarMult(op, scalar).derivative(x) <==>
OperatorLeftScalarMult(op.derivative(scalar * x), scalar)
```

Parameters `x` : `Operator.domain element-like`

Evaluation point of the derivative

`__init__(op, scalar, tmp=None)`

Initialize a new `OperatorLeftScalarMult` instance.

Parameters `op` : `Operator`

The domain of `op` must be a `LinearSpace` or `Field`.

scalar : `op.range.field` *element*

A real or complex number, depending on the field of the operator domain.

tmp : domain *element*, optional

Used to avoid the creation of a temporary when applying the operator.

OperatorRightVectorMult

class `odl.operator.operator.OperatorRightVectorMult` (*op*, *vector*)

Bases: `odl.operator.operator.Operator`

Expression type for the operator right vector multiplication.

`OperatorRightVectorMult(op, vector)(x) <==> op(vector * x)`

The scalar multiplication is well-defined only if `vector in op.domain == True`.

Attributes

<i>adjoint</i>	The operator adjoint.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

OperatorRightVectorMult.adjoint

`OperatorRightVectorMult.adjoint`

The operator adjoint.

The adjoint of the operator vector multiplication is the vector multiplication of the operator adjoint:

`OperatorRightVectorMult(op, vector).adjoint == OperatorLeftVectorMult(op.adjoint, vector)`

$(A \ x)^T = x \ * \ A^T$

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorRightVectorMult.domain

`OperatorRightVectorMult.domain`

Set of objects on which this operator can be evaluated.

OperatorRightVectorMult.inverse

`OperatorRightVectorMult.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorRightVectorMult.is_functional**OperatorRightVectorMult.is_functional**True if the this operator's range is a *Field*.**OperatorRightVectorMult.is_linear****OperatorRightVectorMult.is_linear**

True if this operator is linear.

OperatorRightVectorMult.range**OperatorRightVectorMult.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the derivative at <code>x</code> .

OperatorRightVectorMult.__call__**OperatorRightVectorMult.__call__** (`x`, `out=None`, `**kwargs`)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorRightVectorMult._call

OperatorRightVectorMult._call(x, out=None)
Implement self(x[, out]).

OperatorRightVectorMult.derivative

OperatorRightVectorMult.derivative(x)
Return the derivative at x.

Left vector multiplication and derivative are commutative:

```
OperatorRightVectorMult(op, vector).derivative(x) <==>
OperatorRightVectorMult(op.derivative(x), vector)
```

See also:

[OperatorRightVectorMult](#) the result

__init__(op, vector)
Initialize a new [OperatorRightVectorMult](#) instance.

Parametersop : [Operator](#)

The domain of op must be a vector.space.

vector : [LinearSpaceVector](#) in op.domain

The vector to multiply by

OperatorSum

class odl.operator.operator.OperatorSum(op1, op2, tmp_ran=None, tmp_dom=None)
Bases: [odl.operator.operator.Operator](#)

Expression type for the sum of operators.

```
OperatorSum(op1, op2) <==> (x --> op1(x) + op2(x))
```

The sum is only well-defined for [Operator](#) instances where [Operator.range](#) is a [LinearSpace](#).

Attributes

adjoint	The operator adjoint.
domain	Set of objects on which this operator can be evaluated.

Continued on next page

Table 2.87 – continued from previous page

<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

OperatorSum.adjoint`OperatorSum.adjoint`

The operator adjoint.

The adjoint of the operator sum is the sum of the operator adjoints:

`OperatorSum(op1, op2).adjoint == OperatorSum(op1.adjoint, op2.adjoint)`**RaisesOpNotImplementedError**

If either of the underlying operators are non-linear.

OperatorSum.domain`OperatorSum.domain`

Set of objects on which this operator can be evaluated.

OperatorSum.inverse`OperatorSum.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorSum.is_functional`OperatorSum.is_functional`True if the this operator's range is a *Field*.**OperatorSum.is_linear**`OperatorSum.is_linear`

True if this operator is linear.

OperatorSum.range`OperatorSum.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Implement <code>self(x[, out])</code> .
<code>derivative(x)</code>	Return the operator derivative at <code>x</code> .

OperatorSum.__call__

```
OperatorSum.__call__(x, out=None, **kwargs)
    Return self(x[, out, **kwargs]).
```

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorSum._call

```
OperatorSum._call(x, out=None)
    Implement self(x[, out]).
```

Examples

```
>>> from odl import Rn, IdentityOperator
>>> r3 = Rn(3)
>>> op = IdentityOperator(r3)
>>> x = r3.element([1, 2, 3])
>>> out = r3.element()
>>> OperatorSum(op, op)(x, out) # In place, returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
>>> OperatorSum(op, op)(x)
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorSum.derivative

OperatorSum.**derivative**(x)

Return the operator derivative at x.

The derivative of a sum of two operators is equal to the sum of the derivatives.

Parametersx : *Operator.domain element-like*

Evaluation point of the derivative

__init__(op1, op2, tmp_ran=None, tmp_dom=None)

Initialize a new instance.

Parametersop1 : *Operator*

The first summand. Its *Operator.range* must be a *LinearSpace* or *Field*.

op2 : *Operator*

The second summand. Must have the same *Operator.domain* and *Operator.range* as op1.

tmp_ran : *Operator.range element*, optional

Used to avoid the creation of a temporary when applying the operator.

tmp_dom : *Operator.domain element*, optional

Used to avoid the creation of a temporary when applying the operator adjoint.

Functions

simple_operator([call, inv, deriv, dom, ...]) Create a simple operator.

simple_operator

odl.operator.operator.**simple_operator**(call=None, inv=None, deriv=None, dom=None, ran=None, linear=False)

Create a simple operator.

Mostly intended for simple prototyping rather than final use.

Parameterscall : callable

Function with valid call signature, see *Operator*

inv : *Operator*, optional

The operator inverse

deriv : *Operator*, optional

The operator derivative, linear

dom : *Set*, optional

The domain of the operator Default: *UniversalSpace* if linear, else *UniversalSet*

ran : *Set*, optional

The range of the operator Default: *UniversalSpace* if linear, else *UniversalSet*

linear : bool, optional

True if the operator is linear Default: False

Returnsop : *Operator*

An operator with the provided attributes and methods.

Notes

It suffices to supply one of the functions `call` and `apply`. If `dom` is a *LinearSpace*, a default implementation of the respective other method is automatically provided; if not, a *OpNotImplementedError* is raised when the other method is called.

Examples

```
>>> A = simple_operator(lambda x: 3*x)
>>> A(5)
15
```

oputils

Usefull utility functions on discrete spaces (i.e., either $\mathbb{R}^n/\mathbb{C}^n$ or discretized function spaces), for example obtaining a matrix representation of an operator.

Functions

<code>matrix_representation(op)</code>	Returns a matrix representation of a linear operator.
<code>power_method_opnorm(op, niter[, xstart])</code>	Estimate the operator norm with the power method.

matrix_representation

`odl.operator.oputils.matrix_representation(op)`

Returns a matrix representation of a linear operator.

Parametersop : *Operator*

The linear operator of which one wants a matrix representation.

Returnsmatrix : `numpy.ndarray`

The matrix representation of the operator.

Notes

The algorithm works by letting the operator act on all unit vectors, and stacking the output as a matrix.

power_method_opnorm

`odl.operator.oputils.power_method_opnorm(op, niter, xstart=None)`

Estimate the operator norm with the power method.

Parameters
sop : *Operator*

Operator whose norm is to be estimated. If its *Operator.range* range does not coincide with its *Operator.domain*, an *Operator.adjoint* must be defined (which implies that the operator must be linear).

niter : positive int

Number of iterations to perform

xstart : *Operator.domain element*, optional

Starting point of the iteration. By default, the one element of the *Operator.domain* is used.

Return
est_norm : float

The estimated operator norm

pspace_ops

Default operators defined on any *ProductSpace*.

Classes

<i>BroadcastOperator</i> (*operators)	Broadcast argument to set of operators.
<i>ComponentProjection</i> (space, index)	Projection onto the subspace identified by an index.
<i>ComponentProjectionAdjoint</i> (space, index)	Adjoint operator to <i>ComponentProjection</i> .
<i>ProductSpaceOperator</i> (operators[, dom, ran])	A “matrix of operators” on product spaces.
<i>ReductionOperator</i> (*operators)	Reduce argument over set of operators.

BroadcastOperator

class `odl.operator.pspace_ops.BroadcastOperator(*operators)`

Bases: *odl.operator.operator.Operator*

Broadcast argument to set of operators.

An argument is broadcast by evaluating several operators in the same point

`BroadcastOperator(op1, op2)(x) = [op1(x), op2(x)]`

It is implemented using a *ProductSpaceOperator*.

Attributes

<code>adjoint</code>	Adjoint of the broadcast operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>operators</code>	A tuple of sub-operators
<code>prod_op</code>	The prod-op implementation
<code>range</code>	Set in which the result of an evaluation of this operator lies.

BroadcastOperator.adjoint

`BroadcastOperator.adjoint`

Adjoint of the broadcast operator.

Returns`adjoint` : linear *BroadcastOperator*

The adjoint

Examples

```
>>> import odl
>>> I = odl.IdentityOperator(odl.Rn(3))
>>> op = BroadcastOperator(I, 2 * I)
>>> op.adjoint([[1, 2, 3], [2, 3, 4]])
Rn(3).element([5.0, 8.0, 11.0])
```

BroadcastOperator.domain

`BroadcastOperator.domain`

Set of objects on which this operator can be evaluated.

BroadcastOperator.inverse

`BroadcastOperator.inverse`

Return the operator inverse.

Raises`OpNotImplementedError`

Since the inverse cannot be default implemented.

BroadcastOperator.is_functional

`BroadcastOperator.is_functional`

True if the this operator's range is a *Field*.

BroadcastOperator.is_linear

`BroadcastOperator.is_linear`

True if this operator is linear.

BroadcastOperator.operators

`BroadcastOperator.operators`

A tuple of sub-operators

BroadcastOperator.prod_op`BroadcastOperator.prod_op`

The prod-op implementation

BroadcastOperator.range`BroadcastOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Apply operators to <code>x</code> .
<code>derivative(x)</code>	Derivative of the broadcast operator.

BroadcastOperator.__call__`BroadcastOperator.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

BroadcastOperator._call

BroadcastOperator._call(x, out=None)

Apply operators to x.

Parametersx : domain element

Input vector to be evaluated by operators

out : range element, optional

output vector to write result to

Returnsout : range element

Values of operators evaluated in point

Examples

```
>>> import odl
>>> I = odl.IdentityOperator(odl.Rn(3))
>>> op = BroadcastOperator(I, 2 * I)
>>> x = [1, 2, 3]
>>> op(x)
ProductSpace(Rn(3), 2).element([
    [1.0, 2.0, 3.0],
    [2.0, 4.0, 6.0]
])
```

BroadcastOperator.derivative

BroadcastOperator.derivative(x)

Derivative of the broadcast operator.

Parametersx : domain element

The point to take the derivative in

Returnsadjoint : linear *BroadcastOperator*

The derivative

Examples

Example with affine operator

```
>>> import odl
>>> I = odl.IdentityOperator(odl.Rn(3))
>>> residual_op = odl.ResidualOperator(I, I.domain.element([1, 1, 1]))
>>> op = BroadcastOperator(residual_op, 2 * residual_op)
```

Calling operator gives offset by [1, 1, 1]

```
>>> x = [1, 2, 3]
>>> op(x)
ProductSpace(Rn(3), 2).element([
    [0.0, 1.0, 2.0],
    [0.0, 2.0, 4.0]
])
```

Derivative of affine operator does not have this offset

```
>>> op.derivative(x)(x)
ProductSpace(Rn(3), 2).element([
    [1.0, 2.0, 3.0],
    [2.0, 4.0, 6.0]
])
```

`__init__(*operators)`

ComponentProjection

class odl.operator.pspace_ops.**ComponentProjection**(space, index)

Bases: [odl.operator.operator.Operator](#)

Projection onto the subspace identified by an index.

For a product space $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$, the component projection

$$\mathcal{P}_i : \mathcal{X} \rightarrow \mathcal{X}_i$$

is given by $\mathcal{P}_i(x) = x_i$ for an element $x = (x_1, \dots, x_n) \in \mathcal{X}$.

More generally, for an index set $I \subset \{1, \dots, n\}$, the projection operator \mathcal{P}_I is defined by $\mathcal{P}_I(x) = (x_i)_{i \in I}$.

Note that this is a special case of a product space operator where the “operator matrix” has only one row and contains only identity operators.

Attributes

<i>adjoint</i>	Return the adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>index</i>	Index of the subspace.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ComponentProjection.adjoint

ComponentProjection.adjoint

Return the adjoint operator.

The adjoint is given by extending along [*ComponentProjection.index*](#), and setting zero along the others.

See also:

[*ComponentProjectionAdjoint*](#)

ComponentProjection.domain`ComponentProjection.domain`

Set of objects on which this operator can be evaluated.

ComponentProjection.index`ComponentProjection.index`

Index of the subspace.

ComponentProjection.inverse`ComponentProjection.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ComponentProjection.is_functional`ComponentProjection.is_functional`True if the this operator's range is a *Field*.**ComponentProjection.is_linear**`ComponentProjection.is_linear`

True if this operator is linear.

ComponentProjection.range`ComponentProjection.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Project <code>x</code> onto subspace.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ComponentProjection.__call__`ComponentProjection.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ComponentProjection._call

`ComponentProjection._call(x, out=None)`

Project `x` onto subspace.

Parameters`x` : domain `element`

input vector to be projected

out : range `element`, optional

output vector to write result to

Returns`out` : range `element`

Projection of `x` onto subspace. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
>>> x = X.element([[1], [2, 3], [4, 5, 6]])
```

Projection on n-th component

```
>>> proj = odl.ComponentProjection(X, 0)
>>> proj(x)
Rn(1).element([1.0])
```

Projection on sub-space

```
>>> proj = odl.ComponentProjection(X, [0, 2])
>>> proj(x)
ProductSpace(Rn(1), Rn(3)).element([
    [1.0],
    [4.0, 5.0, 6.0]
])
```

ComponentProjection.derivative

`ComponentProjection.derivative` (*point*)

Return the operator derivative at point.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*space, index*)

Parameters*space* : *ProductSpace*

The space to project from

index : int, slice, or iterable [int]

The indices defining the subspace. If index is not an int, the *Operator.range* of this operator is also a *ProductSpace*.

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
```

Projection on n-th component

```
>>> proj = odl.ComponentProjection(X, 0)
>>> proj.range
Rn(1)
```

Projection on sub-space

```
>>> proj = odl.ComponentProjection(X, [0, 2])
>>> proj.range
ProductSpace(Rn(1), Rn(3))
```

ComponentProjectionAdjoint

class `odl.operator.pspace_ops.ComponentProjectionAdjoint` (*space, index*)

Bases: *odl.operator.operator.Operator*

Adjoint operator to *ComponentProjection*.

As a special case of the adjoint of a *ProductSpaceOperator*, this operator is given as a column vector of identity operators and zero operators, with the identities placed in the positions defined by *ComponentProjectionAdjoint.index*.

In weighted product spaces, the adjoint needs to take the weightings into account. This is currently not supported.

Attributes

<i>adjoint</i>	The adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>index</i>	Index of the subspace.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ComponentProjectionAdjoint.adjoint

`ComponentProjectionAdjoint.adjoint`

The adjoint operator.

The adjoint is given by the *ComponentProjection* related to this operator's *index*.

See also:

ComponentProjection

ComponentProjectionAdjoint.domain

`ComponentProjectionAdjoint.domain`

Set of objects on which this operator can be evaluated.

ComponentProjectionAdjoint.index

`ComponentProjectionAdjoint.index`

Index of the subspace.

ComponentProjectionAdjoint.inverse

`ComponentProjectionAdjoint.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ComponentProjectionAdjoint.is_functional

`ComponentProjectionAdjoint.is_functional`

True if the this operator's range is a *Field*.

ComponentProjectionAdjoint.is_linear

`ComponentProjectionAdjoint.is_linear`

True if this operator is linear.

ComponentProjectionAdjoint.rangeComponentProjectionAdjoint.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Extend <code>x</code> from the subspace.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ComponentProjectionAdjoint.__call__ComponentProjectionAdjoint.**__call__**(`x`, `out=None`, `**kwargs`)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.**See also:**`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ComponentProjectionAdjoint._call

ComponentProjectionAdjoint._call(*x*, *out=None*)

Extend *x* from the subspace.

Parameters*x* : domain *element*

Input vector to be extended

out : range *element*, optional

output vector to write result to

Returns*out* : range *element*

Extension of *x* to superspace. If *out* was provided, the returned object is a reference to it.

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
>>> x = X.element([[1], [2, 3], [4, 5, 6]])
```

Projection on n-th component

```
>>> proj = odl.ComponentProjectionAdjoint(X, 0)
>>> proj(x[0])
ProductSpace(Rn(1), Rn(2), Rn(3)).element([
    [1.0],
    [0.0, 0.0],
    [0.0, 0.0, 0.0]
])
```

Projection on sub-space

```
>>> proj = odl.ComponentProjectionAdjoint(X, [0, 2])
>>> proj(x[0, 2])
ProductSpace(Rn(1), Rn(2), Rn(3)).element([
    [1.0],
    [0.0, 0.0],
    [4.0, 5.0, 6.0]
])
```

ComponentProjectionAdjoint.derivative

ComponentProjectionAdjoint.derivative(*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*space, index*)

Initialize a new instance

Parameters*space* : *ProductSpace*

The space to project to

index : int, slice, or iterable [int]

The indexes to project from

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
```

Projection on n-th component

```
>>> proj = odl.ComponentProjectionAdjoint(X, 0)
>>> proj.domain
Rn(1)
```

Projection on sub-space

```
>>> proj = odl.ComponentProjectionAdjoint(X, [0, 2])
>>> proj.domain
ProductSpace(Rn(1), Rn(3))
```

ProductSpaceOperator

class odl.operator.pspace_ops.**ProductSpaceOperator** (*operators, dom=None, ran=None*)

Bases: *odl.operator.operator.Operator*

A “matrix of operators” on product spaces.

This is intended for the case where an operator can be decomposed as a linear combination of “sub-operators”, e.g.

$$\begin{pmatrix} A & B & 0 \\ 0 & C & 0 \\ 0 & 0 & D \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} A(x) + B(y) \\ C(y) \\ D(z) \end{pmatrix}$$

Mathematically, a *ProductSpaceOperator* is an operator

$$\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Y}$$

between product spaces $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_m$ and $\mathcal{Y} = \mathcal{Y}_1 \times \cdots \times \mathcal{Y}_n$ which can be written in the form

$$\mathcal{A} = (\mathcal{A}_{ij})_{i,j}, \quad i = 1, \dots, n, \quad j = 1, \dots, m$$

with *component operators* $\mathcal{A}_{ij} : \mathcal{X}_j \rightarrow \mathcal{Y}_i$.

Its action on a vector $x = (x_1, \dots, x_m)$ is defined as the matrix multiplication

$$[\mathcal{A}(x)]_i = \sum_{j=1}^m \mathcal{A}_{ij}(x_j).$$

Notes

In many cases it is of interest to have an operator from a *ProductSpace* to any *LinearSpace*. In that case this operator can be used with a slight modification, simply run

```
prod_op = ProductSpaceOperator(prod_space, ProductSpace(linear_space))
```

The same can be done for operators *LinearSpace* -> *ProductSpace*

```
prod_op = ProductSpaceOperator(ProductSpace(linear_space), prod_space)
```

Attributes

<i>adjoint</i>	Adjoint of the product space operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ProductSpaceOperator.adjoint

`ProductSpaceOperator.adjoint`

Adjoint of the product space operator.

The adjoint is given by taking the transpose of the matrix and the adjoint of each component operator.

In weighted product spaces, the adjoint needs to take the weightings into account. This is currently not supported.

Returns`adjoint` : *ProductSpaceOperator*

The adjoint

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
>>> x = X.element([[1, 2, 3], [4, 5, 6]])
```

Matrix is transposed:

```
>>> prod_op = ProductSpaceOperator([[0, I], [0, 0]],
...                                dom=X, ran=X)
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
  [4.0, 5.0, 6.0],
  [0.0, 0.0, 0.0]
])
>>> prod_op.adjoint(x)
ProductSpace(Rn(3), 2).element([
  [0.0, 0.0, 0.0],
  [1.0, 2.0, 3.0]
])
```

ProductSpaceOperator.domain`ProductSpaceOperator.domain`

Set of objects on which this operator can be evaluated.

ProductSpaceOperator.inverse`ProductSpaceOperator.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ProductSpaceOperator.is_functional`ProductSpaceOperator.is_functional`True if the this operator's range is a *Field*.**ProductSpaceOperator.is_linear**`ProductSpaceOperator.is_linear`

True if this operator is linear.

ProductSpaceOperator.range`ProductSpaceOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Call the ProductSpace operators.
<code>derivative(x)</code>	Derivative of the product space operator.

ProductSpaceOperator.__call__`ProductSpaceOperator.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ProductSpaceOperator.__call

`ProductSpaceOperator.__call` (*x*, *out=None*)

Call the ProductSpace operators.

Parameters*x* : domain *element*

input vector to be evaluated

out : range *element*, optional

output vector to write result to

Returns*out* : range *element*

Result of the evaluation. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
>>> x = X.element([[1, 2, 3], [4, 5, 6]])
```

Sum of elements:

```
>>> prod_op = ProductSpaceOperator([I, I])
>>> prod_op(x)
ProductSpace(Rn(3), 1).element([
    [5.0, 7.0, 9.0]
])
```

Diagonal operator – 0 or None means ignore, or the implicit zero operator:

```
>>> prod_op = ProductSpaceOperator([[I, 0], [0, I]])
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
])
```

Complicated combinations:

```
>>> prod_op = ProductSpaceOperator([[I, I], [I, 0]])
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [5.0, 7.0, 9.0],
    [1.0, 2.0, 3.0]
])
```

ProductSpaceOperator.derivative

`ProductSpaceOperator.derivative(x)`

Derivative of the product space operator.

Parameters`x` : domain element

The point to take the derivative in

Returns`adjoint` : linear `'ProductSpaceOperator'`

The derivative

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
>>> x = X.element([[1, 2, 3], [4, 5, 6]])
```

Example with linear operator (derivative is itself)

```
>>> prod_op = ProductSpaceOperator([[0, I], [0, 0]],
...                                dom=X, ran=X)
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [4.0, 5.0, 6.0],
    [0.0, 0.0, 0.0]
])
>>> prod_op.derivative(x)(x)
ProductSpace(Rn(3), 2).element([
    [4.0, 5.0, 6.0],
    [0.0, 0.0, 0.0]
])
```

Example with affine operator

```
>>> residual_op = odl.ResidualOperator(I, r3.element([1, 1, 1]))
>>> op = ProductSpaceOperator([[0, residual_op], [0, 0]],
...                            dom=X, ran=X)
```

Calling operator gives offset by [1, 1, 1]

```
>>> op(x)
ProductSpace(Rn(3), 2).element([
    [3.0, 4.0, 5.0],
    [0.0, 0.0, 0.0]
])
```

Derivative of affine operator does not have this offset

```
>>> op.derivative(x)(x)
ProductSpace(Rn(3), 2).element([
    [4.0, 5.0, 6.0],
    [0.0, 0.0, 0.0]
])
```

`__init__(operators, dom=None, ran=None)`

Initialize a new instance.

Parameters`operators` : *array-like*

An array of *Operator*'s

dom : *ProductSpace*, optional

Domain of the operator. If not provided, it is tried to be inferred from the operators. This requires each **column** to contain at least one operator.

ran : *ProductSpace*, optional

Range of the operator. If not provided, it is tried to be inferred from the operators. This requires each **row** to contain at least one operator.

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
```

Sum of elements

```
>>> prod_op = ProductSpaceOperator([I, I])
```

Diagonal operator, 0 or None means ignore, or the implicit zero op.

```
>>> prod_op = ProductSpaceOperator([[I, 0], [None, I]])
```

Complicated combinations also possible

```
>>> prod_op = ProductSpaceOperator([[I, I], [I, 0]])
```

ReductionOperator

class odl.operator.pspace_ops.**ReductionOperator**(*operators)

Bases: *odl.operator.operator.Operator*

Reduce argument over set of operators.

An argument is reduced by evaluating several operators and summing the result

$\text{ReductionOperator}(\text{op1}, \text{op2})(x) = \text{op1}(x[0]) + \text{op2}(x[1])$

It is implemented using a *ProductSpaceOperator*.

Attributes

<i>adjoint</i>	Adjoint of the reduction operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>operators</i>	A tuple of sub-operators
<i>prod_op</i>	The prod-op implementation
<i>range</i>	Set in which the result of an evaluation of this operator lies.

ReductionOperator.adjoint

`ReductionOperator.adjoint`

Adjoint of the reduction operator.

Returns`adjoint` : linear *BroadcastOperator*

The adjoint

Examples

```
>>> import odl
>>> I = odl.IdentityOperator(odl.Rn(3))
>>> op = ReductionOperator(I, 2 * I)
>>> op.adjoint([1, 2, 3])
ProductSpace(Rn(3), 2).element([
    [1.0, 2.0, 3.0],
    [2.0, 4.0, 6.0]
])
```

ReductionOperator.domain

`ReductionOperator.domain`

Set of objects on which this operator can be evaluated.

ReductionOperator.inverse

`ReductionOperator.inverse`

Return the operator inverse.

Raises`OpNotImplementedError`

Since the inverse cannot be default implemented.

ReductionOperator.is_functional

`ReductionOperator.is_functional`

True if the this operator's range is a *Field*.

ReductionOperator.is_linear**ReductionOperator.is_linear**

True if this operator is linear.

ReductionOperator.operators**ReductionOperator.operators**

A tuple of sub-operators

ReductionOperator.prod_op**ReductionOperator.prod_op**

The prod-op implementation

ReductionOperator.range**ReductionOperator.range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Apply operators to <code>x</code> and sum.
<code>derivative(x)</code>	Derivative of the reduction operator.

ReductionOperator.__call__**ReductionOperator.__call__** (`x`, `out=None`, `**kwargs`)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ReductionOperator._call

`ReductionOperator._call(x, out=None)`

Apply operators to `x` and sum.

Parameters`x` : domain element

Input vector to be evaluated by operators

out : range element, optional

output vector to write result to

Returns`out` : range element

Sum of operators evaluated in point

Examples

```
>>> import odl
>>> I = odl.IdentityOperator(odl.Rn(3))
>>> op = ReductionOperator(I, 2 * I)
>>> op([[1.0, 2.0, 3.0], [4.0, 6.0, 8.0]])
Rn(3).element([9.0, 14.0, 19.0])
```

ReductionOperator.derivative

`ReductionOperator.derivative(x)`

Derivative of the reduction operator.

Parameters`x` : domain element

The point to take the derivative in

Returns`adjoint` : linear *BroadcastOperator*

The derivative

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> I = odl.IdentityOperator(r3)
>>> x = r3.element([1.0, 2.0, 3.0])
>>> y = r3.element([4.0, 6.0, 8.0])
```

Example with linear operator (derivative is itself)

```
>>> op = ReductionOperator(I, 2 * I)
>>> op([x, y])
Rn(3).element([9.0, 14.0, 19.0])
>>> op.derivative([x, y])([x, y])
Rn(3).element([9.0, 14.0, 19.0])
```

Example with affine operator

```
>>> residual_op = odl.ResidualOperator(I, r3.element([1, 1, 1]))
>>> op = ReductionOperator(residual_op, 2 * residual_op)
```

Calling operator gives offset by [3, 3, 3]

```
>>> op([x, y])
Rn(3).element([6.0, 11.0, 16.0])
```

Derivative of affine operator does not have this offset

```
>>> op.derivative([x, y])([x, y])
Rn(3).element([9.0, 14.0, 19.0])
```

`__init__` (*operators)

Functions

[`diagonal_operator`](#)(operators[, dom, ran]) Broadcast argument to set of operators.

`diagonal_operator`

`odl.operator.pspace_ops.diagonal_operator` (operators, dom=None, ran=None)

Broadcast argument to set of operators.

Parameters`operators` : array-like

An array of *Operator*'s

dom : *ProductSpace*, optional

Domain of the operator. If not provided, it is tried to be inferred from the operators. This requires each **column** to contain at least one operator.

ran : *ProductSpace*, optional

Range of the operator. If not provided, it is tried to be inferred from the operators. This requires each **row** to contain at least one operator.

2.8.4 set

Core Spaces and set support.

Modules

domain

Typical domains for inverse problems.

Classes

<i>IntervalProd</i> (begin, end)	An n-dimensional rectangular box.
----------------------------------	-----------------------------------

IntervalProd

class odl.set.domain.**IntervalProd**(begin, end)

Bases: *odl.set.sets.Set*

An n-dimensional rectangular box.

An interval product is a Cartesian product of n intervals, i.e. an n-dimensional rectangular box aligned with the coordinate axes as a subset of R^n .

IntervalProd objects are immutable, all methods involving them return a new *IntervalProd*.

Attributes

<i>area</i>	The length of this interval.
<i>begin</i>	The left interval boundary/boundaries.
<i>end</i>	The right interval boundary/boundaries.
<i>length</i>	The length of this interval.
<i>midpoint</i>	The midpoint of the interval product.
<i>ndim</i>	The number of intervals in the product.
<i>true_ndim</i>	The number of non-degenerate (zero-length) intervals.
<i>volume</i>	The 'dim'-dimensional volume of this interval product.

IntervalProd.area

IntervalProd.**area**

The length of this interval.

IntervalProd.begin

IntervalProd.**begin**

The left interval boundary/boundaries.

IntervalProd.end

IntervalProd.**end**

The right interval boundary/boundaries.

IntervalProd.length`IntervalProd.length`

The length of this interval.

IntervalProd.midpoint`IntervalProd.midpoint`

The midpoint of the interval product.

IntervalProd.ndim`IntervalProd.ndim`

The number of intervals in the product.

IntervalProd.true_ndim`IntervalProd.true_ndim`

The number of non-degenerate (zero-length) intervals.

IntervalProd.volume`IntervalProd.volume`

The ‘dim’-dimensional volume of this interval product.

Methods

<code>__contains__(other)</code>	Return <code>other</code> in <code>self</code> .
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__getitem__(indices)</code>	Return <code>self[indices]</code>
<code>append(other)</code>	Insert at the end.
<code>approx_contains(point, tol)</code>	Test if a point is contained.
<code>approx_equals(other, tol)</code>	Test if <code>other</code> is equal to this set up to <code>tol</code> .
<code>collapse(indices, values)</code>	Partly collapse the interval product to single values.
<code>contains_all(other)</code>	Test if all points defined by <code>other</code> are contained.
<code>contains_set(other[, tol])</code>	Test if another set is contained.
<code>corners([order])</code>	The corner points in a single array.
<code>dist(point[, ord])</code>	Calculate the distance to a point.
<code>element([inp])</code>	Create element in this set.
<code>extent()</code>	The interval length per axis.
<code>insert(index, other)</code>	Return a copy with <code>other</code> inserted before <code>index</code> .
<code>max()</code>	The maximum value in this interval product
<code>measure([ndim])</code>	The (Lebesgue) measure of this interval product.
<code>min()</code>	The minimum value in this interval product
<code>squeeze()</code>	Remove the degenerate dimensions.

IntervalProd.__contains__`IntervalProd.__contains__(other)`Return `other` in `self`.**Parameters**`other`

Arbitrary object to be tested.

Returns`contains` : bool

True if other is inside self.

Examples

```
>>> interv = IntervalProd(0, 1)
>>> 0.5 in interv
True
>>> 2 in interv
False
>>> 'string' in interv
False
```

IntervalProd.__eq__

IntervalProd.__eq__(other)
Return self == other.

IntervalProd.__getitem__

IntervalProd.__getitem__(indices)
Return self[indices]

Parametersindices : numpy style index

Any of: int, slice, list of ints

Returnssubinterval : *IntervalProd*

Interval given by the indices

Examples

```
>>> rbox = IntervalProd([-1, 2, 0], [-0.5, 3, 0.5])
```

By integer

```
>>> rbox[0]
Interval(-1.0, -0.5)
```

By slice

```
>>> rbox[:]
Cuboid([-1.0, 2.0, 0.0], [-0.5, 3.0, 0.5])
>>> rbox[::2]
Rectangle([-1.0, 0.0], [-0.5, 0.5])
```

By list of ints

```
>>> rbox[[0, 1]]
Rectangle([-1.0, 2.0], [-0.5, 3.0])
```

IntervalProd.append

IntervalProd.append(other)
Insert at the end.

Parametersother : *IntervalProd*, float or array-like

The set to be inserted. A float or array *a* is treated as an `IntervalProd(a, a)`.

See also:

`insert`

Examples

```
>>> rbox = IntervalProd([-1, 2], [-0.5, 3])
>>> rbox.append(Interval(-1.0, 0.0))
Cuboid([-1.0, 2.0, -1.0], [-0.5, 3.0, 0.0])
```

`IntervalProd.approx_contains`

`IntervalProd.approx_contains` (*point*, *tol*)

Test if a point is contained.

Parameters*point* : *array-like* or float

The point to be tested. Its length must be equal to the set's dimension. In the 1d case, 'point' can be given as a float.

tol : float

The maximum allowed distance in 'inf'-norm between the point and the set. Default: 0.0

Examples

```
>>> from math import sqrt
>>> b, e = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox = IntervalProd(b, e)
>>> # Numerical error
>>> rbox.approx_contains([-1 + sqrt(0.5)**2, 0., 2.9], tol=0)
False
>>> rbox.approx_contains([-1 + sqrt(0.5)**2, 0., 2.9], tol=1e-9)
True
```

`IntervalProd.approx_equals`

`IntervalProd.approx_equals` (*other*, *tol*)

Test if *other* is equal to this set up to *tol*.

Parameters*other* : object

The object to be tested

tol : float

The maximum allowed difference in 'inf'-norm between the interval endpoints.

Examples

```
>>> from math import sqrt
>>> rbox1 = IntervalProd(0, 0.5)
>>> rbox2 = IntervalProd(0, sqrt(0.5)**2)
>>> rbox1.approx_equals(rbox2, tol=0) # Num error
```



```
False
>>> rbox1.approx_equals(rbox2, tol=1e-15)
True
```

IntervalProd.collapse

IntervalProd.**collapse** (*indices*, *values*)

Partly collapse the interval product to single values.

Note that no changes are made in-place.

Parameters*indices* : int or tuple of int

The indices of the dimensions along which to collapse

values : *array-like* or float

The values to which to collapse. Must have the same length as *indices*. Values must lie within the interval boundaries.

Returns*collapsed* : *IntervalProd*

The collapsed set

Examples

```
>>> b, e = [-1, 0, 2], [-0.5, 1, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.collapse(1, 0)
Cuboid([-1.0, 0.0, 2.0], [-0.5, 0.0, 3.0])
>>> rbox.collapse([1, 2], [0, 2.5])
Cuboid([-1.0, 0.0, 2.5], [-0.5, 0.0, 2.5])
```

IntervalProd.contains_all

IntervalProd.**contains_all** (*other*)

Test if all points defined by *other* are contained.

Parameters*other* :

Can be a single point, a (*d*, *N*) array where *d* is the number of dimensions or a length-*d* meshgrid tuple

Returns*contains* : bool

True if all points are contained, False otherwise

Examples

```
>>> import odl
>>> b, e = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox = IntervalProd(b, e)
```

rrays are expected in (ndim, npoints) shape

```
>>> arr = np.array([[[-1, 0, 2], # defining one point at a time
...                  [-0.5, 0, 2]])
>>> rbox.contains_all(arr.T)
True
```

Implicit meshgrids defined by coordinate vectors

```
>>> from odl.discr.grid import sparse_meshgrid
>>> vec1 = (-1, -0.9, -0.7)
>>> vec2 = (0, 0, 0)
>>> vec3 = (2.5, 2.75, 3)
>>> mg = sparse_meshgrid(vec1, vec2, vec3)
>>> rbox.contains_all(mg)
True
```

Also works with any iterable

```
>>> rbox.contains_all([[ -1, -0.5], # define points by axis
...                  [0, 0],
...                  [2, 2]])
True
```

And with grids

```
>>> agrid = odl.uniform_sampling(rbox.begin, rbox.end, [3, 1, 3])
>>> rbox.contains_all(agrid)
True
```

IntervalProd.contains_set

`IntervalProd.contains_set` (*other*, *tol*=0.0)

Test if another set is contained.

Parameters*other* : *Set*

The set to be tested. It must implement a `min()` and a `max()` method, otherwise a `TypeError` is raised.

tol : float, optional

The maximum allowed distance in ‘inf’-norm between the other set and this interval product. Default: 0.0

Examples

```
>>> b1, e1 = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox1 = IntervalProd(b1, e1)
>>> b2, e2 = [-0.6, 0, 2.1], [-0.5, 0, 2.5]
>>> rbox2 = IntervalProd(b2, e2)
>>> rbox1.contains_set(rbox2)
True
>>> rbox2.contains_set(rbox1)
False
```

IntervalProd.corners

`IntervalProd.corners` (*order*='C')

The corner points in a single array.

Parameters*order* : {'C', 'F'}

The ordering of the axes in which the corners appear in the output. ‘C’ means that the first axis varies slowest and the last one fastest, vice versa in ‘F’ ordering.

Returns*corners* : `numpy.ndarray`

The size of the array is $2^m * \text{ndim}$, where m is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> rbox = IntervalProd([-1, 2, 0], [-0.5, 3, 0.5])
>>> rbox.corners()
array([[ -1. ,  2. ,  0. ],
       [ -1. ,  2. ,  0.5],
       [ -1. ,  3. ,  0. ],
       [ -1. ,  3. ,  0.5],
       [ -0.5,  2. ,  0. ],
       [ -0.5,  2. ,  0.5],
       [ -0.5,  3. ,  0. ],
       [ -0.5,  3. ,  0.5]])
>>> rbox.corners(order='F')
array([[ -1. ,  2. ,  0. ],
       [ -0.5,  2. ,  0. ],
       [ -1. ,  3. ,  0. ],
       [ -0.5,  3. ,  0. ],
       [ -1. ,  2. ,  0.5],
       [ -0.5,  2. ,  0.5],
       [ -1. ,  3. ,  0.5],
       [ -0.5,  3. ,  0.5]])
```

IntervalProd.dist

IntervalProd.**dist** (*point*, *ord=2.0*)

Calculate the distance to a point.

Parameters*point* : *array-like* or float

The point. Its length must be equal to the set's dimension. Can be a float in the 1d case.

ord : non-zero int or float('inf'), optional

The order of the norm (see `numpy.linalg.norm`). Default: 2.0

Returns*dist* : float

Distance to the interior of the IntervalProd. Points strictly inside have distance 0.0, points with NaN have distance infinity.

Examples

```
>>> b, e = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.dist([-5, 3, 2])
5.0
>>> rbox.dist([-5, 3, 2], ord=float('inf'))
4.0
```

IntervalProd.element

IntervalProd.**element** (*inp=None*)

Create element in this set.

Parameters`inp` : float or array-like, optional

Point to be cast to an element in self

Return`element`

Returns `inp` if given, else `self.midpoint`

Raises`TypeError`

If `inp` is not a valid element.

Examples

```
>>> interv = IntervalProd(0, 1)
>>> interv.element(0.5)
0.5
```

IntervalProd.extent

`IntervalProd.extent()`

The interval length per axis.

IntervalProd.insert

`IntervalProd.insert(index, other)`

Return a copy with `other` inserted before `index`.

The given interval product (`ndim=m`) is inserted into the current one (`ndim=n`) before the given index, resulting in a new interval product with `n+m` dimensions.

Parameters`index` : int

Index of the dimension before which `other` is to be inserted. Must fulfill `-ndim <= index <= ndim`. Negative indices count backwards from `self.ndim`.

other : *IntervalProd*

Interval product to be inserted

Returns`newintvp` : *IntervalProd*

Interval product with `other` inserted

Examples

```
>>> rbox = IntervalProd([-1, 2], [-0.5, 3])
>>> rbox2 = IntervalProd([0, 0], [1, 0])
>>> rbox.insert(1, rbox2)
IntervalProd([-1.0, 0.0, 0.0, 2.0], [-0.5, 1.0, 0.0, 3.0])
>>> rbox.insert(-1, rbox2)
IntervalProd([-1.0, 0.0, 0.0, 2.0], [-0.5, 1.0, 0.0, 3.0])
```

IntervalProd.max

`IntervalProd.max()`

The maximum value in this interval product

IntervalProd.measureIntervalProd.**measure** (*ndim=None*)

The (Lebesgue) measure of this interval product.

Parameters*ndim* : int, optionalThe dimension of the measure to apply. Default: *true_ndim***Examples**

```

>>> b, e = [-1, 2.5, 0], [-0.5, 10, 0]
>>> rbox = IntervalProd(b, e)
>>> rbox.measure()
3.75
>>> rbox.measure(ndim=3)
0.0
>>> rbox.measure(ndim=3) == rbox.volume
True
>>> rbox.measure(ndim=1)
inf
>>> rbox.measure() == rbox.squeeze().volume
True

```

IntervalProd.minIntervalProd.**min**()

The minimum value in this interval product

IntervalProd.squeezeIntervalProd.**squeeze**()

Remove the degenerate dimensions.

Note that no changes are made in-place.

Returnssqueezed : *IntervalProd*

The squeezed set

Examples

```

>>> b, e = [-1, 0, 2], [-0.5, 1, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.collapse(1, 0).squeeze()
Rectangle([-1.0, 2.0], [-0.5, 3.0])
>>> rbox.collapse([1, 2], [0, 2.5]).squeeze()
Interval(-1.0, -0.5)
>>> rbox.collapse([0, 1, 2], [-1, 0, 2.5]).squeeze()
IntervalProd([], [])

```

__init__ (*begin, end*)

Initialize a new instance.

Parameters*begin* : *array-like* or float

The lower ends of the intervals in the product

end : *array-like* or float

The upper ends of the intervals in the product

Examples

```
>>> b, e = [-1, 2.5, 70, 80], [-0.5, 10, 75, 90]
>>> rbox = IntervalProd(b, e)
>>> rbox
IntervalProd([-1.0, 2.5, 70.0, 80.0], [-0.5, 10.0, 75.0, 90.0])
```

Functions

<code>Cuboid(begin, end)</code>	Three-dimensional interval product.
<code>Interval(begin, end)</code>	One-dimensional interval product.
<code>Rectangle(begin, end)</code>	Two-dimensional interval product.

Cuboid

`odl.set.domain.Cuboid(begin, end)`
Three-dimensional interval product.

Parameters
begin : *array-like*, shape (3,)
The lower ends of the intervals in the product
end : *array-like*, shape (3,)
The upper ends of the intervals in the product

Interval

`odl.set.domain.Interval(begin, end)`
One-dimensional interval product.

Parameters
begin : *array-like*, shape (1,), or float
The lower ends of the intervals in the product
end : *array-like*, shape (1,), or float
The upper ends of the intervals in the product

Rectangle

`odl.set.domain.Rectangle(begin, end)`
Two-dimensional interval product.

Parameters
begin : *array-like*, shape (2,)
The lower ends of the intervals in the product
end : *array-like*, shape (2,)
The upper ends of the intervals in the product

pspace

Cartesian products of *LinearSpace* instances.

Classes

<i>ProductSpace</i> (*spaces, **kwargs)	Cartesian product of <i>LinearSpace</i> 's.
<i>ProductSpaceVector</i> (space, parts)	Elements of a <i>ProductSpace</i> .

ProductSpace

class odl.set.pspace.**ProductSpace**(*spaces, **kwargs)

Bases: odl.set.space.*LinearSpace*

Cartesian product of *LinearSpace*'s.

Attributes

<i>element_type</i>	<i>ProductSpaceVector</i>
<i>field</i>	The field of this vector space.
<i>size</i>	The number of factors.
<i>spaces</i>	A tuple containing all spaces.
<i>weights</i>	Weighting vector or scalar of this product space.

ProductSpace.element_type

ProductSpace.**element_type**

ProductSpaceVector

ProductSpace.field

ProductSpace.**field**

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

ProductSpace.size

ProductSpace.**size**

The number of factors.

ProductSpace.spaces

ProductSpace.**spaces**

A tuple containing all spaces.

ProductSpace.weightsProductSpace.**weights**

Weighting vector or scalar of this product space.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>__getitem__(indices)</code>	Return self[indices].
<code>_dist(x1, x2)</code>	Distance between two vectors.
<code>_divide(x1, x2, out)</code>	Quotient out = x1 / x2.
<code>_inner(x1, x2)</code>	Inner product of two vectors.
<code>_lincomb(a, x, b, y, out)</code>	Linear combination out = a*x + b*y.
<code>_multiply(x1, x2, out)</code>	Product out = x1 * x2.
<code>_norm(x)</code>	Norm of a vector.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp, cast])</code>	Create an element in the product space.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create the one vector of the product space.
<code>zero()</code>	Create the zero vector of the product space.

ProductSpace.__contains__ProductSpace.**__contains__** (*other*)

Return other in self.

Returnscontains : boolTrue if other is a *LinearSpaceVector* instance and other.space is equal to this space, False otherwise.**Notes**

This is the strict default where spaces must be equal. Subclasses may choose to implement a less strict check.

ProductSpace.__eq__ProductSpace.**__eq__** (*other*)

Return self == other.

Returnsequals : boolTrue if other is a *ProductSpace* instance, has the same length and the same factors. False otherwise.

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> rn, rm = Rn(2), Rn(3)
>>> r2x3, rnxm = ProductSpace(r2, r3), ProductSpace(rn, rm)
>>> r2x3 == rnxm
True
>>> r3x2 = ProductSpace(r3, r2)
>>> r2x3 == r3x2
False
>>> r5 = ProductSpace(*[Rn(1)]*5)
>>> r2x3 == r5
False
>>> r5 = Rn(5)
>>> r2x3 == r5
False
```

ProductSpace.__getitem__

ProductSpace.__getitem__(*indices*)
Return self[indices].

ProductSpace._dist

ProductSpace._dist(*x1*, *x2*)
Distance between two vectors.

ProductSpace._divide

ProductSpace._divide(*x1*, *x2*, *out*)
Quotient out = $x1 / x2$.

ProductSpace._inner

ProductSpace._inner(*x1*, *x2*)
Inner product of two vectors.

ProductSpace._lincomb

ProductSpace._lincomb(*a*, *x*, *b*, *y*, *out*)
Linear combination out = $a*x + b*y$.

ProductSpace._multiply

ProductSpace._multiply(*x1*, *x2*, *out*)
Product out = $x1 * x2$.

ProductSpace._norm

ProductSpace._norm(*x*)
Norm of a vector.

ProductSpace.contains_all

`ProductSpace.contains_all` (*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

ProductSpace.contains_set

`ProductSpace.contains_set` (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

ProductSpace.dist

`ProductSpace.dist` (*x1*, *x2*)

Calculate the distance between two vectors.

Parameters*x1*, *x2* : *LinearSpaceVector*

Vectors whose distance to compute

Returns*dist* : float

Distance between vectors

ProductSpace.divide

`ProductSpace.divide` (*x1*, *x2*, *out=None*)

Calculate the pointwise division of *x1* and *x2*

Parameters*x1* : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns*out* : *LinearSpaceVector*

Ratio of the vectors. If *out* was provided, the returned object is a reference to it.

ProductSpace.element

`ProductSpace.element` (*inp=None*, *cast=True*)

Create an element in the product space.

Parameters*inp* : optional

If *inp* is *None*, a new element is created from scratch by allocation in the spaces. If *inp* is already an element of this space, it is re-wrapped. Otherwise, a new element is created from the components by calling the `element()` methods in the component spaces.

cast : bool

True if casting should be allowed

Return*element* : *ProductSpaceVector*

The new element

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> vec_2, vec_3 = r2.element(), r3.element()
>>> r2x3 = ProductSpace(r2, r3)
>>> vec_2x3 = r2x3.element()
>>> vec_2.space == vec_2x3[0].space
True
>>> vec_3.space == vec_2x3[1].space
True
```

Create an element of the product space

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> prod = ProductSpace(r2, r3)
>>> x2 = r2.element([1, 2])
>>> x3 = r3.element([1, 2, 3])
>>> x = prod.element([x2, x3])
>>> print(x)
{[1.0, 2.0], [1.0, 2.0, 3.0]}
```

ProductSpace.inner

ProductSpace.**inner**(*x1*, *x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1*, *x2* : *LinearSpaceVector*

Factors in the inner product

Returns*out* : *LinearSpace.field* element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

ProductSpace.lincomb

ProductSpace.**lincomb**(*a*, *x1*, *b=None*, *x2=None*, *out=None*)

Linear combination of vectors.

Calculates

$\text{out} = a * x1$

or, if *b* and *y* are given,

$\text{out} = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply x_2 with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns **out** : *LinearSpaceVector*

Result of the linear combination. If **out** was provided, the returned object is a reference to it.

Notes

The vectors **out**, **x1** and **x2** may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

ProductSpace.multiply

`ProductSpace.multiply(x1, x2, out=None)`

Calculate the pointwise product of **x1** and **x2**.

Parameters **x1, x2** : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns **out** : *LinearSpaceVector*

Product of the vectors. If **out** was provided, the returned object is a reference to it.

ProductSpace.norm

`ProductSpace.norm(x)`

Calculate the norm of a vector.

Parameters **x** : *LinearSpaceVector*

The vector

Returns **out** : float

Norm of the vector

ProductSpace.one

`ProductSpace.one()`

Create the one vector of the product space.

The i :th component of the product space one vector is the one vector of the i :th space in the product.

Parameters **None**

Returns **one** : *ProductSpaceVector*

The one vector in the product space

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> one_2, one_3 = r2.one(), r3.one()
>>> r2x3 = ProductSpace(r2, r3)
>>> one_2x3 = r2x3.one()
>>> one_2 == one_2x3[0]
True
>>> one_3 == one_2x3[1]
True
```

ProductSpace.zero

ProductSpace.**zero**()

Create the zero vector of the product space.

The i :th component of the product space zero vector is the zero vector of the i :th space in the product.

ParametersNone

Returnszero : ProductSpaceVector

The zero vector in the product space

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> zero_2, zero_3 = r2.zero(), r3.zero()
>>> r2x3 = ProductSpace(r2, r3)
>>> zero_2x3 = r2x3.zero()
>>> zero_2 == zero_2x3[0]
True
>>> zero_3 == zero_2x3[1]
True
```

__init__(*spaces, **kwargs)

Initialize a new instance.

The Cartesian product $\mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ for linear spaces \mathcal{X}_i is itself a linear space, where the linear combination is defined component-wise.

Parametersspaces : *LinearSpace* or int

Can be specified either as a space and an integer, in which case the power space `space**n` is created, or an arbitrary number of spaces.

ord : float, optional

Order of the product distance/norm, i.e.

`dist(x, y) = np.linalg.norm(x-y, ord=ord)`

`norm(x) = np.linalg.norm(x, ord=ord)`

Default: 2.0

The following `float` values for `ord` can be specified. Note that any value of `ord < 1` only gives a pseudo-norm.

'prod_norm'	Distance Definition
'inf'	$\max(w * z)$
'-inf'	$\min(w * z)$
other	$\text{sum}(w * z^{**ord}) ** (1/ord)$

Here,

```
z = (x[0].dist(y[0]), ..., x[n-1].dist(y[n-1]))
```

and `w = weights`.

Note that $0 \leq \text{ord} < 1$ are not allowed since these pseudo-norms are very unstable numerically.

weights : *array-like*, optional

Array of weights, same size as number of space components. All weights must be positive. It is multiplied with the tuple of distances before applying the `Rn` norm or `prod_norm`. Default: `(1.0, ..., 1.0)`

This option can only be used together with `ord`.

prod_norm : callable, optional

Function that should be applied to the array of distances/norms. Specifying a product norm causes the space to NOT be a Hilbert space.

Default: `np.linalg.norm(x, ord=ord)`.

field : *Field*, optional

The field that should be used. Default: `spaces[0].field`

Returns`prodspace` : *ProductSpace*

Examples

```
>>> from odl import Rn
>>> r2x3 = ProductSpace(Rn(2), Rn(3))
```

ProductSpaceVector

class `odl.set.pspace.ProductSpaceVector` (*space*, *parts*)

Bases: `odl.set.space.LinearSpaceVector`

Elements of a *ProductSpace*.

Attributes

<i>T</i>	The transpose of a vector, the functional given by $(\cdot, T \cdot)$.
<i>parts</i>	The parts of this vector.
<i>size</i>	The number of factors of this vector's space.
<i>space</i>	Space to which this vector belongs.
<i>ufunc</i>	<i>ProductSpaceUFuncs</i> , access to numpy style ufuncs.

ProductSpaceVector.T`ProductSpaceVector.T`

The transpose of a vector, the functional given by (\cdot, self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```

>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0

```

ProductSpaceVector.parts`ProductSpaceVector.parts`

The parts of this vector.

ProductSpaceVector.size`ProductSpaceVector.size`

The number of factors of this vector's space.

ProductSpaceVector.space`ProductSpaceVector.space`

Space to which this vector belongs.

LinearSpace

ProductSpaceVector.ufunc`ProductSpaceVector.ufunc`

ProductSpaceUFuncs, access to numpy style ufuncs.

These are always available if the underlying spaces are *NtuplesBase*.

See also:

`odl.util.ufuncs.NtuplesBaseUFuncs` Base class for ufuncs in *NtuplesBase* spaces, subspaces may override this for greater efficiency.

`odl.util.ufuncs.ProductSpaceUFuncs` For a list of available ufuncs.

Examples

```
>>> from odl import Rn
>>> r22 = ProductSpace(Rn(2), 2)
>>> x = r22.element([[1, -2], [-3, 4]])
>>> x.ufunc.absolute()
ProductSpace(Rn(2), 2).element([
    [1.0, 2.0],
    [3.0, 4.0]
])
```

These functions can also be used with non-vector arguments and support broadcasting, both by element

```
>>> x.ufunc.add([1, 1])
ProductSpace(Rn(2), 2).element([
    [2.0, -1.0],
    [-2.0, 5.0]
])
```

and also recursively

```
>>> x.ufunc.subtract(1)
ProductSpace(Rn(2), 2).element([
    [0.0, -3.0],
    [-4.0, 3.0]
])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
0.0
```

Also supports out parameter

```
>>> y = r22.element()
>>> result = x.ufunc.absolute(out=y)
>>> result
ProductSpace(Rn(2), 2).element([
    [1.0, 2.0],
    [3.0, 4.0]
])
>>> result is y
True
```

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__getitem__(indices)</code>	Return <code>self[indices]</code> .
<code>__setitem__(indices, values)</code>	Implement <code>self[indices] = vals</code> .
<code>assign(other)</code>	Assign the values of <code>other</code> to <code>self</code> .
<code>copy()</code>	Create an identical (deep) copy of <code>self</code> .
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.

Continued on next page

Table 2.111 – continued from previous page

<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, indices])</code>	Display the parts of this vector graphically

ProductSpaceVector.__eq__

`ProductSpaceVector.__eq__ (other)`

Return `self == other`.

Overrides the default `LinearSpace` method since it is implemented with the distance function, which is prone to numerical errors. This function checks equality per component.

ProductSpaceVector.__getitem__

`ProductSpaceVector.__getitem__ (indices)`

Return `self[indices]`.

ProductSpaceVector.__setitem__

`ProductSpaceVector.__setitem__ (indices, values)`

Implement `self[indices] = vals`.

ProductSpaceVector.assign

`ProductSpaceVector.assign (other)`

Assign the values of `other` to `self`.

ProductSpaceVector.copy

`ProductSpaceVector.copy ()`

Create an identical (deep) copy of `self`.

ProductSpaceVector.dist

`ProductSpaceVector.dist (other)`

Distance to `other`.

`LinearSpace.dist`

ProductSpaceVector.divide

`ProductSpaceVector.divide (x, y)`

Divide by `other` inplace.

`LinearSpace.divide`

ProductSpaceVector.inner

`ProductSpaceVector.inner (other)`

Inner product with `other`.

`LinearSpace.inner`

ProductSpaceVector.lincomb`ProductSpaceVector.lincomb(a, x1, b=None, x2=None)`

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

ProductSpaceVector.multiply`ProductSpaceVector.multiply(x, y)`

Multiply by other inplace.

LinearSpace.multiply

ProductSpaceVector.norm`ProductSpaceVector.norm()`

Norm of vector

LinearSpace.norm

ProductSpaceVector.set_zero`ProductSpaceVector.set_zero()`

Set this vector to zero.

LinearSpace.zero

ProductSpaceVector.show`ProductSpaceVector.show(title=None, indices=None, **kwargs)`

Display the parts of this vector graphically

Parameter`title : str`

Title of the figures

indices : index expression, optional

Indices can refer to parts of a *ProductSpaceVector* and slices in the parts in the following way:

Single index (`indices=0`) => display that part

Single slice (`indices=slice(None)`), or index list (`indices=[0, 1, 3]`) => display those parts

Any tuple, for example: Created by `numpy.s_` `indices=np.s_[0, :, :]` or Using a raw tuple `indices=(0, 3, slice(None))` => take the first elements to select the parts and pass the rest on to the underlying show methods.

kwargs

Additional arguments passed on to the underlying vectors

Returns`fig` : list of `matplotlib.figure.Figure`

The resulting figures. It is also shown to the user.

See also:

odl.discr.lp_discr.DiscreteLpVector.show Display of a discretized function

`odl.space.base_ntuples.NtuplesBaseVector.show` Display of sequence type data

`odl.util.graphics.show_discrete_data` Underlying implementation

`__init__` (*space, parts*)
“Initialize a new instance.”

sets

Basic abstract and concrete sets.

Classes

<i>CartesianProduct</i> (*sets)	The Cartesian product of n sets.
<i>ComplexNumbers</i>	The set of complex numbers.
<i>EmptySet</i>	The empty set.
<i>Field</i>	Any set that satisfies the field axioms
<i>Integers</i>	The set of integers.
<i>RealNumbers</i>	The set of real numbers.
<i>Set</i>	An abstract set.
<i>Strings</i> (length)	The set of fixed-length (unicode) strings.
<i>UniversalSet</i>	The set of all objects.

CartesianProduct

class `odl.set.sets.CartesianProduct` (*sets)

Bases: `odl.set.sets.Set`

The Cartesian product of n sets.

The elements of this set are n-tuples where the i-th entry is an element of the i-th set.

Attributes

<i>sets</i>	The factors (sets) as a tuple.
-------------	--------------------------------

CartesianProduct.sets

`CartesianProduct.sets`

The factors (sets) as a tuple.

Methods

<code>__contains__</code> (other)	Test if other is contained in this set.
<code>__eq__</code> (other)	Return self == other.
<code>__getitem__</code> (indcs)	Return self[indcs].
<code>contains_all</code> (other)	Test if all points in other are contained in this set.
<code>contains_set</code> (other)	Test if other is a subset of this set.
<code>element</code> ([inp])	Create a <i>CartesianProduct</i> element.

CartesianProduct.__contains__

`CartesianProduct.__contains__(other)`

Test if `other` is contained in this set.

Returns`contains` : bool

True if `other` has the same length as this Cartesian product and each entry is contained in the set with corresponding index, False otherwise.

CartesianProduct.__eq__

`CartesianProduct.__eq__(other)`

Return `self == other`.

Return`sequals` : bool

True if `other` is a `CartesianProduct` instance, has the same length as this Cartesian product and all sets with the same index are equal, False otherwise.

CartesianProduct.__getitem__

`CartesianProduct.__getitem__(inds)`

Return `self[inds]`.

Examples

```
>>> emp, univ = EmptySet(), UniversalSet()
>>> prod = CartesianProduct(emp, univ, univ, emp, emp)
>>> prod[2]
UniversalSet()
>>> prod[2:4]
CartesianProduct(UniversalSet(), EmptySet())
```

CartesianProduct.contains_all

`CartesianProduct.contains_all(other)`

Test if all points in `other` are contained in this set.

This is a default implementation and should be overridden by subclasses.

CartesianProduct.contains_set

`CartesianProduct.contains_set(other)`

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

CartesianProduct.element

`CartesianProduct.element(inp=None)`

Create a `CartesianProduct` element.

Parameters`inp` : iterable, optional

Collection of input values for the `LinearSpace.element` methods of all sets in the Cartesian product.

Return`selement` : tuple

A tuple of the given input

`__init__` (*sets)
Initialize a new instance.

ComplexNumbers

`class odl.set.sets.ComplexNumbers`

Bases: `odl.set.sets.Field`

The set of complex numbers.

Attributes

`field` The field of scalars for a field is itself.

ComplexNumbers.field

`ComplexNumbers.field`

The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with `LinearSpace`'s.

Methods

<code>__contains__</code> (other)	Test if other is a complex number.
<code>__eq__</code> (other)	Return self == other.
<code>contains_all</code> (array)	Test if array is an array of real or complex numbers.
<code>contains_set</code> (other)	Test if other is a subset of the complex numbers
<code>element</code> ([inp])	Return a complex number from inp or from scratch.

ComplexNumbers.__contains__

`ComplexNumbers.__contains__` (other)

Test if other is a complex number.

ComplexNumbers.__eq__

`ComplexNumbers.__eq__` (other)

Return self == other.

ComplexNumbers.contains_all

`ComplexNumbers.contains_all` (array)

Test if array is an array of real or complex numbers.

ComplexNumbers.contains_set

`ComplexNumbers.contains_set` (other)

Test if other is a subset of the complex numbers

Returnscontained : bool

True if other is *ComplexNumbers*, *RealNumbers* or *Integers*, False else.

Examples

```
>>> C = ComplexNumbers()
>>> C.contains_set(RealNumbers())
True
```

ComplexNumbers.element

`ComplexNumbers.element(inp=None)`

Return a complex number from `inp` or from scratch.

EmptySet

`class odl.set.sets.EmptySet`

Bases: `odl.set.sets.Set`

The empty set.

None is considered as “no element”, i.e. `None in EmptySet()` is True

Methods

<code>__contains__(other)</code>	Test if other is None.
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Return True for the empty set, otherwise False.
<code>element([inp])</code>	Return None.

EmptySet.__contains__

`EmptySet.__contains__(other)`

Test if other is None.

EmptySet.__eq__

`EmptySet.__eq__(other)`

Return `self == other`.

EmptySet.contains_all

`EmptySet.contains_all(other)`

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

EmptySet.contains_set

`EmptySet.contains_set(other)`

Return True for the empty set, otherwise False.

EmptySet.element

EmptySet.**element** (*inp=None*)

Return None.

Field

class odl.set.sets.**Field**

Bases: *odl.set.sets.Set*

Any set that satisfies the field axioms

For example *RealNumbers*, *ComplexNumbers* or the finite field F_2 .

Attributes

<i>field</i>	The field of scalars for a field is itself.
--------------	---

Field.field

Field.**field**

The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with *LinearSpace*'s.

Methods

<i>__contains__</i> (other)	Return other in self.
<i>__eq__</i> (other)	Return self == other.
<i>contains_all</i> (other)	Test if all points in other are contained in this set.
<i>contains_set</i> (other)	Test if other is a subset of this set.
<i>element</i> ([inp])	Return an element from inp or from scratch.

Field.__contains__

Field.**__contains__** (*other*)

Return other in self.

Field.__eq__

Field.**__eq__** (*other*)

Return self == other.

Field.contains_all

Field.**contains_all** (*other*)

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

Field.contains_set`Field.contains_set (other)`Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Field.element`Field.element (inp=None)`Return an element from `inp` or from scratch.

Implementing this method is optional.

Integers`class odl.set.sets.Integers`Bases: `odl.set.sets.Set`

The set of integers.

Methods

<code>__contains__(other)</code>	Test if <code>other</code> is an integer.
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>contains_all(array)</code>	Test if <code>array</code> is an array of integers.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of the real numbers
<code>element([inp])</code>	Return an integer from <code>inp</code> or from scratch.

Integers.__contains__`Integers.__contains__ (other)`Test if `other` is an integer.**Integers.__eq__**`Integers.__eq__ (other)`Return `self == other`.**Integers.contains_all**`Integers.contains_all (array)`Test if `array` is an array of integers.**Integers.contains_set**`Integers.contains_set (other)`Test if `other` is a subset of the real numbers**Returns**`contained : bool`True if `other` is `Integers`, False otherwise.

Examples

```
>>> Z = Integers()
>>> Z.contains_set(RealNumbers())
False
```

Integers.element

`Integers.element(inp=None)`

Return an integer from `inp` or from scratch.

RealNumbers

`class odl.set.sets.RealNumbers`

Bases: `odl.set.sets.Field`

The set of real numbers.

Attributes

`field` The field of scalars for a field is itself.

RealNumbers.field

`RealNumbers.field`

The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with `LinearSpace`'s.

Methods

<code>__contains__(other)</code>	Test if <code>other</code> is a real number.
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>contains_all(array)</code>	Test if <code>array</code> is an array of real numbers.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of the real numbers
<code>element([inp])</code>	Return a real number from <code>inp</code> or from scratch.

RealNumbers.__contains__

`RealNumbers.__contains__(other)`

Test if `other` is a real number.

RealNumbers.__eq__

`RealNumbers.__eq__(other)`

Return `self == other`.

RealNumbers.contains_all

`RealNumbers.contains_all(array)`
Test if `array` is an array of real numbers.

RealNumbers.contains_set

`RealNumbers.contains_set(other)`
Test if `other` is a subset of the real numbers

Returns`contained` : bool

True if `other` is *RealNumbers* or *Integers* False else.

Examples

```
>>> R = RealNumbers()
>>> R.contains_set(RealNumbers())
True
```

RealNumbers.element

`RealNumbers.element(inp=None)`
Return a real number from `inp` or from scratch.

Set

class `odl.set.sets.Set`

Bases: `object`

An abstract set.

Abstract Methods

Each subclass of *Set* must implement two methods: one to check if an object is contained in the set and one to test if two sets are equal.

Membership test: `__contains__(self, other)`

Test if `other` is a member of this set. This function provides the operator overload for `in`.

Parameters:

other[`object`] The object to be tested for membership

Returns:

contains[`bool`] True if `other` is a member of this set, False otherwise.

Equality test: `__eq__(self, other)`

Test if `other` is the same set as this set, i.e. both sets are of the same type and contain the same elements. This function provides the operator overload for `==`.

Parameters:

other[`object`] The object to be tested for equality.

Returns:

equals[`bool`] True if both sets are of the same type and contain the same elements, False otherwise.

A default implementation of the operator overload for `!=` via `__ne__(self, other)` is provided as not `self.__eq__(other)`.

Element creation (optional): `element(self, inp=None)`

Create an element of this set, either from scratch or from an input parameter.

Parameters:

inp[object, optional] The object from which to create the new element

Returns:

element[member of this set] If `inp` is `None`, return an arbitrary element. Otherwise, return the element created from `inp`.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>element([inp])</code>	Return an element from <code>inp</code> or from scratch.

Set.__contains__

Set.**__contains__**(*other*)
Return other in self.

Set.__eq__

Set.**__eq__**(*other*)
Return self == other.

Set.contains_all

Set.**contains_all**(*other*)
Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

Set.contains_set

Set.**contains_set**(*other*)
Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Set.element

Set.**element**(*inp=None*)
Return an element from *inp* or from scratch.

Implementing this method is optional.

Strings

class `odl.set.sets.Strings` (*length*)

Bases: `odl.set.sets.Set`

The set of fixed-length (unicode) strings.

Attributes

<i>length</i>	The length attribute.
---------------	-----------------------

Strings.length

`Strings.length`

The length attribute.

Methods

<code>__contains__</code> (<i>other</i>)	Return <i>other</i> in <i>self</i> .
<code>__eq__</code> (<i>other</i>)	Return <i>self</i> == <i>other</i> .
<code>contains_all</code> (<i>array</i>)	Test if <i>array</i> is an array of strings with correct length.
<code>contains_set</code> (<i>other</i>)	Test if <i>other</i> is a subset of this set.
<code>element</code> (<i>[inp]</i>)	Return a string from <i>inp</i> or from scratch.

Strings.__contains__

`Strings.__contains__` (*other*)

Return *other* in *self*.

True if *other* is a string of at max *length* characters, False otherwise.

Strings.__eq__

`Strings.__eq__` (*other*)

Return *self* == *other*.

Strings.contains_all

`Strings.contains_all` (*array*)

Test if *array* is an array of strings with correct length.

Strings.contains_set

`Strings.contains_set` (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Strings.element

`Strings.element` (*inp=None*)

Return a string from *inp* or from scratch.

`__init__` (*length*)

Initialize a new instance.

Parameters`length : int`

The fixed length of the strings in this set. Must be positive.

UniversalSet

class `odl.set.sets.UniversalSet`

Bases: `odl.set.sets.Set`

The set of all objects.

Forget about set theory for a moment :-).

Methods

<code>__contains__(other)</code>	Return True.
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>contains_all(other)</code>	Test if all points in <code>other</code> are contained in this set.
<code>contains_set(other)</code>	Return True for any set.
<code>element([inp])</code>	Return <code>inp</code> in any case.

UniversalSet.__contains__

`UniversalSet.__contains__(other)`

Return True.

UniversalSet.__eq__

`UniversalSet.__eq__(other)`

Return `self == other`.

UniversalSet.contains_all

`UniversalSet.contains_all(other)`

Test if all points in `other` are contained in this set.

This is a default implementation and should be overridden by subclasses.

UniversalSet.contains_set

`UniversalSet.contains_set(other)`

Return True for any set.

UniversalSet.element

`UniversalSet.element(inp=None)`

Return `inp` in any case.

space

Abstract linear vector spaces.

The classes in this module represent abstract mathematical concepts of vector spaces. They cannot be used directly but are rather intended to be sub-classed by concrete space implementations. The spaces provide default implementations of the most important vector space operations. See the documentation of the respective classes for more details.

The abstract *LinearSpace* class is intended for quick prototyping. It has a number of abstract methods which must be overridden by a subclass. On the other hand, it provides automatic error checking and numerous attributes and methods for convenience.

Classes

<i>LinearSpace</i> (field)	Abstract linear vector space.
<i>LinearSpaceNotImplementedError</i>	Exception for not implemented errors in <i>LinearSpace</i> 's.
<i>LinearSpaceTypeError</i>	Exception for type errors in <i>LinearSpace</i> 's.
<i>LinearSpaceVector</i> (space)	Abstract <i>LinearSpace</i> element.
<i>UniversalSpace</i> ()	A dummy linear space class.

LinearSpace

class odl.set.space.**LinearSpace** (*field*)

Bases: *odl.set.sets.Set*

Abstract linear vector space.

Its elements are represented as instances of the inner *LinearSpaceVector* class.

Attributes

<i>element_type</i>	<i>LinearSpaceVector</i>
<i>field</i>	The field of this vector space.

LinearSpace.element_type

LinearSpace.**element_type**

LinearSpaceVector

LinearSpace.field

LinearSpace.**field**

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

Methods

<i>__contains__</i> (other)	Return other in self.
<i>__eq__</i> (other)	Return self == other.
<i>_dist</i> (x1, x2)	Calculate the distance between x1 and x2.
<i>_inner</i> (x1, x2)	Calculate the inner product of x1 and x2.
<i>_lincomb</i> (a, x1, b, x2, out)	Calculate out = a*x1 + b*x2.

Continued on next page

Table 2.129 – continued from previous page

<code>_multiply(x1, x2, out)</code>	Calculate the pointwise multiplication $out = x1 * x2$.
<code>_norm(x)</code>	Calculate the norm of x .
<code>contains_all(other)</code>	Test if all points in <code>other</code> are contained in this set.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of $x1$ and $x2$
<code>element([inp])</code>	Create a <code>LinearSpaceVector</code> from <code>inp</code> or from scratch.
<code>inner(x1, x2)</code>	Calculate the inner product of $x1$ and $x2$.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of $x1$ and $x2$.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	A one vector in this space.
<code>zero()</code>	A zero vector in this space.

LinearSpace.__contains__

LinearSpace.__contains__(other)

Return other in self.

Returnscontains : bool

True if other is a `LinearSpaceVector` instance and `other.space` is equal to this space, False otherwise.

Notes

This is the strict default where spaces must be equal. Subclasses may choose to implement a less strict check.

LinearSpace.__eq__

LinearSpace.__eq__(other)

Return self == other.

LinearSpace._dist

LinearSpace._dist(x1, x2)

Calculate the distance between $x1$ and $x2$.

This method is intended to be private, public callers should resort to `dist` which is type-checked.

LinearSpace._inner

LinearSpace._inner(x1, x2)

Calculate the inner product of $x1$ and $x2$.

This method is intended to be private, public callers should resort to `inner` which is type-checked.

LinearSpace._lincomb

LinearSpace._lincomb(a, x1, b, x2, out)

Calculate $out = a*x1 + b*x2$.

This method is intended to be private, public callers should resort to `lincomb` which is type-checked.

LinearSpace._multiply`LinearSpace._multiply(x1, x2, out)`

Calculate the pointwise multiplication $out = x1 * x2$.

This method is intended to be private, public callers should resort to `multiply` which is type-checked.

LinearSpace._norm`LinearSpace._norm(x)`

Calculate the norm of x .

This method is intended to be private, public callers should resort to `norm` which is type-checked.

LinearSpace.contains_all`LinearSpace.contains_all(other)`

Test if all points in `other` are contained in this set.

This is a default implementation and should be overridden by subclasses.

LinearSpace.contains_set`LinearSpace.contains_set(other)`

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

LinearSpace.dist`LinearSpace.dist(x1, x2)`

Calculate the distance between two vectors.

Parameters`x1, x2` : `LinearSpaceVector`

Vectors whose distance to compute

Returns`dist` : `float`

Distance between vectors

LinearSpace.divide`LinearSpace.divide(x1, x2, out=None)`

Calculate the pointwise division of $x1$ and $x2$

Parameters`x1` : `LinearSpaceVector`

The dividend

`x2` : `LinearSpaceVector`

The divisor

out : `LinearSpaceVector`, optional

Vector to write the ratio to

Returns`out` : `LinearSpaceVector`

Ratio of the vectors. If `out` was provided, the returned object is a reference to it.

LinearSpace.element

`LinearSpace.element` (*inp=None, **kwargs*)

Create a *LinearSpaceVector* from *inp* or from scratch.

If called without *inp* argument, an arbitrary element of the space is generated without guarantee of its state.

Parameters*inp* : optional

Input data from which to create the element

kwargs :

Optional further arguments

Return*element* : *LinearSpaceVector*

A vector in this space

LinearSpace.inner

`LinearSpace.inner` (*x1, x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1, x2* : *LinearSpaceVector*

Factors in the inner product

Return*sout* : *LinearSpace.field* element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

LinearSpace.lincomb

`LinearSpace.lincomb` (*a, x1, b=None, x2=None, out=None*)

Linear combination of vectors.

Calculates

$out = a * x1$

or, if *b* and *y* are given,

$out = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply *x2* with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Return*sout* : *LinearSpaceVector*

Result of the linear combination. If `out` was provided, the returned object is a reference to it.

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

$$x = x * (1 + 2 + 3.14)$$

LinearSpace.multiply

`LinearSpace.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

LinearSpace.norm

`LinearSpace.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

LinearSpace.one

`LinearSpace.one()`

A one vector in this space.

The one vector is defined as the multiplicative unit of a space.

Returns`sv` : *LinearSpaceVector*

The one vector of this space

LinearSpace.zero

`LinearSpace.zero()`

A zero vector in this space.

The zero vector is defined as the additive unit of a space.

Returns`sv` : *LinearSpaceVector*

The zero vector of this space

`__init__` (*field*)

Initialize a `LinearSpace`.

This method should be called by all inheriting methods so that the field property of the space is set properly.

Parameters*field* : *Field*

The underlying scalar field of the space

LinearSpaceNotImplementedError

exception `odl.set.space.LinearSpaceNotImplementedError`

Exception for not implemented errors in *LinearSpace*'s.

These are raised when a method in *LinearSpace* that has not been defined in a specific space is called.

LinearSpaceTypeError

exception `odl.set.space.LinearSpaceTypeError`

Exception for type errors in *LinearSpace*'s.

These are raised when the wrong type of element is fed to *LinearSpace.lincomb* and related functions.

LinearSpaceVector

class `odl.set.space.LinearSpaceVector` (*space*)

Bases: object

Abstract *LinearSpace* element.

Not intended for creation of vectors, use the space's *LinearSpace.element* method instead.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (.
<i>space</i>	Space to which this vector belongs.

LinearSpaceVector.T

`LinearSpaceVector.T`

The transpose of a vector, the functional given by (., self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

LinearSpaceVector.space

`LinearSpaceVector.space`

Space to which this vector belongs.

LinearSpace

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>assign(other)</code>	Assign the values of <code>other</code> to <code>self</code> .
<code>copy()</code>	Create an identical (deep) copy of <code>self</code> .
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.

LinearSpaceVector.__eq__

`LinearSpaceVector.__eq__(other)`

Return `self == other`.

Two vectors are equal if their distance is 0

Parameters`other` : *LinearSpaceVector*

Vector in this space.

Return`equals` : `bool`

True if the vectors are equal, else false.

Notes

Equality is very sensitive to numerical errors, thus any operations on a vector should be expected to break equality testing.

Examples

```

>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(1, norm=np.linalg.norm)
>>> x = rn.element([0.1])
>>> x == x
True
>>> y = rn.element([0.1])
>>> x == y
True
>>> z = rn.element([0.3])
>>> x+x+x == z
False

```

LinearSpaceVector.assign

`LinearSpaceVector.assign` (*other*)
Assign the values of *other* to self.

LinearSpaceVector.copy

`LinearSpaceVector.copy` ()
Create an identical (deep) copy of self.

LinearSpaceVector.dist

`LinearSpaceVector.dist` (*other*)
Distance to *other*.

LinearSpace.dist

LinearSpaceVector.divide

`LinearSpaceVector.divide` (*x*, *y*)
Divide by *other* inplace.

LinearSpace.divide

LinearSpaceVector.inner

`LinearSpaceVector.inner` (*other*)
Inner product with *other*.

LinearSpace.inner

LinearSpaceVector.lincomb

`LinearSpaceVector.lincomb` (*a*, *x1*, *b=None*, *x2=None*)
Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

LinearSpaceVector.multiply

`LinearSpaceVector.multiply` (*x*, *y*)
Multiply by *other* inplace.

LinearSpace.multiply

LinearSpaceVector.norm`LinearSpaceVector.norm()`

Norm of vector

*LinearSpace.norm***LinearSpaceVector.set_zero**`LinearSpaceVector.set_zero()`

Set this vector to zero.

LinearSpace.zero`__init__(space)`

Default initializer of vectors.

All deriving classes must call this method to set space.

UniversalSpace`class odl.set.space.UniversalSpace`Bases: *odl.set.space.LinearSpace*

A dummy linear space class.

Mostly raising *LinearSpaceNotImplementedError*.**Attributes**

<i>element_type</i>	<i>LinearSpaceVector</i>
<i>field</i>	The field of this vector space.

UniversalSpace.element_type`UniversalSpace.element_type`*LinearSpaceVector***UniversalSpace.field**`UniversalSpace.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns*field* : *Field*

The underlying field.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>__dist(x1, x2)</code>	Dummy distance method.
<code>__divide(x1, x2, out)</code>	Dummy division method.

Continued on next page

Table 2.133 – continued from previous page

<code>__inner(x1, x2)</code>	Dummy inner product method.
<code>__lincomb(a, x1, b, x2, out)</code>	Dummy linear combination.
<code>__multiply(x1, x2, out)</code>	Dummy multiplication method.
<code>__norm(x)</code>	Dummy norm method.
<code>contains_all(other)</code>	Test if all points in <code>other</code> are contained in this set.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of <code>x1</code> and <code>x2</code>
<code>element([inp])</code>	Dummy element creation method.
<code>inner(x1, x2)</code>	Calculate the inner product of <code>x1</code> and <code>x2</code> .
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of <code>x1</code> and <code>x2</code> .
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	A one vector in this space.
<code>zero()</code>	A zero vector in this space.

UniversalSpace.__contains__

UniversalSpace.__contains__(*other*)

Return `other` in `self`.

Dummy membership check, True for any *LinearSpaceVector*.

UniversalSpace.__eq__

UniversalSpace.__eq__(*other*)

Return `self == other`.

Dummy check, True for any *LinearSpace*.

UniversalSpace._dist

UniversalSpace._dist(*x1*, *x2*)

Dummy distance method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._divide

UniversalSpace._divide(*x1*, *x2*, *out*)

Dummy division method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._inner

UniversalSpace._inner(*x1*, *x2*)

Dummy inner product method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._lincomb

UniversalSpace._lincomb(*a*, *x1*, *b*, *x2*, *out*)

Dummy linear combination.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._multiply

`UniversalSpace._multiply(x1, x2, out)`

Dummy multiplication method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._norm

`UniversalSpace._norm(x)`

Dummy norm method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace.contains_all

`UniversalSpace.contains_all(other)`

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

UniversalSpace.contains_set

`UniversalSpace.contains_set(other)`

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

UniversalSpace.dist

`UniversalSpace.dist(x1, x2)`

Calculate the distance between two vectors.

Parameters*x1, x2* : *LinearSpaceVector*

Vectors whose distance to compute

Returns*dist* : float

Distance between vectors

UniversalSpace.divide

`UniversalSpace.divide(x1, x2, out=None)`

Calculate the pointwise division of *x1* and *x2*

Parameters*x1* : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns*out* : *LinearSpaceVector*

Ratio of the vectors. If *out* was provided, the returned object is a reference to it.

UniversalSpace.element

UniversalSpace.**element** (*inp=None*)

Dummy element creation method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace.inner

UniversalSpace.**inner** (*x1, x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1, x2* : *LinearSpaceVector*

Factors in the inner product

Returns*out* : *LinearSpace.field* element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

UniversalSpace.lincomb

UniversalSpace.**lincomb** (*a, x1, b=None, x2=None, out=None*)

Linear combination of vectors.

Calculates

$out = a * x1$

or, if *b* and *y* are given,

$out = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply *x2* with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns*out* : *LinearSpaceVector*

Result of the linear combination. If *out* was provided, the returned object is a reference to it.

Notes

The vectors *out*, *x1* and *x2* may be aligned, thus a call

`space.lincomb(x, 2, x, 3.14, out=x)`

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

UniversalSpace.multiply

`UniversalSpace.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

UniversalSpace.norm

`UniversalSpace.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

UniversalSpace.one

`UniversalSpace.one()`

A one vector in this space.

The one vector is defined as the multiplicative unit of a space.

Returns`sv` : *LinearSpaceVector*

The one vector of this space

UniversalSpace.zero

`UniversalSpace.zero()`

A zero vector in this space.

The zero vector is defined as the additive unit of a space.

Returns`sv` : *LinearSpaceVector*

The zero vector of this space

`__init__()`

Initialize a universal space

2.8.5 solvers

Modules

advanced

Modules

chambolle_pock

First-order primal-dual algorithm developed by Chambolle and Pock.

The Chambolle-Pock algorithm is a flexible method well suited for non-smooth convex optimization problems in imaging. It was first proposed in [\[CP2011a\]](#).

Functions

`chambolle_pock_solver`(*op*, *x*, *tau*, *sigma*, ...) Chambolle-Pock algorithm for non-smooth convex optimization problems.

chambolle_pock_solver

`odl.solvers.advanced.chambolle_pock.chambolle_pock_solver`(*op*, *x*, *tau*, *sigma*,
proximal_primal, *proximal_dual*, *niter=1*,
***kwargs*)

Chambolle-Pock algorithm for non-smooth convex optimization problems.

First order primal-dual hybrid-gradient method for non-smooth convex optimization problems with known saddle-point structure. The primal formulation of the general problem is:

$$\min_{\{x \in X\}} F(Kx) + G(x)$$

where X and Y are finite-dimensional Hilbert spaces, K is a linear operator $K : X \rightarrow Y$. and $G : X \rightarrow [0, +\infty]$ and $F : Y \rightarrow [0, +\infty]$ are proper, convex, lower-semicontinuous functionals.

The Chambolle-Pock algorithm basically consists of alternating a gradient ascent in the dual variable y and a gradient descent in the primal variable x . The proximal operator is used to generate a ascent direction for the convex conjugate of F and descent direction for G . Additionally an over-relaxation in the primal variable is performed.

Parameters*op* : *Operator*

A (product space) operator between Hilbert spaces with domain X and range Y

x : element in the domain of *op*

Starting point of the iteration

tau : positive float

Step size parameter for the update of the primal variable x . Controls the extent to which *proximal_primal* maps points towards the minimum of G .

sigma : positive float

Step size parameter for the update of the dual variable y . Controls the extent to which *proximal_dual* maps points towards the minimum of F_{cc} .

proximal_primal : callable

Evaluated at `tau`, the function returns the proximal operator, `prox_tau[G](x)`, of the functional `G`. The domain of `G` and its proximal operator instance are the space, `X`, of the primal variable `x` i.e. the domain of `op`.

proximal_dual : callable

Evaluated at `sigma`, the function returns the proximal operator, `prox_sigma[F_cc](x)`, of the convex conjugate, `F_cc`, of the function `F`. The domain of `F_cc` and its proximal operator instance are the space, `Y`, of the dual variable `y` i.e. the range of `op`.

niter : non-negative int, optional

Number of iterations

Other Parameters**theta** : float in [0, 1], optional

Relaxation parameter. Default: 1

gamma : non-negative float, optional

Acceleration parameter. If not `None` overwrites `theta` and uses variable relaxation parameter and step sizes with `tau` and `sigma` as initial values. Requires `G` or `F_cc` to be uniformly convex. Default: `None`

partial : *Partial*, optional

If not `None` the *Partial* instance(s) are executed in each iteration, e.g. plotting each iterate. Default: `None`

x_relax : element in the domain of `op`, optional

Required to resume iteration. If `None` it is a copy of the primal variable `x`. Default: `None`

y : element in the range of `op`, optional

Required to resume iteration. If `None` it is set to a zero element in `Y` which is the range of `op`. Default: `None`

Notes

For a more detailed documentation see `chambolle_pock`.

For references on the Chambolle-Pock algorithm see [\[CP2011a\]](#) and [\[CP2011b\]](#).

This implementation of the CP algorithm is along the lines of [\[Sid+2012\]](#).

For more on convex analysis including convex conjugates and resolvent operators see [\[Roc1970\]](#).

For more on proximal operators and algorithms see [\[PB2014\]](#).

proximal_operators

Factory functions for creating proximal operators.

For more details see `proximal_operators` and references therein. For more details on proximal operators including how to evaluate the proximal operator of a variety of functions see [\[PB2014\]](#).

Functions

<code>combine_proximals(factory_list)</code>	Combine proximal operators into a diagonal product space operator.
<code>proximal_convexconjugate_l1(space[, lam, g])</code>	Proximal operator factory of the convex conjugate of the l1-semi-norm.
<code>proximal_convexconjugate_l2(space[, lam, g])</code>	Proximal operator factory of the convex conjugate of the l2-norm.
<code>proximal_nonnegativity(space)</code>	Function to create the proximal operator of $G(x) = \text{ind}(x > 0)$.
<code>proximal_zero(space)</code>	Function to create the proximal operator of $G(x) = 0$.

combine_proximals

`odl.solvers.advanced.proximal_operators.combine_proximals(factory_list)`

Combine proximal operators into a diagonal product space operator.

This assumes the functional to be separable across variables in order to make use of the separable sum property of proximal operators.

$$\text{prox_tau}[f(x) + g(y)](x, y) = (\text{prox_tau}[f](x), \text{prox_tau}[g](y))$$

Parameters`factory_list` : list of *Operator*

A list containing proximal operators which are created by the corresponding factory functions

Returns`diag_op` : *Operator*

Returns a diagonal product space operator to be initialized with the same step size parameter

proximal_convexconjugate_l1

`odl.solvers.advanced.proximal_operators.proximal_convexconjugate_l1(space, lam=1, g=None)`

Proximal operator factory of the convex conjugate of the l1-semi-norm.

Function for the proximal operator of the convex conjugate of the functional F where F is an l1-semi-norm

$$F(x) = \text{lam} \parallel x - g \parallel_p$$

with x and g elements in `space`, scaling factor `lam`, and point-wise magnitude $\parallel x \parallel_p$ of x . If x is vector-valued, $\parallel x \parallel_p$ is the point-wise l2-norm across the vector components.

The convex conjugate, F_{cc} , of F is given by the indicator function of the set `box(lam)`

$$F_{cc}(y) = \text{lam} \text{ind}_{\{\text{box}(\text{lam})\}}(\parallel y \parallel_p + \langle y / \text{lam} \parallel_p, g \rangle)$$

where `box(lam)` is a hypercube centered at the origin with width 2 lam .

The proximal operator of F_{cc} is

$$\text{prox_sigma}[F_{cc}](y) = \text{lam} (y - \text{sigma } g) / (\max(\text{lam } 1_{\{\parallel y \parallel_p\}}, \parallel y - \text{sigma } g \parallel_p))$$

where $\max(.,.)$ thresholds the lower bound of $\parallel y \parallel_p$ point-wise and $1_{\{\parallel y \parallel_p\}}$ is a unit vector in the space of $\parallel y \parallel_p$.

Parameters`space` : *DiscreteLp* or *ProductSpace* of *DiscreteLp* spaces

Domain of the functional F

`g` : *DiscreteLpVector*

An element in `space`

`lam` : positive float

Scaling factor or regularization parameter

Returns`prox` : *Operator*

Returns the proximal operator to be initialized

proximal_convexconjugate_l2

`odl.solvers.advanced.proximal_operators.proximal_convexconjugate_l2(space, lam=1, g=None)`

Proximal operator factory of the convex conjugate of the l2-norm.

Function for the proximal operator of the convex conjugate of the functional F where F is the l2-norm

$$F(x) = \text{lam} \, 1/2 \|x - g\|_2^2$$

with x and g elements in `space`, scaling factor `lam`, and given data `g`.

The convex conjugate, `F_cc`, of F is given by

$$F_{cc}(y) = 1/\text{lam} \, (1/2 \|y/\text{lam}\|_2^2 + \langle y/\text{lam}, g \rangle)$$

The proximal operator of `F_cc` is given by

$$\text{prox_sigma}[F_{cc}](y) = (y - \text{sigma} \, g) / (1 + \text{sigma}/\text{lam})$$

Parameters`space` : *DiscreteLp* or *ProductSpace* of *DiscreteLp*

Domain of `F(x)`

`g` : *DiscreteLpVector*

An element in `space`

lam : positive float

Scaling factor or regularization parameter

Returns`prox` : *Operator*

Returns the proximal operator to be initialized

proximal_nonnegativity

`odl.solvers.advanced.proximal_operators.proximal_nonnegativity(space)`

Function to create the proximal operator of $G(x) = \text{ind}(x > 0)$.

Function for the proximal operator of the functional $G(x) = \text{ind}(x > 0)$ to be initialized.

If `P` is the set of non-negative elements, the indicator function of which is defined as

$$\text{ind}(x > 0) = \{0 \text{ if } x \text{ in } P, \text{infinity if } x \text{ is not in } P\}$$

with `x` being an element in `space`.

The proximal operator of `G` is the point-wise non-negativity thresholding of `x`

$$\text{prox_tau}[G](x) = \{x \text{ if } x > 0, 0 \text{ if } \leq 0\}$$

It is independent of `tau` and invariant under a positive rescaling of `G` which leaves the indicator function as it stands.

Parameters`space` : *DiscreteLp* or *ProductSpace* of *DiscreteLp*

Domain of the functional `G(x)`

Returns`prox` : *Operator*

Returns the proximal operator to be initialized

proximal_zero`odl.solvers.advanced.proximal_operators.proximal_zero(space)`

Function to create the proximal operator of $G(x) = 0$.

Function to initialize the proximal operator of $G(x) = 0$ where x is an element in `space`. The proximal operator of this functional is the identity operator

$$\text{prox_tau}[G](x) = x$$

It is independent of `tau`.

Parameters`space` : *DiscreteLp* or *ProductSpace* of *DiscreteLp* spaces

Domain of the functional G

Returns`prox` : *Operator*

Returns the proximal operator to be initialized

findroot**Modules****newton**

(Quasi-)Newton schemes to find zeros of functions (gradients).

Functions

<code>bfgs_method(grad, x, line_search[, niter, ...])</code>	Quasi-Newton BFGS method to minimize a differentiable function.
<code>broydens_first_method(grad, x, line_search)</code>	Broyden's first method, a quasi-Newton scheme.
<code>broydens_second_method(grad, x, line_search)</code>	Broyden's first method, a quasi-Newton scheme.

bfgs_method`odl.solvers.findroot.newton.bfgs_method(grad, x, line_search, niter=1, partial=None)`

Quasi-Newton BFGS method to minimize a differentiable function.

This is a general and optimized implementation of a quasi-Newton method with BFGS update for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \rightarrow \mathbb{R}$ on a Hilbert space \mathcal{X} . It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \rightarrow \mathcal{X}.$$

The QN method is an approximate Newton method, where the Hessian is approximated and gradually updated in each step. This implementation uses the rank-one BFGS update schema where the inverse of the Hessian is recalculated in each iteration.

The algorithm is described in [GNS2009], Section 12.3 and in the [BFGS Wikipedia article](#)

Parameters`grad` : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

`x` : *element* of the domain of `grad`

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : int, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns None

broydens_first_method

`odl.solvers.findroot.newton.broydens_first_method(grad, x, line_search, niter=1, partial=None)`

Broyden's first method, a quasi-Newton scheme.

This is a general and optimized implementation of Broyden's first (or 'good') method, a quasi-Newton method for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \rightarrow \mathbb{R}$ on a Hilbert space \mathcal{X} . It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \rightarrow \mathcal{X}$$

using a Newton-type update scheme with approximate Hessian.

The algorithm is described in [Bro1965] and [Kva1991], and in a [Wikipedia article](#).

Parameters **grad** : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

x : *element* of the domain of **grad**

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : int, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns None

broydens_second_method

`odl.solvers.findroot.newton.broydens_second_method(grad, x, line_search, niter=1, partial=None)`

Broyden's first method, a quasi-Newton scheme.

This is a general and optimized implementation of Broyden's second (or 'bad') method, a quasi-Newton method for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \rightarrow \mathbb{R}$ on a Hilbert space \mathcal{X} . It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \rightarrow \mathcal{X}$$

using a Newton-type update scheme with approximate Hessian.

The algorithm is described in [\[Bro1965\]](#) and [\[Kva1991\]](#), and in a [Wikipedia article](#)

Parameters**grad** : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

x : *element* of the domain of **grad**

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : *int*, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

iterative

Modules

iterative

Simple iterative type optimization schemes.

Functions

<code>conjugate_gradient</code> (op, x, rhs[, niter, partial])	Optimized implementation of CG for self-adjoint operators.
<code>conjugate_gradient_normal</code> (op, x, rhs[, ...])	Optimized implementation of CG for the normal equation.
<code>exp_zero_seq</code> (base)	The default exponential zero sequence.
<code>gauss_newton</code> (op, x, rhs[, niter, zero_seq, ...])	Optimized implementation of a Gauss-Newton method.
<code>landweber</code> (op, x, rhs[, niter, omega, ...])	Optimized implementation of Landweber's method.

conjugate_gradient

`odl.solvers.iterative.iterative.conjugate_gradient` (op, x, rhs, niter=1, partial=None)

Optimized implementation of CG for self-adjoint operators.

This method solves the inverse problem (of the first kind)

$$Ax = y$$

for a linear and self-adjoint *Operator* A.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is described (for linear systems) in a [Wikipedia article](#).

Parameters**op** : linear *Operator*

Operator in the inverse problem. It must be linear and self-adjoint. This implies in particular that its domain and range are equal.

x : *element* of the domain of **op**

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : *element* of the range of *op*

Right-hand side of the equation defining the inverse problem

niter : int, optional

Maximum number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

See also:

*conjugate_gradient_normal*Solver for nonsymmetric matrices

conjugate_gradient_normal

`odl.solvers.iterative.iterative.conjugate_gradient_normal` (*op*, *x*, *rhs*, *niter=1*, *partial=None*)

Optimized implementation of CG for the normal equation.

This method solves the normal equation

$$A^*Ax = A^*y$$

to the inverse problem (of the first kind)

$$Ax = y$$

with a linear *Operator* *A*.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is described (for linear systems) in a [Wikipedia article](#).

Parameters*op* : *Operator*

Operator in the inverse problem. If not linear, it must have an implementation of *Operator.derivative*, which in turn must implement *Operator.adjoint*, i.e. the call `op.derivative(x).adjoint` must be valid.

x : *element* of the domain of *op*

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : *element* of the range of *op*

Right-hand side of the equation defining the inverse problem

niter : int, optional

Maximum number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

See also:

conjugate_gradient Optimized solver for symmetric matrices

exp_zero_seq

`odl.solvers.iterative.iterative.exp_zero_seq(base)`

The default exponential zero sequence.

It is defined by

$$t_0 = 1.0 \quad t_m = t_{(m-1)} / \text{base}$$

or, in closed form

$$t_m = \text{base}^{-(m-1)}$$

Parameters`base` : float

Base of the sequence. Its absolute value must be larger than 1.

Yields`val` : float

The next value in the exponential sequence

gauss_newton

`odl.solvers.iterative.iterative.gauss_newton(op, x, rhs, niter=1, zero_seq=<generator object exp_zero_seq>, partial=None)`

Optimized implementation of a Gauss-Newton method.

This method solves the inverse problem (of the first kind)

$$A(x) = y$$

for a (Frechet-) differentiable *Operator* `A` using a Gauss-Newton iteration.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

A variant of the method applied to a specific problem is described in a [Wikipedia article](#).

Parameters`op` : *Operator*

Operator in the inverse problem. If not linear, it must have an implementation of *Operator.derivative*, which in turn must implement *Operator.adjoint*, i.e. the call `op.derivative(x).adjoint` must be valid.

`x` : *element* of the domain of `op`

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

`rhs` : *element* of the range of `op`

Right-hand side of the equation defining the inverse problem

`niter` : int, optional

Maximum number of iterations

`zero_seq` : iterable, optional

Zero sequence whose values are used for the regularization of the linearized problem in each Newton step

`partial` : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

landweber

`odl.solvers.iterative.iterative.landweber` (*op*, *x*, *rhs*, *niter*=1, *omega*=1, *projection*=None, *partial*=None)

Optimized implementation of Landweber's method.

This method calculates an approximate least-squares solution of the inverse problem of the first kind

$$\mathcal{A}(x) = y,$$

for a given $y \in \mathcal{Y}$, i.e. an approximate solution x^* to

$$\min_{x \in \mathcal{X}} \|\mathcal{A}(x) - y\|_{\mathcal{Y}}^2$$

for a (Frechet-) differentiable operator $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Y}$ between Hilbert spaces \mathcal{X} and \mathcal{Y} . The method starts from an initial guess x_0 and uses the iteration

$$x_{k+1} = x_k - \omega \partial \mathcal{A}(x)^*(\mathcal{A}(x_k) - y),$$

where $\partial \mathcal{A}(x)$ is the Frechet derivativ of \mathcal{A} at x and ω is a relaxation parameter. For linear problems, a choice $0 < \omega < 2/\|\mathcal{A}\|$ guarantees convergence, where $\|\mathcal{A}\|$ stands for the operator norm of \mathcal{A} .

Users may also optionally provide a projection to project each iterate onto some subset. For example enforcing positivity.

This implementation uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is also described in a [Wikipedia article](#).

Parameters*op* : *Operator*

Operator in the inverse problem. It must have a *Operator.derivative* property, which returns a new operator which in turn has an *Operator.adjoint* property, i.e. *op.derivative(x).adjoint* must be well-defined for *x* in the operator domain.

x : *element* of the domain of *op*

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : *element* of the range of *op*

Right-hand side of the equation defining the inverse problem

niter : int, optional

Maximum number of iterations

omega : positive float, optional

Relaxation parameter in the iteration

projection : callable, optional

Function that can be used to modify the iterates in each iteration, for example enforcing positivity. The function should take one argument and modify it in place.

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

linear

scalar

Gradient-based optimization schemes.

Modules

gradient

Gradient-based optimization schemes.

Functions

<code>steepest_descent</code>	<code>(grad, x[, niter, ...])</code>	Steepest descent method to minimize an objective function.
-------------------------------	--------------------------------------	--

steepest_descent

`odl.solvers.scalar.gradient.steepest_descent` (*grad*, *x*, *niter*=1, *line_search*=1, *projection*=None, *partial*=None)

Steepest descent method to minimize an objective function.

General implementation of steepest decent (also known as gradient decent) for solving

$$\min f(x)$$

The algorithm is intended for unconstrained problems. It needs line search in order guarantee convergence. With appropriate line search, it can also be used for constrained problems where one wants to minimize over some given set C . This can be done by defining $f(x) = \infty$ for $x \notin C$, or by providing a `projection` function that projects the iterates on C .

The algorithm is described in [BV2004], section 9.3–9.4 ([book available online](#)), [GNS2009], Section 12.2, and wikipedia [Gradient_descent](#).

Parameters`grad` : *Operator*

Gradient of the objective function, $x \mapsto \nabla f(x)$

x : *element* of the domain of `deriv`

Starting point of the iteration

niter : `int`, optional

Number of iterations

line_search : `float` or *LineSearch*, optional

Strategy to choose the step length. If a float is given, uses it as a fixed step length.

projection : *callable*, optional

Function that can be used to modify the iterates in each iteration, for example enforcing positivity. The function should take one argument and modify it inplace.

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

See also:

`odl.solvers.iterative.iterative.landweber` Optimized solver for the case $f(x) = \|Ax - b\|_2^2$

`odl.solvers.iterative.iterative.conjugate_gradient` Optimized solver for the case $f(x) = x^T Ax - 2 x^T b$

steplen

Step length computation for optimization schemes.

Classes

<code>BacktrackingLineSearch(function[, tau, c, ...])</code>	Backtracking line search for step length calculation.
<code>BarzilaiBorweinStep(gradf[, step0])</code>	Barzilai-Borwein method to compute a step length.
<code>ConstantLineSearch(constant)</code>	Line search object that returns a constant step length.
<code>LineSearch</code>	Abstract base class for line search step length methods.
<code>StepLength</code>	Abstract base class for step length methods.

BacktrackingLineSearch

class `odl.solvers.scalar.steplen.BacktrackingLineSearch` (*function*, *tau*=0.5, *c*=0.01, *max_num_iter*=None)

Bases: `odl.solvers.scalar.steplen.LineSearch`

Backtracking line search for step length calculation.

This methods approximately finds the longest step length fulfilling the Armijo-Goldstein condition.

The line search algorithm is described in [BV2004], page 464 (book available online) and [GNS2009], pages 378–379. See also `Backtracking_line_search`.

Methods

<code>__call__(x, direction, dir_derivative)</code>	Calculate the optimal step length along a line.
<code>__eq__</code>	Return self==value.

BacktrackingLineSearch.__call__

`BacktrackingLineSearch.__call__(x, direction, dir_derivative)`

Calculate the optimal step length along a line.

Parameters*x* : *Operator.domain element*

The current point

direction : *Operator.domain element*

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returns*step* : float

The computed step length

__init__ (*function*, *tau*=0.5, *c*=0.01, *max_num_iter*=None)

Initialize a new instance.

Parameters*function* : callable

The cost function of the optimization problem to be solved.

tau : float, optional

The amount the step length is decreased in each iteration, as long as it does not fulfill the decrease condition. The step length is updated as `step_length *= tau`

c : float, optional

The ‘discount factor’ on the `step_length * direction derivative`, which the new point needs to be smaller than in order to fulfill the condition and be accepted (see the references).

max_num_iter : int, optional

Maximum number of iterations allowed each time the line search method is called. If not set, this number is calculated to allow a shortest step length of 0.0001.

BarzilaiBorweinStep

class odl.solvers.scalar.steplen.**BarzilaiBorweinStep** (*gradf*, *step0*=0.0005)

Bases: object

Barzilai-Borwein method to compute a step length.

Barzilai-Borwein method to compute a step length for gradient descent methods.

The method is described in [\[BB1988\]](#) and [\[Ray1997\]](#).

Methods

__call__ (<i>x</i> , <i>x0</i>)	Calculate the step length at a point.
__eq__	Return self==value.

BarzilaiBorweinStep.__call__

BarzilaiBorweinStep.**__call__** (*x*, *x0*)

Calculate the step length at a point.

Parameters*xx* : *Operator.domain element*

The current point

x0 : *Operator.domain element*

The previous point

Returns*step* : float

The step length

__init__ (*gradf*, *step0*=0.0005)

Initialize a new instance.

Parameters*gradf* : *Operator*

The gradient of the objective function at a point

step0 : float, optional

Initial step length parameter

ConstantLineSearch

class odl.solvers.scalar.steplen.**ConstantLineSearch** (*constant*)

Bases: odl.solvers.scalar.steplen.LineSearch

Line search object that returns a constant step length.

Methods

<code>__call__(x, direction, dir_derivative)</code>	Calculate the step length at a point.
<code>__eq__</code>	Return self==value.

ConstantLineSearch.__call__

ConstantLineSearch.**__call__** (*x, direction, dir_derivative*)

Calculate the step length at a point.

Parameters*x* : *Operator.domain element*

The current point

direction : *Operator.domain element*

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returns*step* : float

The constant step length

__init__ (*constant*)

Initialize a new instance.

Parameters*constant* : float

The constant step length

LineSearch

class odl.solvers.scalar.steplen.**LineSearch**

Bases: object

Abstract base class for line search step length methods.

Methods

<code>__call__(x, direction, dir_derivative)</code>	Calculate step length in direction.
<code>__eq__</code>	Return self==value.

LineSearch.__call__

LineSearch.**__call__** (*x, direction, dir_derivative*)

Calculate step length in direction.

Parameters*x* : *Operator.domain element*

The current point

direction : *Operator.domain element*

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : float

The step length

StepLength

class odl.solvers.scalar.steplen.**StepLength**

Bases: object

Abstract base class for step length methods.

Methods

<code>__call__(x, direction, dir_derivative)</code>	Calculate the step length at a point.
<code>__eq__</code>	Return self==value.

StepLength.__call__

StepLength.**__call__**(x, direction, dir_derivative)

Calculate the step length at a point.

Parametersx : *Operator.domain element*

The current point

direction : *Operator.domain element*

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : float

The step length

util

Modules

partial

Partial objects for per-iterate actions in iterative methods.

Classes

<code>AndPartial(*partials)</code>	Partial used for combining several partials
<code>ForEachPartial(function)</code>	Simple object for applying a function to each iterate.
<code>Partial</code>	Abstract base class for sending partial results of iterations.
<code>PrintIterationPartial([text])</code>	Print the iteration count.
<code>PrintNormPartial()</code>	Print the current norm.
<code>PrintTimingPartial()</code>	Print the time elapsed since the previous iteration.
<code>ShowPartial(**kwargs)</code>	Show the partial result.
<code>StorePartial([results])</code>	Simple object for storing all partial results of the solvers.

AndPartial

class odl.solvers.util.partial.**AndPartial** (**partials*)

Bases: `odl.solvers.util.partial.Partial`

Partial used for combining several partials

Methods

<code>__call__(result)</code>	Apply all partials to result.
<code>__eq__</code>	Return self==value.

AndPartial.__call__

`AndPartial.__call__ (result)`

Apply all partials to result.

`__init__ (*partials)`

Initialize an instance.

Parameters**partials* : `Partial`'s

Partials to be called in sequence as listed.

ForEachPartial

class odl.solvers.util.partial.**ForEachPartial** (*function*)

Bases: `odl.solvers.util.partial.Partial`

Simple object for applying a function to each iterate.

Methods

<code>__call__(result)</code>	Apply function to result.
<code>__eq__</code>	Return self==value.

ForEachPartial.__call__

`ForEachPartial.__call__ (result)`

Apply function to result.

`__init__ (function)`

Initialize an instance.

Parameters*function* : callable

Function to call for each iteration

Partial**class** odl.solvers.util.partial.**Partial**

Bases: object

Abstract base class for sending partial results of iterations.

Methods

<code>__call__(result)</code>	Apply the partial object to result.
<code>__eq__</code>	Return self==value.

Partial.__call__

Partial.__call__(result)

Apply the partial object to result.

Parametersresult : *LinearSpaceVector*

Partial result after n iterations

ReturnsNone**PrintIterationPartial****class** odl.solvers.util.partial.**PrintIterationPartial** (text=None)Bases: *odl.solvers.util.partial.Partial*

Print the iteration count.

Methods

<code>__call__()</code>	Print the current iteration.
<code>__eq__</code>	Return self==value.

PrintIterationPartial.__call__

PrintIterationPartial.__call__()

Print the current iteration.

`__init__(text=None)`

Initialize an instance.

Parametertext : str

Text to display before the iteration count. Default: 'iter ='

PrintNormPartial**class** odl.solvers.util.partial.**PrintNormPartial**Bases: *odl.solvers.util.partial.Partial*

Print the current norm.

Methods

<code>__call__(result)</code>	Print the current norm.
<code>__eq__</code>	Return self==value.

PrintNormPartial.__call__`PrintNormPartial.__call__(result)`

Print the current norm.

`__init__()`

Initialize an instance.

PrintTimingPartial**class** `odl.solvers.util.partial.PrintTimingPartial`Bases: `odl.solvers.util.partial.Partial`

Print the time elapsed since the previous iteration.

Methods

<code>__call__()</code>	Print time elapsed from the previous iteration.
<code>__eq__</code>	Return self==value.

PrintTimingPartial.__call__`PrintTimingPartial.__call__()`

Print time elapsed from the previous iteration.

`__init__()`

Initialize an instance.

ShowPartial**class** `odl.solvers.util.partial.ShowPartial(**kwargs)`Bases: `odl.solvers.util.partial.Partial`

Show the partial result.

Methods

<code>__call__(x)</code>	Show the current iterate.
<code>__eq__</code>	Return self==value.

ShowPartial.__call__`ShowPartial.__call__(x)`

Show the current iterate.

`__init__(**kwargs)`

Initialize a new instance.

Parameters are passed through to the vectors show method. Additional parameters included.

Parametersdisplay_step : positive `int`, optional

Number of iterations between plots. Default: 1

Other Parameterskwargs :

Optional arguments passed on to `x.show`

StorePartial

class `odl.solvers.util.partial.StorePartial` (*results=None*)

Bases: `odl.solvers.util.partial.Partial`

Simple object for storing all partial results of the solvers.

Attributes

<code>results</code>	The partial results.
----------------------	----------------------

StorePartial.results

`StorePartial.results`

The partial results.

Methods

<code>__call__(result)</code>	Append result to results list.
<code>__eq__</code>	Return self==value.
<code>__getitem__(index)</code>	Get partial result.

StorePartial.__call__

`StorePartial.__call__(result)`

Append result to results list.

StorePartial.__getitem__

`StorePartial.__getitem__(index)`

Get partial result.

`__init__(results=None)`

Initialize an instance.

Parameters`results`: list

List in which to store the partial results. Default: new list (`[]`)

vector

Modules

newton

Newton type optimization schemes.

Functions

`newtons_method`(`op`, `x`, `line_search`[, ...]) Newton's method for solving a system of equations.

newtons_method

`odl.solvers.vector.newton.newtons_method`(`op`, `x`, `line_search`, `num_iter=10`, `cg_iter=None`, `partial=None`)

Newton's method for solving a system of equations.

This is a general and optimized implementation of Newton's method for solving the problem:

$f(x) = 0$

of finding a root of a function.

The algorithm is well-known and there is a vast literature about it. Among others, the method is described in [BV2004], Sections 9.5 and 10.2 (book available online), [GNS2009], Section 2.7 for solving nonlinear equations and Section 11.3 for its use in minimization, and wikipedia on [Newton's_method](#).

Parameters `op` : *Operator*

Gradient of the objective function, `x --> grad f(x)`

`x` : element in the domain of `op`

Starting point of the iteration

`line_search` : *LineSearch*

Strategy to choose the step length

`num_iter` : `int`, optional

Number of iterations

`cg_iter` : `int`, optional

Number of iterations in the the conjugate gradient solver, for computing the search direction.

`partial` : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Notes

The algorithm works by iteratively solving

$$\partial f(x_k)p_k = -f(x_k)$$

and then updating as

$$x_{k+1} = x_k + \alpha x_k,$$

where α is a suitable step length (see the references). In this implementation the system of equations are solved using the conjugate gradient method.

2.8.6 space

Concrete vector spaces.

Modules

base_ntuples

Base classes for implementation of n-tuples.

Classes

<code>FnBase(size, dtype)</code>	Base class for F^n independent of implementation.
<code>FnBaseVector(space, *args, **kwargs)</code>	Abstract class for representation of <code>FnBase</code> vectors.
<code>FnWeightingBase(impl[, exponent, ...])</code>	Abstract base class for weighting of <code>FnBase</code> spaces.
<code>NtuplesBase(size, dtype)</code>	Base class for sets of n-tuples independent of implementation.
<code>NtuplesBaseVector(space, *args, **kwargs)</code>	Abstract class for representation of <code>NtuplesBase</code> elements.

FnBase

class `odl.space.base_ntuples.FnBase` (*size, dtype*)

Bases: `odl.space.base_ntuples.NtuplesBase`, `odl.set.space.LinearSpace`

Base class for F^n independent of implementation.

Attributes

<code>dtype</code>	The data type of each entry.
<code>element_type</code>	<code>FnBaseVector</code>
<code>field</code>	The field of this vector space.
<code>is_cn</code>	Return <code>True</code> if the space represents \mathbb{C}^n , i.e.
<code>is_rn</code>	Return <code>True</code> if the space represents \mathbb{R}^n , i.e.
<code>shape</code>	The shape of this space.
<code>size</code>	The number of entries per tuple.

FnBase.dtype

`FnBase.dtype`

The data type of each entry.

FnBase.element_type

`FnBase.element_type`

`FnBaseVector`

FnBase.field

`FnBase.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns`field` : `Field`

The underlying field.

FnBase.is_cnFnBase.**is_cn**Return True if the space represents C^n , i.e. complex tuples.**FnBase.is_rn**FnBase.**is_rn**Return True if the space represents R^n , i.e. real tuples.**FnBase.shape**FnBase.**shape**

The shape of this space.

FnBase.sizeFnBase.**size**

The number of entries per tuple.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__dist(x1, x2)</code>	Calculate the distance between x1 and x2.
<code>__divide(x1, x2, out)</code>	The entry-wise division of two vectors, assigned to out.
<code>__inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>__lincomb(a, x1, b, x2, out)</code>	Calculate <code>out = a*x1 + b*x2</code> .
<code>__multiply(x1, x2, out)</code>	The entry-wise product of two vectors, assigned to out.
<code>__norm(x)</code>	Calculate the norm of x.
<code>astype(dtype)</code>	Return a copy of this space with new dtype.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp])</code>	Create a <i>LinearSpaceVector</i> from inp or from scratch.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create a vector of ones.
<code>zero()</code>	Create a vector of zeros.

FnBase.__contains__FnBase.**__contains__**(other)

Return other in self.

Returnscontains : bool

True if other is an *NtuplesBaseVector* instance and other.space is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

FnBase.__eq__

FnBase.__eq__(other)

Return self == other.

Returnsequals: bool

True if other is an instance of this space's type with the same *size* and *dtype*, otherwise False.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

FnBase._dist

FnBase._dist(x1, x2)

Calculate the distance between x1 and x2.

This method is intended to be private, public callers should resort to *dist* which is type-checked.

FnBase._divide

FnBase._divide(x1, x2, out)

The entry-wise division of two vectors, assigned to out.

FnBase._inner`FnBase._inner(x1, x2)`Calculate the inner product of `x1` and `x2`.This method is intended to be private, public callers should resort to `inner` which is type-checked.**FnBase._lincomb**`FnBase._lincomb(a, x1, b, x2, out)`Calculate `out = a*x1 + b*x2`.This method is intended to be private, public callers should resort to `lincomb` which is type-checked.**FnBase._multiply**`FnBase._multiply(x1, x2, out)`The entry-wise product of two vectors, assigned to `out`.**FnBase._norm**`FnBase._norm(x)`Calculate the norm of `x`.This method is intended to be private, public callers should resort to `norm` which is type-checked.**FnBase.astype**`FnBase.astype(dtype)`Return a copy of this space with new `dtype`.**Parametersdtype :**Data type of the returned space. Can be given in any way `numpy.dtype` understands, e.g. as string ('complex64') or data type (`complex`).**Returnsnewspace : *FnBase***

The version of this space with given data type

FnBase.contains_all`FnBase.contains_all(other)`Test if all points in `other` are contained in this set.

This is a default implementation and should be overridden by subclasses.

FnBase.contains_set`FnBase.contains_set(other)`Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

FnBase.dist`FnBase.dist(x1, x2)`

Calculate the distance between two vectors.

Parametersx1, x2 : *LinearSpaceVector*

Vectors whose distance to compute

Returns`dist` : `float`

Distance between vectors

FnBase.divide

`FnBase.divide` (`x1`, `x2`, `out=None`)

Calculate the pointwise division of `x1` and `x2`

Parameters`x1` : `LinearSpaceVector`

The dividend

`x2` : `LinearSpaceVector`

The divisor

out : `LinearSpaceVector`, optional

Vector to write the ratio to

Returns`out` : `LinearSpaceVector`

Ratio of the vectors. If `out` was provided, the returned object is a reference to it.

FnBase.element

`FnBase.element` (`inp=None`, `**kwargs`)

Create a `LinearSpaceVector` from `inp` or from scratch.

If called without `inp` argument, an arbitrary element of the space is generated without guarantee of its state.

Parameters`inp` : optional

Input data from which to create the element

kwargs :

Optional further arguments

Return`element` : `LinearSpaceVector`

A vector in this space

FnBase.inner

`FnBase.inner` (`x1`, `x2`)

Calculate the inner product of `x1` and `x2`.

Parameters`x1`, `x2` : `LinearSpaceVector`

Factors in the inner product

Returns`out` : `LinearSpace.field` element

Product of the vectors. If `out` was provided, the returned object is a reference to it.

FnBase.lincomb

`FnBase.lincomb` (`a`, `x1`, `b=None`, `x2=None`, `out=None`)

Linear combination of vectors.

Calculates

`out = a * x1`

or, if `b` and `y` are given,

```
out = a*x1 + b*x2
```

with error checking of types.

Parameters`a` : Scalar in the field of this space

Scalar to multiply `x1` with.

`x1` : *LinearSpaceVector*

The first of the summands

`b` : Scalar, optional

Scalar to multiply `x2` with.

`x2` : *LinearSpaceVector*, optional

The second of the summands

`out` : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns`out` : *LinearSpaceVector*

Result of the linear combination. If `out` was provided, the returned object is a reference to it.

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

FnBase.multiply

`FnBase.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

`out` : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

FnBase.norm

`FnBase.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out : float`

Norm of the vector

FnBase.one

`FnBase.one()`

Create a vector of ones.

FnBase.zero

`FnBase.zero()`

Create a vector of zeros.

`__init__(size, dtype)`

Initialize a new instance.

Parameters`size : int`

The number of dimensions of the space

dtype : object

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string. Only scalar data types (numbers) are allowed.

FnBaseVector

class `odl.space.base_ntuples.FnBaseVector(space, *args, **kwargs)`

Bases: `odl.space.base_ntuples.NtuplesBaseVector`, `odl.set.space.LinearSpaceVector`

Abstract class for representation of *FnBase* vectors.

Defines abstract attributes and concrete ones which are independent of data representation.

Attributes

<code>T</code>	The transpose of a vector, the functional given by <code>(.</code>
<code>dtype</code>	Length of this vector, equal to space size.
<code>itemsizes</code>	The size in bytes on one element of this type.
<code>nbytes</code>	The number of bytes this vector uses in memory.
<code>ndim</code>	Number of dimensions, always 1.
<code>shape</code>	Number of entries per axis, equals <code>(size,)</code> for linear storage.
<code>size</code>	Length of this vector, equal to space size.
<code>space</code>	Space to which this vector.
<code>ufunc</code>	<i>NtuplesBaseUFuncs</i> , access to numpy style ufuncs.

FnBaseVector.T

`FnBaseVector.T`

The transpose of a vector, the functional given by `(. , self)`

Return`transpose : InnerProductOperator`

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```

>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0

```

FnBaseVector.dtype

`FnBaseVector.dtype`

Length of this vector, equal to space size.

FnBaseVector.itemsize

`FnBaseVector.itemsize`

The size in bytes on one element of this type.

FnBaseVector.nbytes

`FnBaseVector.nbytes`

The number of bytes this vector uses in memory.

FnBaseVector.ndim

`FnBaseVector.ndim`

Number of dimensions, always 1.

FnBaseVector.shape

`FnBaseVector.shape`

Number of entries per axis, equals (size,) for linear storage.

FnBaseVector.size

`FnBaseVector.size`

Length of this vector, equal to space size.

FnBaseVector.space

`FnBaseVector.space`

Space to which this vector.

FnBaseVector.ufunc

`FnBaseVector.ufunc`

NtuplesBaseUFuncs, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

<code>__eq__(other)</code>	
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to self.
<code>copy()</code>	
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, show, fig])</code>	Display the function graphically.

FnBaseVector.__eq__

`FnBaseVector.__eq__(other)`

FnBaseVector.__getitem__

`FnBaseVector.__getitem__(indices)`

Access values of this vector.

Parameters`indices` : `int` or `slice`

The position(s) that should be accessed

Returns`values` : `NtuplesBase.dtype` or `NtuplesBaseVector`

The value(s) at the index (`indices`)

FnBaseVector.__setitem__

`FnBaseVector.__setitem__(indices, values)`

Set values of this vector.

Parameters`indices` : `int` or `slice`

The position(s) that should be set

values : scalar, *array-like* or `NtuplesBaseVector`

The value(s) that are to be assigned.

If `index` is an integer, `value` must be single value.

If `index` is a slice, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

FnBaseVector.asarray

`FnBaseVector.asarray(start=None, stop=None, step=None, out=None)`

Extract the data of this array as a numpy array.

Parameters`start` : `int`, optional

Start position. `None` means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : numpy.ndarray

Array to write result to.

Returns asarray : numpy.ndarray

Numpy array of the same type as the space.

FnBaseVector.assign

FnBaseVector.**assign** (*other*)

Assign the values of *other* to self.

FnBaseVector.copy

FnBaseVector.**copy** ()

FnBaseVector.dist

FnBaseVector.**dist** (*other*)

Distance to *other*.

LinearSpace.dist

FnBaseVector.divide

FnBaseVector.**divide** (*x*, *y*)

Divide by *other* inplace.

LinearSpace.divide

FnBaseVector.inner

FnBaseVector.**inner** (*other*)

Inner product with *other*.

LinearSpace.inner

FnBaseVector.lincomb

FnBaseVector.**lincomb** (*a*, *x1*, *b=None*, *x2=None*)

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

FnBaseVector.multiply

FnBaseVector.**multiply** (*x*, *y*)

Multiply by *other* inplace.

LinearSpace.multiply

FnBaseVector.norm`FnBaseVector.norm()`

Norm of vector

*LinearSpace.norm***FnBaseVector.set_zero**`FnBaseVector.set_zero()`

Set this vector to zero.

*LinearSpace.zero***FnBaseVector.show**`FnBaseVector.show(title=None, method='scatter', show=False, fig=None, **kwargs)`

Display the function graphically.

Parameter`title` : str, optional

Set the title of the figure

method : str, optional

Id methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.FigureThe figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.**kwargs** : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig` : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

See also:*odl.util.graphics.show_discrete_data* Underlying implementation`__init__(space, *args, **kwargs)`

Initialize a new instance.

FnWeightingBase`class odl.space.base_ntuples.FnWeightingBase(impl, exponent=2.0, dist_using_inner=False)`

Bases: object

Abstract base class for weighting of *FnBase* spaces.

This class and its subclasses serve as a simple means to evaluate and compare weighted inner products, norms and metrics semantically rather than by identity on a pure function level.

The functions are implemented similarly to *Operator*, but without extra type checks of input parameters - this is done in the callers of the *LinearSpace* instance where these functions used.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

FnWeightingBase.exponent

`FnWeightingBase.exponent`

Exponent of this weighting.

FnWeightingBase.impl

`FnWeightingBase.impl`

Implementation backend of this weighting.

Methods

<i>__eq__</i> (other)	Return self == other.
<i>dist</i> (x1, x2)	Calculate the distance between two vectors.
<i>equiv</i> (other)	Test if other is an equivalent inner product.
<i>inner</i> (x1, x2)	Calculate the inner product of two vectors.
<i>norm</i> (x)	Calculate the norm of a vector.

FnWeightingBase.__eq__

`FnWeightingBase.__eq__(other)`

Return self == other.

Return`sequal` : bool

True if other is a the same weighting, False otherwise.

Notes

This operation must be computationally cheap, i.e. no large arrays may be compared element-wise. That is the task of the *equiv* method.

FnWeightingBase.dist

`FnWeightingBase.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using *norm*. Subclasses should override it for optimization purposes.

Parameters`x1, x2` : *FnBaseVector*

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

FnWeightingBase.equiv

`FnWeightingBase.equiv` (*other*)

Test if *other* is an equivalent inner product.

Should be overwritten, default tests for equality.

Return`seivalent` : bool

True if *other* is a `FnWeightingBase` instance which yields the same result as this inner product for any input, False otherwise.

FnWeightingBase.inner

`FnWeightingBase.inner` (*x1*, *x2*)

Calculate the inner product of two vectors.

Parameters`x1`, `x2` : `FnBaseVector`

Vectors whose inner product is calculated

Return`sinner` : float or complex

The inner product of the two provided vectors

FnWeightingBase.norm

`FnWeightingBase.norm` (*x*)

Calculate the norm of a vector.

This is the standard implementation using `inner`. Subclasses should override it for optimization purposes.

Parameters`x1` : `FnBaseVector`

Vector whose norm is calculated

Return`snorm` : float

The norm of the vector

__init__ (*impl*, *exponent*=2.0, *dist_using_inner*=False)

Initialize a new instance.

Parameters`simpl` : str

Specifier for the implementation backend

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) *x* and *y*.

This option can only be used if `exponent` is 2.0.

Default: False.

NtuplesBase

class odl.space.base_ntuples.**NtuplesBase** (*size, dtype*)

Bases: *odl.set.sets.Set*

Base class for sets of n-tuples independent of implementation.

Attributes

<i>dtype</i>	The data type of each entry.
<i>element_type</i>	<i>NtuplesBaseVector</i>
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.

NtuplesBase.dtype

NtuplesBase.**dtype**

The data type of each entry.

NtuplesBase.element_type

NtuplesBase.**element_type**

NtuplesBaseVector

NtuplesBase.shape

NtuplesBase.**shape**

The shape of this space.

NtuplesBase.size

NtuplesBase.**size**

The number of entries per tuple.

Methods

<i>__contains__</i> (<i>other</i>)	Return <i>other</i> in <i>self</i> .
<i>__eq__</i> (<i>other</i>)	Return <i>self</i> == <i>other</i> .
<i>contains_all</i> (<i>other</i>)	Test if all points in <i>other</i> are contained in this set.
<i>contains_set</i> (<i>other</i>)	Test if <i>other</i> is a subset of this set.
<i>element</i> ([<i>inp</i>])	Return an element from <i>inp</i> or from scratch.

NtuplesBase.__contains__

NtuplesBase.**__contains__** (*other*)

Return *other* in *self*.

Returns*contains* : bool

True if *other* is an *NtuplesBaseVector* instance and *other.space* is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

NtuplesBase.__eq__

NtuplesBase.__eq__(other)

Return self == other.

Returnsequals: bool

True if other is an instance of this space's type with the same *size* and *dtype*, otherwise False.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

NtuplesBase.contains_all

NtuplesBase.contains_all(other)

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

NtuplesBase.contains_set

NtuplesBase.contains_set(other)

Test if other is a subset of this set.

Implementing this method is optional. Default it tests for equality.

NtuplesBase.element

`NtuplesBase.element` (*inp=None*)

Return an element from *inp* or from scratch.

Implementing this method is optional.

`__init__` (*size, dtype*)

Initialize a new instance.

Parameters*size* : non-negative int

The number of entries per tuple

dtype :

The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

NtuplesBaseVector

class `odl.space.base_ntuples.NtuplesBaseVector` (*space, *args, **kwargs*)

Bases: `object`

Abstract class for representation of *NtuplesBase* elements.

Defines abstract attributes and concrete ones which are independent of data representation.

Attributes

<i>dtype</i>	Length of this vector, equal to <i>space</i> size.
<i>itemsize</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
<i>shape</i>	Number of entries per axis, equals (<i>size</i> ,) for linear storage.
<i>size</i>	Length of this vector, equal to <i>space</i> size.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>NtuplesBaseUFuncs</i> , access to numpy style ufuncs.

NtuplesBaseVector.dtype

`NtuplesBaseVector.dtype`

Length of this vector, equal to *space* size.

NtuplesBaseVector.itemsize

`NtuplesBaseVector.itemsize`

The size in bytes on one element of this type.

NtuplesBaseVector.nbytes

`NtuplesBaseVector.nbytes`

The number of bytes this vector uses in memory.

NtuplesBaseVector.ndim

`NtuplesBaseVector.ndim`

Number of dimensions, always 1.

`NtuplesBaseVector.shape`

`NtuplesBaseVector.shape`

Number of entries per axis, equals (size,) for linear storage.

`NtuplesBaseVector.size`

`NtuplesBaseVector.size`

Length of this vector, equal to space size.

`NtuplesBaseVector.space`

`NtuplesBaseVector.space`

Space to which this vector.

`NtuplesBaseVector.ufunc`

`NtuplesBaseVector.ufunc`

NtuplesBaseUFuncs, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>show([title, method, show, fig])</code>	Display the function graphically.

`NtuplesBaseVector.__eq__`

`NtuplesBaseVector.__eq__(other)`

Return `self == other`.

Returnsequals : bool

True if all entries of `other` are equal to this vector's entries, False otherwise.

`NtuplesBaseVector.__getitem__`

`NtuplesBaseVector.__getitem__(indices)`

Access values of this vector.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : *NtuplesBase.dtype* or *NtuplesBaseVector*

The value(s) at the index (indices)

`NtuplesBaseVector.__setitem__`

`NtuplesBaseVector.__setitem__(indices, values)`

Set values of this vector.

Parameters`indices` : int or slice

The position(s) that should be set

values : scalar, *array-like* or *NtuplesBaseVector*

The value(s) that are to be assigned.

If `index` is an integer, `value` must be single value.

If `index` is a slice, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

NtuplesBaseVector.asarray

`NtuplesBaseVector.asarray(start=None, stop=None, step=None, out=None)`

Extract the data of this array as a numpy array.

Parameters`start` : int, optional

Start position. `None` means the first element.

start : int, optional

One element past the last element to be extracted. `None` means the last element.

start : int, optional

Step length. `None` means 1.

out : `numpy.ndarray`

Array to write result to.

Returns`asarray` : `numpy.ndarray`

Numpy array of the same type as the space.

NtuplesBaseVector.copy

`NtuplesBaseVector.copy()`

Create an identical (deep) copy of this vector.

NtuplesBaseVector.show

`NtuplesBaseVector.show(title=None, method='scatter', show=False, fig=None, **kwargs)`

Display the function graphically.

Parameter`title` : str, optional

Set the title of the figure

method : str, optional

Id methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig: `matplotlib.figure.Figure`

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.

kwargs: {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns**fig**: `matplotlib.figure.Figure`

The resulting figure. It is also shown to the user.

See also:

`odl.util.graphics.show_discrete_data` Underlying implementation

`__init__` (*space*, **args*, ***kwargs*)

Initialize a new instance.

cu_ntuples

CUDA implementation of n-dimensional Cartesian spaces.

Classes

<code>CudaFn</code> (<i>size</i> , <i>dtype</i> , ** <i>kwargs</i>)	The space <code>FnBase</code> , implemented in CUDA.
<code>CudaFnConstWeighting</code> (<i>constant</i> [, <i>exponent</i>])	Weighting of <code>CudaFn</code> by a constant.
<code>CudaFnCustomDist</code> (<i>dist</i>)	Custom distance on <code>CudaFn</code> , removes <code>norm</code> and <code>inner</code> .
<code>CudaFnCustomInnerProduct</code> (<i>inner</i> [, ...])	Custom inner product on <code>CudaFn</code> .
<code>CudaFnCustomNorm</code> (<i>norm</i>)	Custom norm on <code>CudaFn</code> , removes <code>inner</code> .
<code>CudaFnNoWeighting</code> ([<i>exponent</i>])	Weighting of <code>CudaFn</code> with constant 1.
<code>CudaFnVector</code> (<i>space</i> , <i>data</i>)	Representation of a <code>CudaFn</code> element.
<code>CudaFnVectorWeighting</code> (<i>vector</i> [, <i>exponent</i>])	Vector weighting for <code>CudaFn</code> .
<code>CudaNtuples</code> (<i>size</i> , <i>dtype</i>)	The space <code>NtuplesBase</code> , implemented in CUDA.
<code>CudaNtuplesVector</code> (<i>space</i> , <i>data</i>)	Representation of a <code>CudaNtuples</code> element.

CudaFn

class `odl.space.cu_ntuples.CudaFn` (*size*, *dtype*, ***kwargs*)

Bases: `odl.space.base_ntuples.FnBase`, `odl.space.cu_ntuples.CudaNtuples`

The space `FnBase`, implemented in CUDA.

Requires the compiled ODL extension `odlpp`.

Attributes

<code>dtype</code>	The data type of each entry.
<code>element_type</code>	<code>CudaFnVector</code>

Continued on next page

Table 2.168 – continued from previous page

<i>exponent</i>	Exponent of the norm and distance.
<i>field</i>	The field of this vector space.
<i>is_cn</i>	Return <code>True</code> if the space represents \mathbb{C}^n , i.e.
<i>is_rn</i>	Return <code>True</code> if the space represents \mathbb{R}^n , i.e.
<i>is_weighted</i>	Return <code>True</code> if the weighting is not <code>CudaFnNoWeighting</code> .
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.
<i>weighting</i>	This space’s weighting scheme.

CudaFn.dtype`CudaFn.dtype`

The data type of each entry.

CudaFn.element_type`CudaFn.element_type``CudaFnVector`**CudaFn.exponent**`CudaFn.exponent`

Exponent of the norm and distance.

CudaFn.field`CudaFn.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns`field : Field`

The underlying field.

CudaFn.is_cn`CudaFn.is_cn`Return `True` if the space represents \mathbb{C}^n , i.e. complex tuples.**CudaFn.is_rn**`CudaFn.is_rn`Return `True` if the space represents \mathbb{R}^n , i.e. real tuples.**CudaFn.is_weighted**`CudaFn.is_weighted`Return `True` if the weighting is not `CudaFnNoWeighting`.**CudaFn.shape**`CudaFn.shape`

The shape of this space.

CudaFn.size**CudaFn.size**

The number of entries per tuple.

CudaFn.weighting**CudaFn.weighting**

This space's weighting scheme.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	<code>s.__eq__(other) <==> s == other</code> .
<code>_dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>_divide(x1, x2, out)</code>	The pointwise division of two vectors, assigned to out.
<code>_inner(x1, x2)</code>	Calculate the inner product of x and y.
<code>_lincomb(a, x1, b, x2, out)</code>	Linear combination of x1 and x2, assigned to out.
<code>_multiply(x1, x2, out)</code>	The pointwise product of two vectors, assigned to out.
<code>_norm(x)</code>	Calculate the norm of x.
<code>astype(dtype)</code>	Return a copy of this space with new dtype.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>default_dtype(field)</code>	Return the default of <i>CudaFn</i> data type for a given field.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp, data_ptr])</code>	Create a new element.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create a vector of ones.
<code>zero()</code>	Create a vector of zeros.

CudaFn.__contains__**CudaFn.__contains__**(other)

Return other in self.

Returns`contains` : boolTrue if other is an *NtuplesBaseVector* instance and other.space is equal to this space, False otherwise.**Examples**

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

CudaFn.__eq__

CudaFn.__eq__(other)

s.__eq__(other) <==> s == other.

Return equals : bool

True if other is an instance of this space's type with the same size, dtype and space functions, otherwise False.

Examples

```
>>> from numpy.linalg import norm
>>> def dist(x, y, ord):
...     return norm(x - y, ord)
```

```
>>> from functools import partial
>>> dist2 = partial(dist, ord=2)
>>> r3 = CudaRn(3, dist=dist2)
>>> r3_same = CudaRn(3, dist=dist2)
>>> r3 == r3_same
True
```

Different dist functions result in different spaces - the same applies for norm and inner:

```
>>> dist1 = partial(dist, ord=1)
>>> r3_1 = CudaRn(3, dist=dist1)
>>> r3_2 = CudaRn(3, dist=dist2)
>>> r3_1 == r3_2
False
```

Be careful with Lambdas - they result in non-identical function objects:

```
>>> r3_lambda1 = CudaRn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> r3_lambda2 = CudaRn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> r3_lambda1 == r3_lambda2
False
```

CudaFn._dist

CudaFn._dist(x1, x2)

Calculate the distance between two vectors.

Parameters x1, x2 : *CudaFnVector*

The vectors whose mutual distance is calculated

Returns dist : float

Distance between the vectors

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([3, 8])
>>> y = r2.element([0, 4])
>>> r2.dist(x, y)
5.0
```

CudaFn._divide**CudaFn._divide** (*x1*, *x2*, *out*)The pointwise division of two vectors, assigned to *out*.

This is defined as:

 $\text{multiply}(z, x, y) := [x[0]/y[0], x[1]/y[1], \dots, x[n-1]/y[n-1]]$ **Parameters***x1*, *x2* : *CudaFnVector*

Factors in the product

out : *CudaFnVector*

Element to which the result is written

ReturnsNone**Examples**

```
>>> rn = CudaRn(3)
>>> x1 = rn.element([5, 3, 2])
>>> x2 = rn.element([1, 2, 2])
>>> out = rn.element()
>>> rn.divide(x1, x2, out) # out is returned
CudaRn(3).element([5.0, 1.5, 1.0])
>>> out
CudaRn(3).element([5.0, 1.5, 1.0])
```

CudaFn._inner**CudaFn._inner** (*x1*, *x2*)Calculate the inner product of *x* and *y*.**Parameters***x1*, *x2* : *CudaFnVector***Returns**inner: float or complexThe inner product of *x* and *y***Examples**

```
>>> uc3 = CudaFn(3, 'uint8')
>>> x = uc3.element([1, 2, 3])
>>> y = uc3.element([3, 1, 5])
>>> uc3.inner(x, y)
20.0
```

CudaFn._lincomb**CudaFn._lincomb** (*a*, *x1*, *b*, *x2*, *out*)Linear combination of *x1* and *x2*, assigned to *out*.Calculate $z = a * x + b * y$ using optimized CUDA routines.**Parameters***a*, *b* : *LinearSpace.field element*Scalar to multiply *x* and *y* with.**x**, **y** : *CudaFnVector*

The summands

out : *CudaFnVector*

The Vector that the result is written to.

ReturnsNone

Examples

```
>>> r3 = CudaRn(3)
>>> x = r3.element([1, 2, 3])
>>> y = r3.element([4, 5, 6])
>>> out = r3.element()
>>> r3.lincomb(2, x, 3, y, out) # out is returned
CudaRn(3).element([14.0, 19.0, 24.0])
>>> out
CudaRn(3).element([14.0, 19.0, 24.0])
```

CudaFn._multiply

CudaFn._**multiply**(x1, x2, out)

The pointwise product of two vectors, assigned to out.

This is defined as:

$\text{multiply}(x, y, \text{out}) := [x[0]*y[0], x[1]*y[1], \dots, x[n-1]*y[n-1]]$

Parametersx1, x2 : *CudaFnVector*

Factors in product

out : *CudaFnVector*

Element to which the result is written

ReturnsNone

Examples

```
>>> rn = CudaRn(3)
>>> x1 = rn.element([5, 3, 2])
>>> x2 = rn.element([1, 2, 3])
>>> out = rn.element()
>>> rn.multiply(x1, x2, out) # out is returned
CudaRn(3).element([5.0, 6.0, 6.0])
>>> out
CudaRn(3).element([5.0, 6.0, 6.0])
```

CudaFn._norm

CudaFn._**norm**(x)

Calculate the norm of x.

This method is implemented separately from `sqrt(inner(x, x))` for efficiency reasons.

Parametersx : *CudaFnVector*

Returnsnorm : float

The norm of x

Examples

```
>>> uc3 = CudaFn(3, 'uint8')
>>> x = uc3.element([2, 3, 6])
>>> uc3.norm(x)
7.0
```

CudaFn.astype

CudaFn.**astype** (*dtype*)

Return a copy of this space with new dtype.

Parametersdtype :

Data type of the returned space. Can be given in any way `numpy.dtype` understands, e.g. as string ('complex64') or data type (`complex`).

Returnsnewspace : *FnBase*

The version of this space with given data type

CudaFn.contains_all

CudaFn.**contains_all** (*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

CudaFn.contains_set

CudaFn.**contains_set** (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

CudaFn.default_dtype

static CudaFn.**default_dtype** (*field*)

Return the default of *CudaFn* data type for a given field.

Parametersfield : *Field*

Set of numbers to be represented by a data type. Currently supported: *RealNumbers*.

Returnsdtype : type

Numpy data type specifier. The returned defaults are:

```
RealNumbers() : , np.dtype('float32')
```

CudaFn.dist

CudaFn.**dist** (*x1*, *x2*)

Calculate the distance between two vectors.

Parametersx1, x2 : *LinearSpaceVector*

Vectors whose distance to compute

Returnsdist : float

Distance between vectors

CudaFn.divide

CudaFn.**divide** (*x1*, *x2*, *out=None*)

Calculate the pointwise division of *x1* and *x2*

Parameters*x1* : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns*out* : *LinearSpaceVector*

Ratio of the vectors. If *out* was provided, the returned object is a reference to it.

CudaFn.element

CudaFn.**element** (*inp=None*, *data_ptr=None*)

Create a new element.

Parameters*inp* : *array-like* or scalar, optional

Input to initialize the new element.

If *inp* is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied (with broadcasting if necessary).

If a single value is given, it is copied to all entries.

If both *inp* and *data_ptr* are `None`, an empty element is created with no guarantee of its state (memory allocation only).

data_ptr : `int`, optional

Memory address of a CUDA array container

Cannot be combined with *inp*.

Return*element* : *CudaNtuplesVector*

The new element

Notes

This method preserves “array views” of correct size and type, see the examples below.

TODO: No, it does not yet!

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> x = uc3.element(np.array([1, 2, 3], dtype='uint8'))
>>> x
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> y = uc3.element([1, 2, 3])
```

```
>>> y
CudaNtuples(3, 'uint8').element([1, 2, 3])
```

CudaFn.inner

CudaFn.**inner** (*x1*, *x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1*, *x2* : *LinearSpaceVector*

Factors in the inner product

Returns*out* : *LinearSpace.field* element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

CudaFn.lincomb

CudaFn.**lincomb** (*a*, *x1*, *b=None*, *x2=None*, *out=None*)

Linear combination of vectors.

Calculates

$out = a * x1$

or, if *b* and *y* are given,

$out = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply *x2* with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns*out* : *LinearSpaceVector*

Result of the linear combination. If *out* was provided, the returned object is a reference to it.

Notes

The vectors *out*, *x1* and *x2* may be aligned, thus a call

`space.lincomb(x, 2, x, 3.14, out=x)`

is (mathematically) equivalent to

$x = x * (1 + 2 + 3.14)$

CudaFn.multiply

`CudaFn.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

CudaFn.norm

`CudaFn.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

CudaFn.one

`CudaFn.one()`

Create a vector of ones.

CudaFn.zero

`CudaFn.zero()`

Create a vector of zeros.

__init__(*size, dtype, **kwargs*)

Initialize a new instance.

Parameters`size` : positive int

The number of dimensions of the space

dtype : object

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only scalar data types are allowed.

weight : optional

Use weighted inner product, norm, and dist. The following types are supported as `weight`:

FnWeightingBase : Use this weighting as-is. Compatibility with this space's elements is not checked during init.

float : Weighting by a constant

array-like: Weighting by a vector (1-dim. array, corresponds to a diagonal matrix). Note that the array is stored in main memory, which results in slower space functions due to a copy during evaluation.

CudaFnVector: same as 1-dim. array-like, except that copying is avoided if the dtype of the vector is the same as this space's dtype.

Default: no weighting

This option cannot be combined with `dist`, `norm` or `inner`.

exponent: positive float, optional

Exponent of the norm. For values other than 2.0, no inner product is defined.

This option is ignored if `dist`, `norm` or `inner` is given.

Default: 2.0

dist: callable, optional

The distance function defining a metric on *CudaFn*. It must accept two *CudaFnVector* arguments, return a float and fulfill the following mathematical conditions for any three vectors x , y , z :

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

By default, `dist(x, y)` is calculated as `norm(x - y)`. This creates an intermediate array $x-y$, which can be avoided by choosing `dist_using_inner=True`.

This option cannot be combined with `weight`, `norm` or `inner`.

norm: callable, optional

The norm implementation. It must accept an *CudaFnVector* argument, return a float and satisfy the following conditions for all vectors x, y and scalars s :

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

By default, `norm(x)` is calculated as `inner(x, x)`.

This option cannot be combined with `weight`, `dist` or `inner`.

inner: callable, optional

The inner product implementation. It must accept two *CudaFnVector* arguments, return an element from the field of the space (real or complex number) and satisfy the following conditions for all vectors x, y, z and scalars s :

- $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- $\langle sx, y \rangle = s \langle x, y \rangle$
- $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
- $\langle x, x \rangle = 0 \Leftrightarrow x = 0$

This option cannot be combined with `weight`, `dist` or `norm`.

CudaFnConstWeighting

class `odl.space.cu_ntuples.CudaFnConstWeighting` (*constant*, *exponent*=2.0)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Weighting of `CudaFn` by a constant.

For exponent 2.0, a new weighted inner product with constant c is defined as

$$\langle a, b \rangle_c := c \, b^H a$$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is defined as

$$\|a\|_{c,\infty} := c \|a\|_\infty,$$

otherwise it is

$$\|a\|_{c,p} := c^{1/p} \|a\|_p.$$

Not that this definition does **not** fulfill the limit property in p , i.e.

$$\lim_{p \rightarrow \infty} \|a\|_{c,p} = \|a\|_\infty \neq \|a\|_{c,\infty}$$

unless $c = 1$.

The constant c must be positive.

Attributes

<code>const</code>	Weighting constant of this inner product.
<code>exponent</code>	Exponent of this weighting.
<code>impl</code>	Implementation backend of this weighting.

CudaFnConstWeighting.const

`CudaFnConstWeighting.const`

Weighting constant of this inner product.

CudaFnConstWeighting.exponent

`CudaFnConstWeighting.exponent`

Exponent of this weighting.

CudaFnConstWeighting.impl

`CudaFnConstWeighting.impl`

Implementation backend of this weighting.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
----------------------------	-------------------------------------

Continued on next page

Table 2.171 – continued from previous page

<i>dist</i> (x1, x2)	Calculate the constant-weighted distance between two vectors.
<i>equiv</i> (other)	Test if <i>other</i> is an equivalent weighting.
<i>inner</i> (x1, x2)	Calculate the constant-weighted inner product of two vectors.
<i>norm</i> (x)	Calculate the constant-weighted norm of a vector.

CudaFnConstWeighting.__eq__

CudaFnConstWeighting.**__eq__**(*other*)

Return self == other.

Returnsequal : bool

True if *other* is a *CudaFnConstWeighting* instance with the same constant,
False otherwise.

CudaFnConstWeighting.dist

CudaFnConstWeighting.**dist**(*x1*, *x2*)

Calculate the constant-weighted distance between two vectors.

Parameters*x1*, *x2* : *CudaFnVector*

Vectors whose mutual distance is calculated

Returns*dist* : float

The distance between the vectors

CudaFnConstWeighting.equiv

CudaFnConstWeighting.**equiv**(*other*)

Test if *other* is an equivalent weighting.

Returnsequivalent : bool

True if *other* is a *FnWeightingBase* instance with the same
FnWeightingBase.impl, which yields the same result as this inner product
for any input, False otherwise. This is the same as equality if *other* is a
CudaFnConstWeighting instance, otherwise by entry-wise comparison of this
inner product's constant with the matrix of *other*.

CudaFnConstWeighting.inner

CudaFnConstWeighting.**inner**(*x1*, *x2*)

Calculate the constant-weighted inner product of two vectors.

Parameters*x1*, *x2* : *CudaFnVector*

Vectors whose inner product is calculated

Returns*inner* : float or complex

The inner product of the two vectors

CudaFnConstWeighting.norm

CudaFnConstWeighting.**norm**(*x*)

Calculate the constant-weighted norm of a vector.

Parameters*x1* : *CudaFnVector*

Vector whose norm is calculated

Returns`norm` : float

The norm of the vector

`__init__` (*constant*, *exponent*=2.0)

Initialize a new instance.

Parameters`constant` : positive finite float

Weighting constant of the inner product.

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

CudaFnCustomDist

class `odl.space.cu_ntuples.CudaFnCustomDist` (*dist*)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Custom distance on `CudaFn`, removes norm and inner.

Attributes

<code>exponent</code>	Exponent of this weighting.
<code>impl</code>	Implementation backend of this weighting.

CudaFnCustomDist.exponent

`CudaFnCustomDist.exponent`

Exponent of this weighting.

CudaFnCustomDist.impl

`CudaFnCustomDist.impl`

Implementation backend of this weighting.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Custom distance of this instance..
<code>equiv(other)</code>	Test if <code>other</code> is an equivalent inner product.
<code>inner(x1, x2)</code>	Inner product is not defined for custom distance.
<code>norm(x)</code>	Norm is not defined for custom distance.

CudaFnCustomDist.__eq__

`CudaFnCustomDist.__eq__(other)`

Return `self == other`.

Return`sequal` : bool

True if `other` is a `CudaFnCustomDist` instance with the same norm, False otherwise.

CudaFnCustomDist.dist

`CudaFnCustomDist.dist` ($x1, x2$)
Custom distance of this instance..

CudaFnCustomDist.equiv

`CudaFnCustomDist.equiv` (*other*)
Test if *other* is an equivalent inner product.
Should be overwritten, default tests for equality.

Return`equiv` : bool

True if *other* is a *FnWeightingBase* instance which yields the same result as this inner product for any input, False otherwise.

CudaFnCustomDist.inner

`CudaFnCustomDist.inner` ($x1, x2$)
Inner product is not defined for custom distance.

CudaFnCustomDist.norm

`CudaFnCustomDist.norm` (x)
Norm is not defined for custom distance.
`__init__` (*dist*)
Initialize a new instance.

Parameters`dist` : callable

The distance function defining a metric on \mathbb{F}^n . It must accept two *CudaFnVector* arguments and fulfill the following mathematical conditions for any three vectors x, y, z :

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

CudaFnCustomInnerProduct

`class odl.space.cu_ntuples.CudaFnCustomInnerProduct` (*inner*, *dist_using_inner=True*)
Bases: `odl.space.base_ntuples.FnWeightingBase`
Custom inner product on *CudaFn*.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

CudaFnCustomInnerProduct.exponent

`CudaFnCustomInnerProduct.exponent`
Exponent of this weighting.

CudaFnCustomInnerProduct.impl

`CudaFnCustomInnerProduct.impl`

Implementation backend of this weighting.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>equiv(other)</code>	Test if <code>other</code> is an equivalent inner product.
<code>inner(x1, x2)</code>	Custom inner product of this instance..
<code>norm(x)</code>	Calculate the norm of a vector.

CudaFnCustomInnerProduct.__eq__

`CudaFnCustomInnerProduct.__eq__(other)`

Return `self == other`.

Return`sequal` : bool

True if `other` is a `CudaFnCustomInnerProduct` instance with the same inner product, False otherwise.

CudaFnCustomInnerProduct.dist

`CudaFnCustomInnerProduct.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using `norm`. Subclasses should override it for optimization purposes.

Parameters`x1, x2` : `FnBaseVector`

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

CudaFnCustomInnerProduct.equiv

`CudaFnCustomInnerProduct.equiv(other)`

Test if `other` is an equivalent inner product.

Should be overwritten, default tests for equality.

Return`sequivalent` : bool

True if `other` is a `FnWeightingBase` instance which yields the same result as this inner product for any input, False otherwise.

CudaFnCustomInnerProduct.inner

`CudaFnCustomInnerProduct.inner(x1, x2)`

Custom inner product of this instance..

CudaFnCustomInnerProduct.norm

`CudaFnCustomInnerProduct.norm(x)`

Calculate the norm of a vector.

This is the standard implementation using `inner`. Subclasses should override it for optimization purposes.

Parameters**x1** : *FnBaseVector*

Vector whose norm is calculated

Returns**norm** : float

The norm of the vector

__init__(*inner, dist_using_inner=True*)

Initialize a new instance.

Parameters**inner** : callable

The inner product implementation. It must accept two *CudaFnVector* arguments, return a complex number and satisfy the following conditions for all vectors x, y, z and scalars s :

- $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- $\langle sx, y \rangle = s \langle x, y \rangle$
- $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
- $\langle x, x \rangle = 0 \Leftrightarrow x = 0$

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

CudaFnCustomNorm

class `odl.space.cu_ntuples.CudaFnCustomNorm`(*norm*)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Custom norm on *CudaFn*, removes `inner`.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

CudaFnCustomNorm.exponent

`CudaFnCustomNorm.exponent`

Exponent of this weighting.

CudaFnCustomNorm.impl

`CudaFnCustomNorm.impl`

Implementation backend of this weighting.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>equiv(other)</code>	Test if <code>other</code> is an equivalent inner product.
<code>inner(x1, x2)</code>	Inner product is not defined for custom distance.
<code>norm(x)</code>	Custom norm of this instance..

CudaFnCustomNorm.__eq__

`CudaFnCustomNorm.__eq__(other)`

Return `self == other`.

Return`sequal` : bool

True if `other` is a `CudaFnCustomNorm` instance with the same norm, False otherwise.

CudaFnCustomNorm.dist

`CudaFnCustomNorm.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using `norm`. Subclasses should override it for optimization purposes.

Parameters`x1, x2` : `FnBaseVector`

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

CudaFnCustomNorm.equiv

`CudaFnCustomNorm.equiv(other)`

Test if `other` is an equivalent inner product.

Should be overwritten, default tests for equality.

Return`sequivalent` : bool

True if `other` is a `FnWeightingBase` instance which yields the same result as this inner product for any input, False otherwise.

CudaFnCustomNorm.inner

`CudaFnCustomNorm.inner(x1, x2)`

Inner product is not defined for custom distance.

CudaFnCustomNorm.norm

`CudaFnCustomNorm.norm(x)`

Custom norm of this instance..

`__init__(norm)`

Initialize a new instance.

Parameters`norm` : callable

The norm implementation. It must accept an `CudaFnVector` argument, return a float and satisfy the following conditions for all vectors x, y and scalars s :

$$\bullet \|x\| \geq 0$$

- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

CudaFnNoWeighting

class `odl.space.cu_ntuples.CudaFnNoWeighting` (*exponent=2.0*)

Bases: `odl.space.cu_ntuples.CudaFnConstWeighting`

Weighting of *CudaFn* with constant 1.

For exponent 2.0, the unweighted inner product is defined as

$$\langle a, b \rangle := b^H a$$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined.

Attributes

<i>const</i>	Weighting constant of this inner product.
<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

CudaFnNoWeighting.const

`CudaFnNoWeighting.const`

Weighting constant of this inner product.

CudaFnNoWeighting.exponent

`CudaFnNoWeighting.exponent`

Exponent of this weighting.

CudaFnNoWeighting.impl

`CudaFnNoWeighting.impl`

Implementation backend of this weighting.

Methods

<i>__eq__</i> (other)	Return <code>self == other</code> .
<i>dist</i> (x1, x2)	Calculate the constant-weighted distance between two vectors.
<i>equiv</i> (other)	Test if <code>other</code> is an equivalent weighting.
<i>inner</i> (x1, x2)	Calculate the constant-weighted inner product of two vectors.
<i>norm</i> (x)	Calculate the constant-weighted norm of a vector.

CudaFnNoWeighting.__eq__

`CudaFnNoWeighting.__eq__(other)`

Return `self == other`.

Returnsequal : bool

True if other is a *CudaFnConstWeighting* instance with the same constant,
False otherwise.

CudaFnNoWeighting.dist

CudaFnNoWeighting.**dist** (*x1*, *x2*)

Calculate the constant-weighted distance between two vectors.

Parameters*x1*, *x2* : *CudaFnVector*

Vectors whose mutual distance is calculated

Returns*dist* : float

The distance between the vectors

CudaFnNoWeighting.equiv

CudaFnNoWeighting.**equiv** (*other*)

Test if other is an equivalent weighting.

Returnsequivalent : bool

True if other is a *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, False otherwise. This is the same as equality if other is a *CudaFnConstWeighting* instance, otherwise by entry-wise comparison of this inner product's constant with the matrix of other.

CudaFnNoWeighting.inner

CudaFnNoWeighting.**inner** (*x1*, *x2*)

Calculate the constant-weighted inner product of two vectors.

Parameters*x1*, *x2* : *CudaFnVector*

Vectors whose inner product is calculated

Returns*sinner* : float or complex

The inner product of the two vectors

CudaFnNoWeighting.norm

CudaFnNoWeighting.**norm** (*x*)

Calculate the constant-weighted norm of a vector.

Parameters*x1* : *CudaFnVector*

Vector whose norm is calculated

Returns*norm* : float

The norm of the vector

__init__ (*exponent=2.0*)

Initialize a new instance.

CudaFnVector

class `odl.space.cu_ntuples.CudaFnVector(space, data)`

Bases: `odl.space.base_ntuples.FnBaseVector`, `odl.space.cu_ntuples.CudaNtuplesVector`

Representation of a *CudaFn* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (. , self)
<i>data</i>	The data of this vector.
<i>data_ptr</i>	A raw pointer to the data of this vector.
<i>dtype</i>	Length of this vector, equal to space size.
<i>itemsizes</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	Length of this vector, equal to space size.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>CudaNtuplesUFuncs</i> , access to numpy style ufuncs.

CudaFnVector.T

`CudaFnVector.T`

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

CudaFnVector.data

`CudaFnVector.data`

The data of this vector.

Parameters*None*

Returns*ptr* : `CudaFnVectorImpl`

Underlying cuda data representation

CudaFnVector.data_ptr`CudaFnVector.data_ptr`

A raw pointer to the data of this vector.

CudaFnVector.dtype`CudaFnVector.dtype`

Length of this vector, equal to space size.

CudaFnVector.itemsize`CudaFnVector.itemsize`

The size in bytes on one element of this type.

CudaFnVector.nbytes`CudaFnVector.nbytes`

The number of bytes this vector uses in memory.

CudaFnVector.ndim`CudaFnVector.ndim`

Number of dimensions, always 1.

CudaFnVector.shape`CudaFnVector.shape`

Number of entries per axis, equals (size,) for linear storage.

CudaFnVector.size`CudaFnVector.size`

Length of this vector, equal to space size.

CudaFnVector.space`CudaFnVector.space`

Space to which this vector.

CudaFnVector.ufunc`CudaFnVector.ufunc`

CudaNtuplesUFuncs, access to numpy style ufuncs.

See also:

odl.util.ufuncs.NtuplesBaseUFuncs Base class for ufuncs in *NtuplesBase* spaces.

Notes

Not all ufuncs are currently optimized, some use the default numpy implementation. This can be improved in the future.

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
CudaRn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
CudaRn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
CudaRn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
CudaRn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

<code>__eq__(other)</code>	
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to self.
<code>copy()</code>	
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, show, fig])</code>	Display the function graphically.

CudaFnVector.__eq__

CudaFnVector.__eq__(*other*)

CudaFnVector.__getitem__

`CudaFnVector.__getitem__(indices)`

Access values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parameters`indices` : int or slice

The position(s) that should be accessed

Returns`values` : scalar or `CudaNtuplesVector`

The value(s) at the index (indices)

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0]
1
>>> z = y[1:3]
>>> z
CudaNtuples(2, 'uint8').element([2, 3])
>>> y[:2]
CudaNtuples(2, 'uint8').element([1, 3])
>>> y[:-1]
CudaNtuples(3, 'uint8').element([3, 2, 1])
```

The returned value is a view, modifications are reflected in the original data:

```
>>> z[:] = [4, 5]
>>> y
CudaNtuples(3, 'uint8').element([1, 4, 5])
```

CudaFnVector.__setitem__

`CudaFnVector.__setitem__(indices, values)`

Set values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parameters`indices` : int or slice

The position(s) that should be set

values : scalar, *array-like* or `CudaNtuplesVector`

The value(s) that are to be assigned.

If index is an int, value must be single value.

If index is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

Returns`None`

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 2, 3])
>>> y[1:3] = [7, 8]
>>> y
CudaNtuples(3, 'uint8').element([5, 7, 8])
>>> y[:] = np.array([0, 0, 0])
>>> y
CudaNtuples(3, 'uint8').element([0, 0, 0])
```

Scalar assignment

```
>>> y[:] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 5, 5])
```

CudaFnVector.asarray

CudaFnVector.**asarray** (*start=None, stop=None, step=None, out=None*)

Extract the data of this array as a numpy array.

Parameters**start** : int, optional

Start position. None means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : numpy.ndarray

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returns**asarray** : numpy.ndarray

Numpy array of the same type as the space.

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y.asarray()
array([1, 2, 3], dtype=uint8)
>>> y.asarray(1, 3)
array([2, 3], dtype=uint8)
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='uint8')
>>> result = y.asarray(out=out)
>>> out
array([1, 2, 3], dtype=uint8)
>>> result is out
True
```

CudaFnVector.assign

CudaFnVector.**assign** (*other*)
Assign the values of *other* to self.

CudaFnVector.copy

CudaFnVector.**copy** ()

CudaFnVector.dist

CudaFnVector.**dist** (*other*)
Distance to *other*.
LinearSpace.dist

CudaFnVector.divide

CudaFnVector.**divide** (*x*, *y*)
Divide by *other* inplace.
LinearSpace.divide

CudaFnVector.inner

CudaFnVector.**inner** (*other*)
Inner product with *other*.
LinearSpace.inner

CudaFnVector.lincomb

CudaFnVector.**lincomb** (*a*, *x1*, *b=None*, *x2=None*)
Assign a linear combination to this vector.
Implemented as `space.lincomb(a, x1, b, x2, out=self)`.
LinearSpace.lincomb

CudaFnVector.multiply

CudaFnVector.**multiply** (*x*, *y*)
Multiply by *other* inplace.
LinearSpace.multiply

CudaFnVector.norm

CudaFnVector.**norm** ()
Norm of vector
LinearSpace.norm

CudaFnVector.set_zero

CudaFnVector.**set_zero** ()
Set this vector to zero.
LinearSpace.zero

CudaFnVector.show

`CudaFnVector.show` (*title=None, method='scatter', show=False, fig=None, **kwargs*)

Display the function graphically.

Parameter*title* : str, optional

Set the title of the figure

method : str, optional

Id methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.Figure

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that *fig* is the return value from an earlier call to this function.

kwargs : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns*fig* : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

See also:

[*odl.util.graphics.show_discrete_data*](#) Underlying implementation

__init__ (*space, data*)

Initialize a new instance.

CudaFnVectorWeighting

class `odl.space.cu_ntuples.CudaFnVectorWeighting` (*vector, exponent=2.0*)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Vector weighting for [*CudaFn*](#).

For exponent 2.0, a new weighted inner product with vector *w* is defined as

$$\langle a, b \rangle_w := b^H (w \odot a)$$

with b^H standing for transposed complex conjugate, and $w \odot a$ being element-wise multiplication.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$$\|a\|_{w,\infty} := \|w \odot a\|_{\infty},$$

otherwise it is

$$\|a\|_{w,p} := \|w^{1/p} \odot a\|_p.$$

Not that this definition does **not** fulfill the limit property in *p*, i.e.

$$\lim_{p \rightarrow \infty} \|a\|_{w,p} = \|a\|_{\infty} \neq \|a\|_{w,\infty}$$

unless $w = (1, \dots, 1)$.

The vector may only have positive entries, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.
<i>vector</i>	Weighting vector of this inner product.

CudaFnVectorWeighting.exponent

CudaFnVectorWeighting.**exponent**

Exponent of this weighting.

CudaFnVectorWeighting.impl

CudaFnVectorWeighting.**impl**

Implementation backend of this weighting.

CudaFnVectorWeighting.vector

CudaFnVectorWeighting.**vector**

Weighting vector of this inner product.

Methods

<i>__eq__</i> (other)	Return <code>self == other</code> .
<i>dist</i> (x1, x2)	Calculate the vector-weighted distance between two vectors.
<i>equiv</i> (other)	Test if <code>other</code> is an equivalent weighting.
<i>inner</i> (x1, x2)	Calculate the vector weighted inner product of two vectors.
<i>norm</i> (x)	Calculate the vector-weighted norm of a vector.
<i>vector_is_valid</i> ()	Test if the vector is a valid weight, i.e.

CudaFnVectorWeighting.__eq__

CudaFnVectorWeighting.**__eq__**(other)

Return `self == other`.

Returnsequals : bool

True if other is a *CudaFnVectorWeighting* instance with **identical** vector,
False otherwise.

See also:

equiv test for equivalent inner products

CudaFnVectorWeighting.dist

CudaFnVectorWeighting.**dist**(x1, x2)

Calculate the vector-weighted distance between two vectors.

Parameters x1, x2 : *CudaFnVector*

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

CudaFnVectorWeighting.equiv

`CudaFnVectorWeighting.equiv` (*other*)

Test if *other* is an equivalent weighting.

Return`seivalent` : bool

True if *other* is a *FnWeightingBase* instance which yields the same result as this inner product for any input, False otherwise. This is checked by entry-wise comparison of matrices/vectors/constant of this inner product and *other*.

CudaFnVectorWeighting.inner

`CudaFnVectorWeighting.inner` (*x1*, *x2*)

Calculate the vector weighted inner product of two vectors.

Parameters`x1`, `x2` : *CudaFnVector*

Vectors whose inner product is calculated

Returns`sinner` : float or complex

The inner product of the two provided vectors

CudaFnVectorWeighting.norm

`CudaFnVectorWeighting.norm` (*x*)

Calculate the vector-weighted norm of a vector.

Parameters`x` : *CudaFnVector*

Vector whose norm is calculated

Returns`norm` : float

The norm of the provided vector

CudaFnVectorWeighting.vector_is_valid

`CudaFnVectorWeighting.vector_is_valid` ()

Test if the vector is a valid weight, i.e. positive.

Notes

This operation copies the vector to the CPU memory if necessary and uses `numpy.all`, which can be very time-consuming in total.

__init__ (*vector*, *exponent*=2.0)

Initialize a new instance.

Parameters`vector` : *array-like*, one-dim.

Weighting vector of the inner product

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

CudaNtuples

class `odl.space.cu_ntuples.CudaNtuples` (*size*, *dtype*)

Bases: `odl.space.base_ntuples.NtuplesBase`

The space *NtuplesBase*, implemented in CUDA.

Attributes

<i>dtype</i>	The data type of each entry.
<i>element_type</i>	<i>CudaNtuplesVector</i>
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.

CudaNtuples.dtype

`CudaNtuples.dtype`

The data type of each entry.

CudaNtuples.element_type

`CudaNtuples.element_type`

CudaNtuplesVector

CudaNtuples.shape

`CudaNtuples.shape`

The shape of this space.

CudaNtuples.size

`CudaNtuples.size`

The number of entries per tuple.

Methods

<code>__contains__(other)</code>	Return <i>other</i> in <i>self</i> .
<code>__eq__(other)</code>	Return <i>self</i> == <i>other</i> .
<code>contains_all(other)</code>	Test if all points in <i>other</i> are contained in this set.
<code>contains_set(other)</code>	Test if <i>other</i> is a subset of this set.
<code>element([inp, data_ptr])</code>	Create a new element.

CudaNtuples.__contains__

`CudaNtuples.__contains__` (*other*)

Return *other* in *self*.

Returns`contains` : bool

True if *other* is an *NtuplesBaseVector* instance and *other.space* is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

CudaNtuples.__eq__

CudaNtuples.__eq__(other)

Return self == other.

Returnsequals: bool

True if other is an instance of this space's type with the same *size* and *dtype*, otherwise False.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

CudaNtuples.contains_all

CudaNtuples.contains_all(other)

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

CudaNtuples.contains_set

CudaNtuples.contains_set(other)

Test if other is a subset of this set.

Implementing this method is optional. Default it tests for equality.

CudaNtuples.element`CudaNtuples.element` (*inp=None, data_ptr=None*)

Create a new element.

Parameters*inp* : *array-like* or scalar, optional

Input to initialize the new element.

If *inp* is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied (with broadcasting if necessary).

If a single value is given, it is copied to all entries.

If both *inp* and *data_ptr* are `None`, an empty element is created with no guarantee of its state (memory allocation only).

data_ptr : `int`, optional

Memory address of a CUDA array container

Cannot be combined with *inp*.**Return***element* : *CudaNtuplesVector*

The new element

Notes

This method preserves “array views” of correct size and type, see the examples below.

TODO: No, it does not yet!

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> x = uc3.element(np.array([1, 2, 3], dtype='uint8'))
>>> x
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> y = uc3.element([1, 2, 3])
>>> y
CudaNtuples(3, 'uint8').element([1, 2, 3])
```

__init__ (*size, dtype*)

Initialize a new instance.

Parameters*size* : `int`

The number entries per tuple

dtype : `object`

The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy’s internal datatype objects or as string.

Check `CUDA_DTYPES` for a list of available data types.

CudaNtuplesVector

class `odl.space.cu_ntuples.CudaNtuplesVector(space, data)`

Bases: `odl.space.base_ntuples.NtuplesBaseVector`, `odl.set.space.LinearSpaceVector`

Representation of a *CudaNtuples* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (. , self)
<i>data</i>	The data of this vector.
<i>data_ptr</i>	A raw pointer to the data of this vector.
<i>dtype</i>	Length of this vector, equal to space size.
<i>itemsize</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	Length of this vector, equal to space size.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>CudaNtuplesUFuncs</i> , access to numpy style ufuncs.

CudaNtuplesVector.T

`CudaNtuplesVector.T`

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

CudaNtuplesVector.data

`CudaNtuplesVector.data`

The data of this vector.

ParametersNone

Returns`ptr` : `CudaFnVectorImpl`

Underlying cuda data representation

CudaNtuplesVector.data_ptr`CudaNtuplesVector.data_ptr`

A raw pointer to the data of this vector.

CudaNtuplesVector.dtype`CudaNtuplesVector.dtype`

Length of this vector, equal to space size.

CudaNtuplesVector.itemsize`CudaNtuplesVector.itemsize`

The size in bytes on one element of this type.

CudaNtuplesVector.nbytes`CudaNtuplesVector.nbytes`

The number of bytes this vector uses in memory.

CudaNtuplesVector.ndim`CudaNtuplesVector.ndim`

Number of dimensions, always 1.

CudaNtuplesVector.shape`CudaNtuplesVector.shape`

Number of entries per axis, equals (size,) for linear storage.

CudaNtuplesVector.size`CudaNtuplesVector.size`

Length of this vector, equal to space size.

CudaNtuplesVector.space`CudaNtuplesVector.space`

Space to which this vector.

CudaNtuplesVector.ufunc`CudaNtuplesVector.ufunc`*CudaNtuplesUFuncs*, access to numpy style ufuncs.**See also:***odl.util.ufuncs.NtuplesBaseUFuncs* Base class for ufuncs in *NtuplesBase* spaces.**Notes**

Not all ufuncs are currently optimized, some use the default numpy implementation. This can be improved in the future.

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
CudaRn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
CudaRn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
CudaRn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
CudaRn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to <code>self</code> .
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, show, fig])</code>	Display the function graphically.

CudaNtuplesVector.__eq__

CudaNtuplesVector.__eq__(other)

Return `self == other`.

Return `sequals` : bool

True if all elements of `other` are equal to this vector's elements, False otherwise

Examples

```
>>> r3 = CudaNtuples(3, 'float32')
>>> x = r3.element([1, 2, 3])
>>> x == x
True
>>> y = r3.element([1, 2, 3])
>>> x == y
True
>>> y = r3.element([0, 0, 0])
>>> x == y
False
>>> r3_2 = CudaNtuples(3, 'uint8')
>>> z = r3_2.element([1, 2, 3])
>>> x != z
True
```

CudaNtuplesVector.__getitem__

CudaNtuplesVector.__getitem__(*indices*)

Access values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parameters*indices*: int or slice

The position(s) that should be accessed

Returns*values*: scalar or *CudaNtuplesVector*

The value(s) at the index (*indices*)

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0]
1
>>> z = y[1:3]
>>> z
CudaNtuples(2, 'uint8').element([2, 3])
>>> y[:2]
CudaNtuples(2, 'uint8').element([1, 3])
>>> y[::-1]
CudaNtuples(3, 'uint8').element([3, 2, 1])
```

The returned value is a view, modifications are reflected in the original data:

```
>>> z[:] = [4, 5]
>>> y
CudaNtuples(3, 'uint8').element([1, 4, 5])
```

CudaNtuplesVector.__setitem__

`CudaNtuplesVector.__setitem__(indices, values)`

Set values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parameters`indices` : int or slice

The position(s) that should be set

values : scalar, *array-like* or `CudaNtuplesVector`

The value(s) that are to be assigned.

If index is an int, value must be single value.

If index is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

Returns`None`

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 2, 3])
>>> y[1:3] = [7, 8]
>>> y
CudaNtuples(3, 'uint8').element([5, 7, 8])
>>> y[:] = np.array([0, 0, 0])
>>> y
CudaNtuples(3, 'uint8').element([0, 0, 0])
```

Scalar assignment

```
>>> y[:] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 5, 5])
```

`CudaNtuplesVector.asarray`

`CudaNtuplesVector.asarray(start=None, stop=None, step=None, out=None)`

Extract the data of this array as a numpy array.

Parameters`start` : int, optional

Start position. None means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : `numpy.ndarray`

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returns`asarray` : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y.asarray()
array([1, 2, 3], dtype=uint8)
>>> y.asarray(1, 3)
array([2, 3], dtype=uint8)
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='uint8')
>>> result = y.asarray(out=out)
>>> out
array([1, 2, 3], dtype=uint8)
>>> result is out
True
```

CudaNtuplesVector.assign

CudaNtuplesVector.**assign**(*other*)

Assign the values of *other* to self.

CudaNtuplesVector.copy

CudaNtuplesVector.**copy**()

Create an identical (deep) copy of this vector.

Returns*copy* : CudaNtuplesVector

The deep copy

Examples

```
>>> vec1 = CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> vec2 = vec1.copy()
>>> vec2
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> vec1 == vec2
True
>>> vec1 is vec2
False
```

CudaNtuplesVector.dist

CudaNtuplesVector.**dist**(*other*)

Distance to *other*.

LinearSpace.dist

CudaNtuplesVector.divide

`CudaNtuplesVector.divide(x, y)`
Divide by other inplace.

LinearSpace.divide

CudaNtuplesVector.inner

`CudaNtuplesVector.inner(other)`
Inner product with other.

LinearSpace.inner

CudaNtuplesVector.lincomb

`CudaNtuplesVector.lincomb(a, x1, b=None, x2=None)`
Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

CudaNtuplesVector.multiply

`CudaNtuplesVector.multiply(x, y)`
Multiply by other inplace.

LinearSpace.multiply

CudaNtuplesVector.norm

`CudaNtuplesVector.norm()`
Norm of vector

LinearSpace.norm

CudaNtuplesVector.set_zero

`CudaNtuplesVector.set_zero()`
Set this vector to zero.

LinearSpace.zero

CudaNtuplesVector.show

`CudaNtuplesVector.show(title=None, method='scatter', show=False, fig=None, **kwargs)`
Display the function graphically.

Parameter`title` : str, optional

Set the title of the figure

method : str, optional

1d methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.Figure

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.

kwargs : { ‘figsize’, ‘saveto’, ... }

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig` : `matplotlib.figure.Figure`

The resulting figure. It is also shown to the user.

See also:

`odl.util.graphics.show_discrete_data` Underlying implementation

`__init__` (*space*, *data*)
Initialize a new instance.

Functions

<code>CudaRn</code> (<i>size</i> [, <i>dtype</i>])	The real space R^n , implemented in CUDA.
<code>cu_weighted_dist</code> (<i>weight</i> [, <i>exponent</i>])	Weighted distance on <code>CudaFn</code> spaces as free function.
<code>cu_weighted_inner</code> (<i>weight</i>)	Weighted inner product on <code>CudaFn</code> spaces as free function.
<code>cu_weighted_norm</code> (<i>weight</i> [, <i>exponent</i>])	Weighted norm on <code>CudaFn</code> spaces as free function.

CudaRn

`odl.space.cu_ntuples.CudaRn` (*size*, *dtype*=‘float32’, ***kwargs*)

The real space R^n , implemented in CUDA.

Requires the compiled ODL extension `odlpp`.

Parameters`size` : positive `int`

The number of dimensions of the space

dtype : optional

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy’s internal datatype objects or as string.

Only real floating-point data types are allowed.

kwargs : { ‘weight’, ‘exponent’, ‘dist’, ‘norm’, ‘inner’ }

See `CudaFn`

cu_weighted_dist

`odl.space.cu_ntuples.cu_weighted_dist` (*weight*, *exponent*=2.0)

Weighted distance on `CudaFn` spaces as free function.

Parameters`weight` : scalar, *array-like* or `CudaFnVector`

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

exponent : positive float

Exponent of the distance

Returns**dist** : callable

Distance function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, CudaFnVectorWeighting

cu_weighted_inner

`odl.space.cu_ntuples.cu_weighted_inner(weight)`

Weighted inner product on *CudaFn* spaces as free function.

Parameters**weight** : scalar, *array-like* or *CudaFnVector*

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

Returns**inner** : callable

Inner product function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, CudaFnVectorWeighting

cu_weighted_norm

`odl.space.cu_ntuples.cu_weighted_norm(weight, exponent=2.0)`

Weighted norm on *CudaFn* spaces as free function.

Parameters**weight** : scalar, *array-like* or *CudaFnVector*

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

exponent : positive float

Exponent of the norm. If *weight* is a sparse matrix, only 1.0, 2.0 and *inf* are allowed.

Returns**norm** : callable

Norm function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, CudaFnVectorWeighting

fspace

Spaces of functions with common domain and range.

Classes

<i>FunctionSet</i> (domain, range[, out_dtype])	A general set of functions with common domain and range.
<i>FunctionSetVector</i> (fset, fcall[, out_dtype])	Representation of a <i>FunctionSet</i> element.
<i>FunctionSpace</i> (domain[, field, out_dtype])	A vector space of functions.
<i>FunctionSpaceVector</i> (fspace, fcall)	Representation of a <i>FunctionSpace</i> element.

FunctionSet

class odl.space.fspace.**FunctionSet** (domain, range, out_dtype=None)

Bases: *odl.set.sets.Set*

A general set of functions with common domain and range.

Attributes

<i>domain</i>	Common domain of all functions in this set.
<i>element_type</i>	<i>FunctionSetVector</i>
<i>out_dtype</i>	Output data type of functions in this space.
<i>range</i>	Common range of all functions in this set.

FunctionSet.domain

FunctionSet.**domain**

Common domain of all functions in this set.

FunctionSet.element_type

FunctionSet.**element_type**

FunctionSetVector

FunctionSet.out_dtype

FunctionSet.**out_dtype**

Output data type of functions in this space.

FunctionSet.range

FunctionSet.**range**

Common range of all functions in this set.

Methods

<i>__contains__</i> (other)	Return other in self.
<i>__eq__</i> (other)	Return self == other.
<i>contains_all</i> (other)	Test if all points in other are contained in this set.
<i>contains_set</i> (other)	Test if other is a subset of this set.
<i>element</i> ([fcall, vectorized])	Create a <i>FunctionSet</i> element.

FunctionSet.__contains__

`FunctionSet.__contains__(other)`

Return other in self.

Returnsequals : bool

True if other is a `FunctionSetVector` whose `FunctionSetVector.space` attribute equals this space, False otherwise.

FunctionSet.__eq__

`FunctionSet.__eq__(other)`

Return self == other.

Returnsequals : bool

True if other is a `FunctionSet` with same `FunctionSet.domain` and `FunctionSet.range`, False otherwise.

FunctionSet.contains_all

`FunctionSet.contains_all(other)`

Test if all points in other are contained in this set.

This is a default implementation and should be overridden by subclasses.

FunctionSet.contains_set

`FunctionSet.contains_set(other)`

Test if other is a subset of this set.

Implementing this method is optional. Default it tests for equality.

FunctionSet.element

`FunctionSet.element(fcall=None, vectorized=True)`

Create a `FunctionSet` element.

Parameters`fcall` : callable, optional

The actual instruction for out-of-place evaluation. It must return an `FunctionSet.range` element or a `numpy.ndarray` of such (vectorized call).

vectorized : bool

Whether `fcall` supports vectorized evaluation.

Returnselement : `FunctionSetVector`

The new element, always supports vectorization

See also:

`odl.discr.grid.TensorGrid.meshgrid` efficient grids for function evaluation

`__init__(domain, range, out_dtype=None)`

Initialize a new instance.

Parameters`domain` : `Set`

The domain of the functions.

range : `Set`

The range of the functions.

out_dtype : optional

Data type of the return value of a function in this space. Can be given in any way `numpy.dtype` understands, e.g. as string ('bool') or data type (`bool`). If no data type is given, a “lazy” evaluation is applied, i.e. an adequate data type is inferred during function evaluation.

FunctionSetVector

class `odl.space.fspace.FunctionSetVector` (*fset, fcall, out_dtype=None*)

Bases: `odl.operator.operator.Operator`

Representation of a *FunctionSet* element.

Attributes

<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.
<i>space</i>	The space or set this function belongs to.

FunctionSetVector.adjoint

`FunctionSetVector.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSetVector.domain

`FunctionSetVector.domain`

Set of objects on which this operator can be evaluated.

FunctionSetVector.inverse

`FunctionSetVector.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSetVector.is_functional

`FunctionSetVector.is_functional`

True if the this operator’s range is a *Field*.

FunctionSetVector.is_linear

FunctionSetVector.**is_linear**

True if this operator is linear.

FunctionSetVector.range

FunctionSetVector.**range**

Set in which the result of an evaluation of this operator lies.

FunctionSetVector.space

FunctionSetVector.**space**

The space or set this function belongs to.

Methods

<code>__call__(x[, out])</code>	Return self(x[, out, **kwargs]).
<code>__eq__(other)</code>	Returns vec == other.
<code>__call__(x[, out])</code>	Raw evaluation method.
<code>assign(other)</code>	Assign other to this vector.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>derivative(point)</code>	Return the operator derivative at point.

FunctionSetVector.__call__

FunctionSetVector.**__call__**(x, out=None, **kwargs)

Return self(x[, out, **kwargs]).

Parametersx : domain *element-like*, *meshgrid* or `numpy.ndarray`

Input argument for the function evaluation. Conditions on x depend on its type:

element-like: must be a castable to a domain element

meshgrid: length must be `space.ndim`, and the arrays must be broadcastable against each other.

array: shape must be (d, N), where d is the number of dimensions of the function domain

out : `numpy.ndarray`, optional

Output argument holding the result of the function evaluation, can only be used for vectorized functions. Its shape must be equal to `np.broadcast(*x).shape`.

bounds_check : bool

If True, check if all input points lie in the function domain in the case of vectorized evaluation. This requires the domain to implement *Set.contains_all*. Default: True

Returnsout : range element or array of elements

Result of the function evaluation. If out was provided, the returned object is a reference to it.

RaisesTypeError

If x is not a valid vectorized evaluation argument

If `out` is not a range element or a `numpy.ndarray` of range elements

ValueError

If evaluation points fall outside the valid domain

FunctionSetVector.__eq__

`FunctionSetVector.__eq__(other)`

Returns `vec == other`.

Returnsequals : bool

True if `other` is a `FunctionSetVector` with `other.space` equal to this vector's space and evaluation function of `other` and this vector is equal. False otherwise.

FunctionSetVector._call

`FunctionSetVector._call(x, out=None, **kwargs)`

Raw evaluation method.

FunctionSetVector.assign

`FunctionSetVector.assign(other)`

Assign `other` to this vector.

This is implemented without `FunctionSpace.lincomb` to ensure that `vec == other` evaluates to True after `vec.assign(other)`.

FunctionSetVector.copy

`FunctionSetVector.copy()`

Create an identical (deep) copy of this vector.

FunctionSetVector.derivative

`FunctionSetVector.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*fset, fcall, out_dtype=None*)

Initialize a new instance.

Parameters*fset* : `FunctionSet`

The set of functions this element lives in

fcall : callable

The actual instruction for out-of-place evaluation. It must return an `FunctionSet.range` element or a `numpy.ndarray` of such (vectorized call).

out_d

FunctionSpace

class `odl.space.fspace.FunctionSpace` (*domain*, *field=None*, *out_dtype=None*)
Bases: `odl.space.fspace.FunctionSet`, `odl.set.space.LinearSpace`

A vector space of functions.

Attributes

<i>domain</i>	Common domain of all functions in this set.
<i>element_type</i>	<i>FunctionSpaceVector</i>
<i>field</i>	The field of this vector space.
<i>out_dtype</i>	Output data type of functions in this space.
<i>range</i>	Common range of all functions in this set.

FunctionSpace.domain

`FunctionSpace.domain`
Common domain of all functions in this set.

FunctionSpace.element_type

`FunctionSpace.element_type`
FunctionSpaceVector

FunctionSpace.field

`FunctionSpace.field`
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns`field` : *Field*

The underlying field.

FunctionSpace.out_dtype

`FunctionSpace.out_dtype`
Output data type of functions in this space.

FunctionSpace.range

`FunctionSpace.range`
Common range of all functions in this set.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Returns <code>s == other</code> .
<code>_dist(x1, x2)</code>	Calculate the distance between <code>x1</code> and <code>x2</code> .
<code>_divide(x1, x2, out)</code>	Raw pointwise division of two functions.

Continued on next page

Table 2.195 – continued from previous page

<code>_inner(x1, x2)</code>	Calculate the inner product of <code>x1</code> and <code>x2</code> .
<code>_lincomb(a, x1, b, x2, out)</code>	Raw linear combination of <code>x1</code> and <code>x2</code> .
<code>_multiply(x1, x2, out)</code>	Raw pointwise multiplication of two functions.
<code>_norm(x)</code>	Calculate the norm of <code>x</code> .
<code>astype(out_dtype)</code>	Return a copy of this space with new <code>out_dtype</code> .
<code>contains_all(other)</code>	Test if all points in <code>other</code> are contained in this set.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of this set.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of <code>x1</code> and <code>x2</code>
<code>element([fcall, vectorized])</code>	Create a <i>FunctionSpace</i> element.
<code>inner(x1, x2)</code>	Calculate the inner product of <code>x1</code> and <code>x2</code> .
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of <code>x1</code> and <code>x2</code> .
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	The function mapping everything to one.
<code>zero()</code>	The function mapping everything to zero.

FunctionSpace.__contains__

FunctionSpace.__contains__(other)

Return other in self.

Returnsequals : bool

True if other is a *FunctionSetVector* whose *FunctionSetVector.space* attribute equals this space, False otherwise.

FunctionSpace.__eq__

FunctionSpace.__eq__(other)

Returns s == other.

Returnsequals : bool

True if other is a *FunctionSpace* with same *FunctionSpace.domain* and *FunctionSpace.range*, False otherwise.

FunctionSpace._dist

FunctionSpace._dist(x1, x2)

Calculate the distance between `x1` and `x2`.

This method is intended to be private, public callers should resort to *dist* which is type-checked.

FunctionSpace._divide

FunctionSpace._divide(x1, x2, out)

Raw pointwise division of two functions.

FunctionSpace._inner

FunctionSpace._inner(x1, x2)

Calculate the inner product of `x1` and `x2`.

This method is intended to be private, public callers should resort to *inner* which is type-checked.

FunctionSpace.lincomb

FunctionSpace.**lincomb**(*a*, *x1*, *b*, *x2*, *out*)

Raw linear combination of *x1* and *x2*.

Notes

The additions and multiplications are implemented via simple Python functions, so non-vectorized versions are slow.

FunctionSpace.multiply

FunctionSpace.**multiply**(*x1*, *x2*, *out*)

Raw pointwise multiplication of two functions.

Notes

The multiplication is implemented with a simple Python function, so the non-vectorized versions are slow.

FunctionSpace.norm

FunctionSpace.**norm**(*x*)

Calculate the norm of *x*.

This method is intended to be private, public callers should resort to *norm* which is type-checked.

FunctionSpace.astype

FunctionSpace.**astype**(*out_dtype*)

Return a copy of this space with new *out_dtype*.

Parameters*out_dtype* : optional

Output data type of the returned space. Can be given in any way `numpy.dtype` understands, e.g. as string ('complex64') or data type (`complex`). None is interpreted as 'float64'.

Returns*newspace* : *FunctionSpace*

The version of this space with given data type

FunctionSpace.contains_all

FunctionSpace.**contains_all**(*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

FunctionSpace.contains_set

FunctionSpace.**contains_set**(*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

FunctionSpace.distFunctionSpace.**dist** (*x1*, *x2*)

Calculate the distance between two vectors.

Parameters*x1*, *x2* : *LinearSpaceVector*

Vectors whose distance to compute

Returns*dist* : float

Distance between vectors

FunctionSpace.divideFunctionSpace.**divide** (*x1*, *x2*, *out=None*)Calculate the pointwise division of *x1* and *x2***Parameters***x1* : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns*out* : *LinearSpaceVector*Ratio of the vectors. If *out* was provided, the returned object is a reference to it.**FunctionSpace.element**FunctionSpace.**element** (*fcall=None*, *vectorized=True*)Create a *FunctionSpace* element.**Parameters***fcall* : callable, optionalThe actual instruction for out-of-place evaluation. It must return an *FunctionSet.range* element or a `numpy.ndarray` of such (vectorized call).If *fcall* is a *FunctionSetVector*, it is wrapped as a new *FunctionSpaceVector*.**vectorized** : boolWhether *fcall* supports vectorized evaluation.**Return***element* : *FunctionSpaceVector*

The new element, always supports vectorization

Notes

If you specify `vectorized=False`, the function is decorated with a vectorizer, which makes two elements created this way from the same function being regarded as *not equal*.

FunctionSpace.inner`FunctionSpace.inner(x1, x2)`

Calculate the inner product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Factors in the inner product

Returns`out` : *LinearSpace.field* element

Product of the vectors. If `out` was provided, the returned object is a reference to it.

FunctionSpace.lincomb`FunctionSpace.lincomb(a, x1, b=None, x2=None, out=None)`

Linear combination of vectors.

Calculates

`out = a * x1`

or, if `b` and `y` are given,

`out = a*x1 + b*x2`

with error checking of types.

Parameters`a` : Scalar in the field of this space

Scalar to multiply `x1` with.

`x1` : *LinearSpaceVector*

The first of the summands

`b` : Scalar, optional

Scalar to multiply `x2` with.

`x2` : *LinearSpaceVector*, optional

The second of the summands

`out` : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns`out` : *LinearSpaceVector*

Result of the linear combination. If `out` was provided, the returned object is a reference to it.

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

`space.lincomb(x, 2, x, 3.14, out=x)`

is (mathematically) equivalent to

`x = x * (1 + 2 + 3.14)`

FunctionSpace.multiply

`FunctionSpace.multiply(x1, x2, out=None)`

Calculate the pointwise product of `x1` and `x2`.

Parameters`x1, x2` : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns`out` : *LinearSpaceVector*

Product of the vectors. If `out` was provided, the returned object is a reference to it.

FunctionSpace.norm

`FunctionSpace.norm(x)`

Calculate the norm of a vector.

Parameters`x` : *LinearSpaceVector*

The vector

Returns`out` : float

Norm of the vector

FunctionSpace.one

`FunctionSpace.one()`

The function mapping everything to one.

This function is the multiplicative unit in the function space.

FunctionSpace.zero

`FunctionSpace.zero()`

The function mapping everything to zero.

This function is the additive unit in the function space.

Since *FunctionSpace.lincomb* may be slow, we implement this function directly.

__init__(*domain, field=None, out_dtype=None*)

Initialize a new instance.

Parameters`domain` : *Set*

The domain of the functions

field : *Field*, optional

The range of the functions, usually the *RealNumbers* or *ComplexNumbers*. If not given, the field is either inferred from `out_dtype`, or, if the latter is also `None`, set to `RealNumbers()`.

out_dtype : optional

Data type of the return value of a function in this space. Can be given in any way `numpy.dtype` understands, e.g. as string ('float64') or data type (`float`). By default, 'float64' is used for real and 'complex128' for complex spaces.

FunctionSpaceVector

class odl.space.fspace.**FunctionSpaceVector** (*fspace*, *fcall*)

Bases: *odl.set.space.LinearSpaceVector*, *odl.space.fspace.FunctionSetVector*

Representation of a *FunctionSpace* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (.
<i>adjoint</i>	The operator adjoint (abstract).
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>imag</i>	Pointwise imaginary part of this function.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.
<i>real</i>	Pointwise real part of this function.
<i>space</i>	Space to which this vector belongs.

FunctionSpaceVector.T

FunctionSpaceVector.**T**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

FunctionSpaceVector.adjoint

FunctionSpaceVector.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSpaceVector.domain`FunctionSpaceVector.domain`

Set of objects on which this operator can be evaluated.

FunctionSpaceVector.imag`FunctionSpaceVector.imag`

Pointwise imaginary part of this function.

FunctionSpaceVector.inverse`FunctionSpaceVector.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSpaceVector.is_functional`FunctionSpaceVector.is_functional`True if the this operator's range is a *Field*.**FunctionSpaceVector.is_linear**`FunctionSpaceVector.is_linear`

True if this operator is linear.

FunctionSpaceVector.range`FunctionSpaceVector.range`

Set in which the result of an evaluation of this operator lies.

FunctionSpaceVector.real`FunctionSpaceVector.real`

Pointwise real part of this function.

FunctionSpaceVector.space`FunctionSpaceVector.space`

Space to which this vector belongs.

*LinearSpace***Methods**

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__(other)</code>	Returns <code>vec == other</code> .
<code>_call(x[, out])</code>	Raw evaluation method.
<code>assign(other)</code>	Assign <code>other</code> to this vector.
<code>conj()</code>	Pointwise complex conjugate of this function.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .
<code>dist(other)</code>	Distance to <code>other</code> .

Continued on next page

Table 2.197 – continued from previous page

<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.

FunctionSpaceVector.__call__`FunctionSpaceVector.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.**Parameters**`x` : domain *element-like*, *meshgrid* or `numpy.ndarray`Input argument for the function evaluation. Conditions on `x` depend on its type:

element-like: must be a castable to a domain element

meshgrid: length must be `space.ndim`, and the arrays must be broadcastable against each other.array: shape must be `(d, N)`, where `d` is the number of dimensions of the function domain**out** : `numpy.ndarray`, optionalOutput argument holding the result of the function evaluation, can only be used for vectorized functions. Its shape must be equal to `np.broadcast(*x).shape`.**bounds_check** : `bool`If `True`, check if all input points lie in the function domain in the case of vectorized evaluation. This requires the domain to implement *Set.contains_all*. Default: `True`**Returns**`out` : range element or array of elementsResult of the function evaluation. If `out` was provided, the returned object is a reference to it.**RaisesTypeError**If `x` is not a valid vectorized evaluation argumentIf `out` is not a range element or a `numpy.ndarray` of range elements**ValueError**

If evaluation points fall outside the valid domain

FunctionSpaceVector.__eq__`FunctionSpaceVector.__eq__(other)`Returns `vec == other`.**Return**`equals` : `bool``True` if `other` is a *FunctionSetVector* with `other.space` equal to this vector's space and evaluation function of `other` and this vector is equal. `False` otherwise.

FunctionSpaceVector._call

`FunctionSpaceVector._call(x, out=None, **kwargs)`

Raw evaluation method.

FunctionSpaceVector.assign

`FunctionSpaceVector.assign(other)`

Assign other to this vector.

This is implemented without `FunctionSpace.lincomb` to ensure that `vec == other` evaluates to `True` after `vec.assign(other)`.

FunctionSpaceVector.conj

`FunctionSpaceVector.conj()`

Pointwise complex conjugate of this function.

FunctionSpaceVector.copy

`FunctionSpaceVector.copy()`

Create an identical (deep) copy of this vector.

FunctionSpaceVector.derivative

`FunctionSpaceVector.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

FunctionSpaceVector.dist

`FunctionSpaceVector.dist(other)`

Distance to other.

LinearSpace.dist

FunctionSpaceVector.divide

`FunctionSpaceVector.divide(x, y)`

Divide by other inplace.

LinearSpace.divide

FunctionSpaceVector.inner

`FunctionSpaceVector.inner(other)`

Inner product with other.

LinearSpace.inner

FunctionSpaceVector.lincomb

`FunctionSpaceVector.lincomb(a, x1, b=None, x2=None)`

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

FunctionSpaceVector.multiply`FunctionSpaceVector.multiply(x, y)`

Multiply by other inplace.

*LinearSpace.multiply***FunctionSpaceVector.norm**`FunctionSpaceVector.norm()`

Norm of vector

*LinearSpace.norm***FunctionSpaceVector.set_zero**`FunctionSpaceVector.set_zero()`

Set this vector to zero.

LinearSpace.zero`__init__(fspace, fcall)`

Initialize a new instance.

Parameters`fspace` : *FunctionSpace*

The set of functions this element lives in

fcall : callable

The actual instruction for out-of-place evaluation. It must return an *FunctionSet.range* element or a `numpy.ndarray` of such (vectorized call).

ntuples

CPU implementations of n -dimensional Cartesian spaces.

This is a default implementation of A^n for an arbitrary set A as well as the real and complex spaces R^n and C^n . The data is represented by NumPy arrays.

Classes

<i>Fn</i> (size, dtype, **kwargs)	The vector space F^n with vector multiplication.
<i>FnConstWeighting</i> (constant[, exponent, ...])	Weighting of <i>Fn</i> by a constant.
<i>FnCustomDist</i> (dist)	Custom distance on <i>Fn</i> , removes norm and inner.
<i>FnCustomInnerProduct</i> (inner[, dist_using_inner])	Custom inner product on <i>Fn</i> .
<i>FnCustomNorm</i> (norm)	Custom norm on <i>Fn</i> , removes inner.
<i>FnMatrixWeighting</i> (matrix[, exponent, ...])	Matrix weighting for <i>Fn</i> .
<i>FnNoWeighting</i> ([exponent, dist_using_inner])	Weighting of <i>Fn</i> with constant 1.
<i>FnVector</i> (space, data)	Representation of an <i>Fn</i> element.
<i>FnVectorWeighting</i> (vector[, exponent, ...])	Vector weighting for <i>Fn</i> .
<i>MatVecOperator</i> (matrix[, dom, ran])	Matrix multiply operator $\mathbb{F}^n \rightarrow \mathbb{F}^m$.
<i>Ntuples</i> (size, dtype)	The set of n -tuples of arbitrary type.
<i>NtuplesVector</i> (space, data)	Representation of an <i>Ntuples</i> element.

Fn

class `odl.space.ntuples.Fn` (*size*, *dtype*, ***kwargs*)

Bases: `odl.space.base_ntuples.FnBase`, `odl.space.ntuples.Ntuples`

The vector space F^n with vector multiplication.

This space implements n-tuples of elements from a *Field* F , which is usually the real or complex numbers.

Its elements are represented as instances of the *FnVector* class.

Attributes

<i>dtype</i>	The data type of each entry.
<i>element_type</i>	Return <i>FnVector</i> .
<i>exponent</i>	Exponent of the norm and distance.
<i>field</i>	The field of this vector space.
<i>is_cn</i>	Return <code>True</code> if the space represents C^n , i.e.
<i>is_rn</i>	Return <code>True</code> if the space represents R^n , i.e.
<i>is_weighted</i>	Return <code>True</code> if the weighting is not <i>FnNoWeighting</i> .
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.
<i>weighting</i>	This space's weighting scheme.

Fn.dtype

`Fn.dtype`

The data type of each entry.

Fn.element_type

`Fn.element_type`

Return *FnVector*.

Fn.exponent

`Fn.exponent`

Exponent of the norm and distance.

Fn.field

`Fn.field`

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns`field : Field`

The underlying field.

Fn.is_cn

`Fn.is_cn`

Return `True` if the space represents C^n , i.e. complex tuples.

Fn.is_rn**Fn.is_rn**Return True if the space represents R^n , i.e. real tuples.**Fn.is_weighted****Fn.is_weighted**Return True if the weighting is not *FnNoWeighting*.**Fn.shape****Fn.shape**

The shape of this space.

Fn.size**Fn.size**

The number of entries per tuple.

Fn.weighting**Fn.weighting**

This space's weighting scheme.

Methods

<code>__contains__(other)</code>	Return other in self.
<code>__eq__(other)</code>	Return self == other.
<code>_dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>_divide(x1, x2, out)</code>	The entry-wise division of two vectors, assigned to out.
<code>_inner(x1, x2)</code>	Raw inner product of two vectors.
<code>_lincomb(a, x1, b, x2, out)</code>	Linear combination of x1 and x2.
<code>_multiply(x1, x2, out)</code>	The entry-wise product of two vectors, assigned to out.
<code>_norm(x)</code>	Calculate the norm of a vector.
<code>astype(dtype)</code>	Return a copy of this space with new dtype.
<code>contains_all(other)</code>	Test if all points in other are contained in this set.
<code>contains_set(other)</code>	Test if other is a subset of this set.
<code>default_dtype(field)</code>	Return the default of <i>Fn</i> data type for a given field.
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>divide(x1, x2[, out])</code>	Calculate the pointwise division of x1 and x2
<code>element([inp, data_ptr])</code>	Create a new element.
<code>inner(x1, x2)</code>	Calculate the inner product of x1 and x2.
<code>lincomb(a, x1[, b, x2, out])</code>	Linear combination of vectors.
<code>multiply(x1, x2[, out])</code>	Calculate the pointwise product of x1 and x2.
<code>norm(x)</code>	Calculate the norm of a vector.
<code>one()</code>	Create a vector of ones.
<code>zero()</code>	Create a vector of zeros.

Fn.__contains__**Fn.__contains__** (*other*)

Return other in self.

Returnscontains : bool

True if other is an *NtuplesBaseVector* instance and other.space is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

Fn.__eq__

Fn.__eq__(other)
Return self == other.

Returnsequals: bool

True if other is an instance of this space's type with the same *NtuplesBase.size* and *NtuplesBase.dtype*, and identical distance function, otherwise False.

Examples

```
>>> from numpy.linalg import norm
>>> def dist(x, y, ord):
...     return norm(x - y, ord)
```

```
>>> from functools import partial
>>> dist2 = partial(dist, ord=2)
>>> c3 = Cn(3, dist=dist2)
>>> c3_same = Cn(3, dist=dist2)
>>> c3 == c3_same
True
```

Different dist functions result in different spaces - the same applies for norm and inner:

```
>>> dist1 = partial(dist, ord=1)
>>> c3_1 = Cn(3, dist=dist1)
>>> c3_2 = Cn(3, dist=dist2)
>>> c3_1 == c3_2
False
```

Be careful with Lambdas - they result in non-identical function objects:

```
>>> c3_lambda1 = Cn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> c3_lambda2 = Cn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> c3_lambda1 == c3_lambda2
False
```

An *Fn* space with the same data type is considered equal:

```
>>> c3 = Cn(3)
>>> f3_cdoube = Fn(3, dtype='complex128')
```

```
>>> c3 == f3_cdoube
True
```

Fn._dist**Fn._dist** (*x1*, *x2*)

Calculate the distance between two vectors.

Parameters*x1*, *x2* : *FnVector*

Vectors whose mutual distance is calculated

Returns*dist* : float

Distance between the vectors

Examples

```
>>> from numpy.linalg import norm
>>> c2_2 = Cn(2, dist=lambda x, y: norm(x - y, ord=2))
>>> x = c2_2.element([3+1j, 4])
>>> y = c2_2.element([1j, 4-4j])
>>> c2_2.dist(x, y)
5.0
```

```
>>> c2_2 = Cn(2, dist=lambda x, y: norm(x - y, ord=1))
>>> x = c2_2.element([3+1j, 4])
>>> y = c2_2.element([1j, 4-4j])
>>> c2_2.dist(x, y)
7.0
```

Fn._divide**Fn._divide** (*x1*, *x2*, *out*)

The entry-wise division of two vectors, assigned to out.

Parameters*x1*, *x2* : *FnVector*

Dividend and divisor in the quotient

out : *FnVector*

Vector to which the result is written

ReturnsNone**Examples**

```
>>> r3 = Rn(3)
>>> x = r3.element([3, 5, 6])
>>> y = r3.element([1, 2, 2])
>>> out = r3.element()
>>> r3.divide(x, y, out) # out is returned
Rn(3).element([3.0, 2.5, 3.0])
>>> out
Rn(3).element([3.0, 2.5, 3.0])
```

Fn.inner**Fn._inner** (*x1*, *x2*)

Raw inner product of two vectors.

Parameters*x1*, *x2* : *FnVector*

The vectors whose inner product is calculated

Returns*inner* : *field element*

Inner product of the vectors

Examples

```
>>> import numpy as np
>>> c3 = Cn(2, inner=lambda x, y: np.vdot(y, x))
>>> x = c3.element([5+1j, -2j])
>>> y = c3.element([1, 1+1j])
>>> c3.inner(x, y) == (5+1j)*1 + (-2j)*(1-1j)
True
```

Define a space with custom inner product:

```
>>> weights = np.array([1., 2.])
>>> def weighted_inner(x, y):
...     return np.vdot(weights * y.data, x.data)
```

```
>>> c3w = Cn(2, inner=weighted_inner)
>>> x = c3w.element(x) # elements must be cast (no copy)
>>> y = c3w.element(y)
>>> c3w.inner(x, y) == 1*(5+1j)*1 + 2*(-2j)*(1-1j)
True
```

Fn.lincomb**Fn._lincomb** (*a*, *x1*, *b*, *x2*, *out*)Linear combination of *x1* and *x2*.Calculate $out = a*x1 + b*x2$ using optimized BLAS routines if possible.**Parameters***a*, *b* : *FnBase.field*Scalars to multiply *x1* and *x2* with**x1**, **x2** : *FnVector*

Summands in the linear combination

out : *FnVector*

Vector to which the result is written

Returns*None***Examples**

```
>>> c3 = Cn(3)
>>> x = c3.element([1+1j, 2-1j, 3])
>>> y = c3.element([4+0j, 5, 6+0.5j])
>>> out = c3.element()
```

```
>>> c3.lincomb(2j, x, 3-1j, y, out) # out is returned
Cn(3).element([(10-2j), (17-1j), (18.5+1.5j)])
>>> out
Cn(3).element([(10-2j), (17-1j), (18.5+1.5j)])
```

Fn._multiply

Fn._multiply(x1, x2, out)

The entry-wise product of two vectors, assigned to out.

Parametersx1, x2 : *FnVector*

Factors in the product

out : *FnVector*

Vector to which the result is written

ReturnsNone

Examples

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> y = c3.element([1, 2+1j, 3-1j])
>>> out = c3.element()
>>> c3.multiply(x, y, out) # out is returned
Cn(3).element([(5+1j), (6+3j), (4-8j)])
>>> out
Cn(3).element([(5+1j), (6+3j), (4-8j)])
```

Fn._norm

Fn._norm(x)

Calculate the norm of a vector.

Parametersx : *FnVector*

The vector whose norm is calculated

Returnsnorm : float

Norm of the vector

Examples

```
>>> import numpy as np
>>> c2_2 = Cn(2, norm=np.linalg.norm) # 2-norm
>>> x = c2_2.element([3+1j, 1-5j])
>>> c2_2.norm(x)
6.0
```

```
>>> from functools import partial
>>> c2_1 = Cn(2, norm=partial(np.linalg.norm, ord=1))
>>> x = c2_1.element([3-4j, 12+5j])
>>> c2_1.norm(x)
18.0
```


Fn.astype**Fn.astype** (*dtype*)

Return a copy of this space with new dtype.

Parametersdtype :Data type of the returned space. Can be given in any way `numpy.dtype` understands, e.g. as string ('complex64') or data type (`complex`).**Returnsnewspace :** *FnBase*

The version of this space with given data type

Fn.contains_all**Fn.contains_all** (*other*)Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

Fn.contains_set**Fn.contains_set** (*other*)Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Fn.default_dtype**static Fn.default_dtype** (*field*)Return the default of *Fn* data type for a given field.**Parametersfield :** *Field*Set of numbers to be represented by a data type. Currently supported : *RealNumbers*, *ComplexNumbers***Returnsdtype :** *type*

Numpy data type specifier. The returned defaults are:

`RealNumbers() : np.dtype('float64')``ComplexNumbers() : np.dtype('complex128')`**Fn.dist****Fn.dist** (*x1*, *x2*)

Calculate the distance between two vectors.

Parametersx1, x2 : *LinearSpaceVector*

Vectors whose distance to compute

Returnsdist : *float*

Distance between vectors

Fn.divide**Fn.divide** (*x1*, *x2*, *out=None*)Calculate the pointwise division of *x1* and *x2***Parametersx1 :** *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to

Returns **out** : *LinearSpaceVector*

Ratio of the vectors. If **out** was provided, the returned object is a reference to it.

Fn.element

Fn.element (*inp=None, data_ptr=None*)

Create a new element.

Parameters **inp** : *array-like*, optional

Input to initialize the new element.

If **inp** is *None*, an empty element is created with no guarantee of its state (memory allocation only).

If **inp** is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied.

Returns **element** : *NtuplesVector*

The new element created (from **inp**).

Notes

This method preserves “array views” of correct size and type, see the examples below.

Examples

```
>>> strings3 = Ntuples(3, dtype='U1') # 1-char strings
>>> x = strings3.element(['w', 'b', 'w'])
>>> print(x)
[w, b, w]
>>> x.space
Ntuples(3, '<U1')
```

Construction from data pointer:

```
>>> int3 = Ntuples(3, dtype='int')
>>> x = int3.element([1, 2, 3])
>>> y = int3.element(data_ptr=x.data_ptr)
>>> print(y)
[1, 2, 3]
>>> y[0] = 5
>>> print(x)
[5, 2, 3]
```

Fn.inner

Fn.**inner** (*x1*, *x2*)

Calculate the inner product of *x1* and *x2*.

Parameters*x1*, *x2* : *LinearSpaceVector*

Factors in the inner product

Returns*out* : *LinearSpace.field* element

Product of the vectors. If *out* was provided, the returned object is a reference to it.

Fn.lincomb

Fn.**lincomb** (*a*, *x1*, *b=None*, *x2=None*, *out=None*)

Linear combination of vectors.

Calculates

$out = a * x1$

or, if *b* and *y* are given,

$out = a*x1 + b*x2$

with error checking of types.

Parameters*a* : Scalar in the field of this space

Scalar to multiply *x1* with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply *x2* with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returns*out* : *LinearSpaceVector*

Result of the linear combination. If *out* was provided, the returned object is a reference to it.

Notes

The vectors *out*, *x1* and *x2* may be aligned, thus a call

`space.lincomb(x, 2, x, 3.14, out=x)`

is (mathematically) equivalent to

$x = x * (1 + 2 + 3.14)$

Fn.multiply**Fn.multiply** (*x1*, *x2*, *out=None*)Calculate the pointwise product of *x1* and *x2*.**Parameters***x1*, *x2* : *LinearSpaceVector*

Multiplicands in the product

out : *LinearSpaceVector*, optional

Vector to write the product to

Returns*out* : *LinearSpaceVector*Product of the vectors. If *out* was provided, the returned object is a reference to it.**Fn.norm****Fn.norm** (*x*)

Calculate the norm of a vector.

Parameters*x* : *LinearSpaceVector*

The vector

Returns*out* : float

Norm of the vector

Fn.one**Fn.one** ()

Create a vector of ones.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.one()
>>> x
Cn(3).element([(1+0j), (1+0j), (1+0j)])
```

Fn.zero**Fn.zero** ()

Create a vector of zeros.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.zero()
>>> x
Cn(3).element([0j, 0j, 0j])
```

__init__ (*size*, *dtype*, ***kwargs*)

Initialize a new instance.

Parameters*size* : positive int

The number of dimensions of the space

dtype : object

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as `numpy.dtype` or as `string`.

Only scalar data types are allowed.

weight : optional

Use weighted inner product, norm, and dist. The following types are supported as `weight`:

FnWeightingBase: Use this weighting as-is. Compatibility with this space's elements is not checked during init.

float: Weighting by a constant

array-like: Weighting by a matrix (2-dim. array) or a vector (1-dim. array, corresponds to a diagonal matrix). A matrix can also be given as a sparse matrix (`scipy.sparse.spmatrix`).

Default: no weighting

This option cannot be combined with `dist`, `norm` or `inner`.

exponent : positive float, optional

Exponent of the norm. For values other than 2.0, no inner product is defined. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

This option is ignored if `dist`, `norm` or `inner` is given.

Default: 2.0

dist : callable, optional

The distance function defining a metric on \mathbb{F}^n . It must accept two *FnVector* arguments and fulfill the following mathematical conditions for any three vectors x, y, z :

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

By default, `dist(x, y)` is calculated as `norm(x - y)`. This creates an intermediate array `x-y`, which can be avoided by choosing `dist_using_inner=True`.

This option cannot be combined with `weight`, `norm` or `inner`.

norm : callable, optional

The norm implementation. It must accept an *FnVector* argument, return a float and satisfy the following conditions for all vectors x, y and scalars s :

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

By default, `norm(x)` is calculated as `inner(x, x)`.

This option cannot be combined with `weight`, `dist` or `inner`.

inner : callable, optional

The inner product implementation. It must accept two *FnVector* arguments, return a element from the field of the space (real or complex number) and satisfy the following conditions for all vectors x, y, z and scalars s :

- $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- $\langle sx, y \rangle = s \langle x, y \rangle$
- $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
- $\langle x, x \rangle = 0 \Leftrightarrow x = 0$

This option cannot be combined with `weight`, `dist` or `norm`.

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

This option can only be used if `exponent` is 2.0.

Default: `False`.

FnConstWeighting

class `odl.space.ntuples.FnConstWeighting` (*constant*, *exponent=2.0*, *dist_using_inner=False*)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Weighting of *Fn* by a constant.

For exponent 2.0, a new weighted inner product with constant c is defined as

$$\langle a, b \rangle_c := c b^H a$$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is defined as

$$\|a\|_{c,\infty} := c \|a\|_\infty,$$

otherwise it is

$$\|a\|_{c,p} := c^{1/p} \|a\|_p.$$

Not that this definition does **not** fulfill the limit property in p , i.e.

$$\lim_{p \rightarrow \infty} \|a\|_{c,p} = \|a\|_\infty \neq \|a\|_{c,\infty}$$

unless $c = 1$.

The constant c must be positive.

Attributes

<i>const</i>	Weighting constant of this inner product.
--------------	---

Continued on next page

Table 2.201 – continued from previous page

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

FnConstWeighting.constFnConstWeighting.**const**

Weighting constant of this inner product.

FnConstWeighting.exponentFnConstWeighting.**exponent**

Exponent of this weighting.

FnConstWeighting.implFnConstWeighting.**impl**

Implementation backend of this weighting.

Methods

<i>__eq__</i> (other)	Return <code>self == other</code> .
<i>dist</i> (x1, x2)	Calculate the constant-weighted distance between two vectors.
<i>equiv</i> (other)	Test if other is an equivalent weighting.
<i>inner</i> (x1, x2)	Calculate the constant-weighted inner product of two vectors.
<i>norm</i> (x)	Calculate the constant-weighted norm of a vector.

FnConstWeighting.__eq__FnConstWeighting.**__eq__**(other)Return `self == other`.**Return** `sequal` : boolTrue if other is an *FnConstWeighting* instance with the same constant, False otherwise.**FnConstWeighting.dist**FnConstWeighting.**dist**(x1, x2)

Calculate the constant-weighted distance between two vectors.

Parameters `x1, x2` : *FnVector*

Vectors whose mutual distance is calculated

Returns `dist` : float

The distance between the vectors

FnConstWeighting.equivFnConstWeighting.**equiv**(other)

Test if other is an equivalent weighting.

Return `sequal` : bool

True if `other` is an `FnWeightingBase` instance with the same `FnWeightingBase.impl`, which yields the same result as this inner product for any input, False otherwise. This is the same as equality if `other` is an `FnConstWeighting` instance, otherwise the `equiv` method of `other` is called.

FnConstWeighting.inner

`FnConstWeighting.inner(x1, x2)`

Calculate the constant-weighted inner product of two vectors.

Parameters`x1, x2` : `FnVector`

Vectors whose inner product is calculated

Returns`inner` : float or complex

The inner product of the two provided vectors

FnConstWeighting.norm

`FnConstWeighting.norm(x)`

Calculate the constant-weighted norm of a vector.

Parameters`x1` : `FnVector`

Vector whose norm is calculated

Returns`norm` : float

The norm of the vector

__init__(`constant`, `exponent=2.0`, `dist_using_inner=False`)

Initialize a new instance.

Parameters`constant` : positive float

Weighting constant of the inner product.

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

Can only be used if `exponent` is 2.0.

FnCustomDist

class `odl.space.ntuples.FnCustomDist(dist)`

Bases: `odl.space.base_ntuples.FnWeightingBase`

Custom distance on `Fn`, removes `norm` and `inner`.

Attributes

<i>dist</i>	Custom distance of this instance..
<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.

FnCustomDist.distFnCustomDist.**dist**

Custom distance of this instance..

FnCustomDist.exponentFnCustomDist.**exponent**

Exponent of this weighting.

FnCustomDist.implFnCustomDist.**impl**

Implementation backend of this weighting.

Methods

<i>__eq__</i> (other)	Return self == other.
<i>equiv</i> (other)	Test if other is an equivalent inner product.
<i>inner</i> (x1, x2)	Inner product is not defined for custom distance.
<i>norm</i> (x)	Norm is not defined for custom distance.

FnCustomDist.__eq__FnCustomDist.**__eq__**(other)

Return self == other.

Returnsequal : boolTrue if other is an *FnCustomDist* instance with the same norm, False otherwise.**FnCustomDist.equiv**FnCustomDist.**equiv**(other)

Test if other is an equivalent inner product.

Should be overwritten, default tests for equality.

Returnsequivalent : boolTrue if other is a *FnWeightingBase* instance which yields the same result as this inner product for any input, False otherwise.**FnCustomDist.inner**FnCustomDist.**inner**(x1, x2)

Inner product is not defined for custom distance.

FnCustomDist.normFnCustomDist.**norm**(x)

Norm is not defined for custom distance.

`__init__(dist)`

Initialize a new instance.

Parameters`dist` : callable

The distance function defining a metric on \mathbb{F}^n . It must accept two *FnVector* arguments and fulfill the following mathematical conditions for any three vectors x, y, z :

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

FnCustomInnerProduct

class `odl.space.ntuples.FnCustomInnerProduct` (*inner*, *dist_using_inner=False*)

Bases: `odl.space.base_ntuples.FnWeightingBase`

Custom inner product on *Fn*.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.
<i>inner</i>	Custom inner product of this instance..

FnCustomInnerProduct.exponent

`FnCustomInnerProduct.exponent`

Exponent of this weighting.

FnCustomInnerProduct.impl

`FnCustomInnerProduct.impl`

Implementation backend of this weighting.

FnCustomInnerProduct.inner

`FnCustomInnerProduct.inner`

Custom inner product of this instance..

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>equiv(other)</code>	Test if <code>other</code> is an equivalent inner product.
<code>norm(x)</code>	Calculate the norm of a vector.

FnCustomInnerProduct.__eq__

`FnCustomInnerProduct.__eq__(other)`

Return `self == other`.

Returnsequal : bool

True if other is an *FnCustomInnerProduct* instance with the same inner product,
False otherwise.

FnCustomInnerProduct.dist

FnCustomInnerProduct.**dist** (*x1*, *x2*)

Calculate the distance between two vectors.

This is the standard implementation using *norm*. Subclasses should override it for optimization purposes.

Parameters*x1*, *x2* : *FnBaseVector*

Vectors whose mutual distance is calculated

Returns*dist* : float

The distance between the vectors

FnCustomInnerProduct.equiv

FnCustomInnerProduct.**equiv** (*other*)

Test if other is an equivalent inner product.

Should be overwritten, default tests for equality.

Returnsequivalent : bool

True if other is a *FnWeightingBase* instance which yields the same result as this
inner product for any input, False otherwise.

FnCustomInnerProduct.norm

FnCustomInnerProduct.**norm** (*x*)

Calculate the norm of a vector.

This is the standard implementation using *inner*. Subclasses should override it for optimization purposes.

Parameters*x1* : *FnBaseVector*

Vector whose norm is calculated

Returns*norm* : float

The norm of the vector

__init__ (*inner*, *dist_using_inner=False*)

Initialize a new instance.

Parameters*inner* : callable

The inner product implementation. It must accept two *FnVector* arguments, return a
element from the field of the space (real or complex number) and satisfy the following
conditions for all vectors *x*, *y*, *z* and scalars *s*:

- $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- $\langle sx, y \rangle = s \langle x, y \rangle$
- $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
- $\langle x, x \rangle = 0 \Leftrightarrow x = 0$

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

FnCustomNorm

class `odl.space.ntuples.FnCustomNorm(norm)`
Bases: `odl.space.base_ntuples.FnWeightingBase`
Custom norm on *Fn*, removes inner.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.
<i>norm</i>	Custom norm of this instance..

FnCustomNorm.exponent
`FnCustomNorm.exponent`
Exponent of this weighting.

FnCustomNorm.impl
`FnCustomNorm.impl`
Implementation backend of this weighting.

FnCustomNorm.norm
`FnCustomNorm.norm`
Custom norm of this instance..

Methods

<i>__eq__</i> (other)	Return <code>self == other</code> .
<i>dist</i> (x1, x2)	Calculate the distance between two vectors.
<i>equiv</i> (other)	Test if <code>other</code> is an equivalent inner product.
<i>inner</i> (x1, x2)	Inner product is not defined for custom distance.

FnCustomNorm.__eq__
`FnCustomNorm.__eq__(other)`
Return `self == other`.

Returnsequal : bool

True if `other` is an *FnCustomNorm* instance with the same norm, False otherwise.

FnCustomNorm.dist

`FnCustomNorm.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using `norm`. Subclasses should override it for optimization purposes.

Parameters`x1, x2` : `FnBaseVector`

Vectors whose mutual distance is calculated

Returns`dist` : `float`

The distance between the vectors

FnCustomNorm.equiv

`FnCustomNorm.equiv(other)`

Test if `other` is an equivalent inner product.

Should be overwritten, default tests for equality.

Return`seivalent` : `bool`

True if `other` is a `FnWeightingBase` instance which yields the same result as this inner product for any input, False otherwise.

FnCustomNorm.inner

`FnCustomNorm.inner(x1, x2)`

Inner product is not defined for custom distance.

__init__(`norm`)

Initialize a new instance.

Parameters`norm` : callable

The norm implementation. It must accept an `FnVector` argument, return a `float` and satisfy the following conditions for all vectors x, y and scalars s :

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

FnMatrixWeighting

`class odl.space.ntuples.FnMatrixWeighting(matrix, exponent=2.0, dist_using_inner=False, **kwargs)`

Bases: `odl.space.base_ntuples.FnWeightingBase`

Matrix weighting for `Fn`.

For exponent 2.0, a new weighted inner product with matrix W is defined as

$$\langle a, b \rangle_W := b^H W a$$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$$\|a\|_{W, \infty} := \|W a\|_{\infty},$$

otherwise it is

$$\|a\|_{W,p} := \|W^{1/p}a\|_p.$$

Not that this definition does **not** fulfill the limit property in p , i.e.

$$\lim_{p \rightarrow \infty} \|a\|_{W,p} = \|a\|_\infty \neq \|a\|_{W,\infty}$$

unless W is the identity matrix.

The matrix must be Hermitian and positive definite, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

<code>exponent</code>	Exponent of this weighting.
<code>impl</code>	Implementation backend of this weighting.
<code>matrix</code>	Weighting matrix of this inner product.
<code>matrix_issparse</code>	Whether the representing matrix is sparse or not.

FnMatrixWeighting.exponent

`FnMatrixWeighting.exponent`
Exponent of this weighting.

FnMatrixWeighting.impl

`FnMatrixWeighting.impl`
Implementation backend of this weighting.

FnMatrixWeighting.matrix

`FnMatrixWeighting.matrix`
Weighting matrix of this inner product.

FnMatrixWeighting.matrix_issparse

`FnMatrixWeighting.matrix_issparse`
Whether the representing matrix is sparse or not.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Calculate the distance between two vectors.
<code>equiv(other)</code>	Test if <code>other</code> is an equivalent weighting.
<code>inner(x1, x2)</code>	Calculate the matrix-weighted inner product of two vectors.
<code>matrix_isvalid()</code>	Test if the matrix is positive definite Hermitian.
<code>norm(x)</code>	Calculate the matrix-weighted norm of a vector.

FnMatrixWeighting.__eq__

`FnMatrixWeighting.__eq__(other)`
Return `self == other`.

Returnsequals : bool

True if `other` is an `FnMatrixWeighting` instance with **identical** matrix, False

otherwise.

See also:

equiv test for equivalent inner products

FnMatrixWeighting.dist

`FnMatrixWeighting.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using *norm*. Subclasses should override it for optimization purposes.

Parameters`x1, x2` : *FnBaseVector*

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

FnMatrixWeighting.equiv

`FnMatrixWeighting.equiv(other)`

Test if `other` is an equivalent weighting.

Return`seivalent` : bool

True if `other` is an *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, False otherwise. This is checked by entry-wise comparison of this inner product's matrix with the matrix or constant of `other`.

FnMatrixWeighting.inner

`FnMatrixWeighting.inner(x1, x2)`

Calculate the matrix-weighted inner product of two vectors.

Parameters`x1, x2` : *FnVector*

Vectors whose inner product is calculated

Returns`sinner` : float or complex

The inner product of the vectors

FnMatrixWeighting.matrix_isvalid

`FnMatrixWeighting.matrix_isvalid()`

Test if the matrix is positive definite Hermitian.

This test tries to calculate a Cholesky decomposition and can be very time-consuming for large matrices. Sparse matrices are not supported.

FnMatrixWeighting.norm

`FnMatrixWeighting.norm(x)`

Calculate the matrix-weighted norm of a vector.

Parameters`x` : *FnVector*

Vector whose norm is calculated

Returns`norm` : float

The norm of the vector

`__init__(matrix, exponent=2.0, dist_using_inner=False, **kwargs)`

Initialize a new instance.

Parameters`matrix`: `scipy.sparse.spmatrix` or *array-like*, 2-dim.

Square weighting matrix of the inner product

exponent: positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined. If `matrix` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

dist_using_inner: bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

Can only be used if `exponent` is 2.0.

precomp_mat_pow: bool

If `True`, precompute the matrix power $W^{1/p}$ during initialization. This has no effect if `exponent` is 1.0, 2.0 or `inf`.

Default: `False`

cache_mat_pow: bool

If `True`, cache the matrix power $W^{1/p}$ during the first call to `norm` or `dist`. This has no effect if `exponent` is 1.0, 2.0 or `inf`.

Default: `False`

FnNoWeighting

class `odl.space.ntuples.FnNoWeighting(exponent=2.0, dist_using_inner=False)`

Bases: `odl.space.ntuples.FnConstWeighting`

Weighting of Fn with constant 1.

For exponent 2.0, the unweighted inner product is defined as

$$\langle a, b \rangle := b^H a$$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined.

Attributes

<code>const</code>	Weighting constant of this inner product.
<code>exponent</code>	Exponent of this weighting.
<code>impl</code>	Implementation backend of this weighting.

FnNoWeighting.const

`FnNoWeighting.const`

Weighting constant of this inner product.

FnNoWeighting.exponent

`FnNoWeighting.exponent`

Exponent of this weighting.

FnNoWeighting.impl

`FnNoWeighting.impl`

Implementation backend of this weighting.

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>dist(x1, x2)</code>	Calculate the constant-weighted distance between two vectors.
<code>equiv(other)</code>	Test if other is an equivalent weighting.
<code>inner(x1, x2)</code>	Calculate the constant-weighted inner product of two vectors.
<code>norm(x)</code>	Calculate the constant-weighted norm of a vector.

FnNoWeighting.__eq__

`FnNoWeighting.__eq__(other)`

Return `self == other`.

Return`sequal` : bool

True if other is an `FnConstWeighting` instance with the same constant, False otherwise.

FnNoWeighting.dist

`FnNoWeighting.dist(x1, x2)`

Calculate the constant-weighted distance between two vectors.

Parameters`x1, x2` : `FnVector`

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

FnNoWeighting.equiv

`FnNoWeighting.equiv(other)`

Test if other is an equivalent weighting.

Return`sequivalent` : bool

True if other is an `FnWeightingBase` instance with the same `FnWeightingBase.impl`, which yields the same result as this inner product for any input, False otherwise. This is the same as equality if other is an `FnConstWeighting` instance, otherwise the `equiv` method of other is called.

FnNoWeighting.inner

`FnNoWeighting.inner(x1, x2)`

Calculate the constant-weighted inner product of two vectors.

Parameters`x1, x2` : *FnVector*

Vectors whose inner product is calculated

Returns`inner` : float or complex

The inner product of the two provided vectors

FnNoWeighting.norm

`FnNoWeighting.norm(x)`

Calculate the constant-weighted norm of a vector.

Parameters`x1` : *FnVector*

Vector whose norm is calculated

Returns`norm` : float

The norm of the vector

`__init__` (*exponent=2.0, dist_using_inner=False*)

Initialize a new instance.

Parameter`exponent` : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

Can only be used if `exponent` is 2.0.

FnVector

class `odl.space.ntuples.FnVector(space, data)`

Bases: `odl.space.base_ntuples.FnBaseVector`, `odl.space.ntuples.NtuplesVector`

Representation of an *Fn* element.

Attributes

<i>T</i>	The transpose of a vector, the functional given by (\cdot) .
<i>data</i>	The raw <code>numpy.ndarray</code> representing the data.
<i>data_ptr</i>	A raw pointer to the data container.
<i>dtype</i>	Length of this vector, equal to space size.
<i>imag</i>	The imaginary part of this vector.
<i>itemsiz</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
Continued on next page	

Table 2.213 – continued from previous page

<i>real</i>	The real part of this vector.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	Length of this vector, equal to space size.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>NtuplesUFuncs</i> , access to numpy style ufuncs.

FnVector.TFnVector.**T**The transpose of a vector, the functional given by (*.*, self)**Return**transpose : *InnerProductOperator***Notes**

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

FnVector.dataFnVector.**data**

The raw numpy.ndarray representing the data.

FnVector.data_ptrFnVector.**data_ptr**

A raw pointer to the data container.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'int32').element([1, 2, 3])
>>> arr_type = ctypes.c_int32 * 3
>>> buffer = arr_type.from_address(vec.data_ptr)
>>> arr = np.frombuffer(buffer, dtype='int32')
>>> print(arr)
[1 2 3]
```

In-place modification via pointer:

```
>>> arr[0] = 5
>>> print(vec)
[5, 2, 3]
```

FnVector.dtype**FnVector.dtype**

Length of this vector, equal to space size.

FnVector.imag**FnVector.imag**

The imaginary part of this vector.

Returns**imag** : *FnVector*The imaginary part this vector as a vector in *Rn***Examples**

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> x.imag
Rn(3).element([1.0, 0.0, -2.0])
```

The *Rn* vector is really a view, so changes affect the original array:

```
>>> x.imag *= 2
>>> x
Cn(3).element([(5+2j), (3+0j), (2-4j)])
```

FnVector.itemsize**FnVector.itemsize**

The size in bytes on one element of this type.

FnVector.nbytes**FnVector.nbytes**

The number of bytes this vector uses in memory.

FnVector.ndim**FnVector.ndim**

Number of dimensions, always 1.

FnVector.real**FnVector.real**

The real part of this vector.

Returns**real** : *FnVector* view with dtypeThe real part this vector as a vector in *Rn***Examples**

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> x.real
Rn(3).element([5.0, 3.0, 2.0])
```

The *Rn* vector is really a view, so changes affect the original array:

```
>>> x.real *= 2
>>> x
Cn(3).element([(10+1j), (6+0j), (4-2j)])
```

FnVector.shape

`FnVector.shape`

Number of entries per axis, equals (size,) for linear storage.

FnVector.size

`FnVector.size`

Length of this vector, equal to space size.

FnVector.space

`FnVector.space`

Space to which this vector.

FnVector.ufunc

`FnVector.ufunc`

NtuplesUFuncs, access to numpy style ufuncs.

Notes

These are optimized for use with ntuples and incur no overhead.

Examples

```
>>> r2 = Rn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
Rn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
Rn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
Rn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

They also support an out parameter

```

>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
Rn(2).element([4.0, 2.0])
>>> result is out
True

```

Methods

<code>__eq__(other)</code>	
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>assign(other)</code>	Assign the values of <code>other</code> to self.
<code>conj([out])</code>	The complex conjugate of this vector.
<code>copy()</code>	
<code>dist(other)</code>	Distance to <code>other</code> .
<code>divide(x, y)</code>	Divide by <code>other</code> inplace.
<code>inner(other)</code>	Inner product with <code>other</code> .
<code>lincomb(a, x1[, b, x2])</code>	Assign a linear combination to this vector.
<code>multiply(x, y)</code>	Multiply by <code>other</code> inplace.
<code>norm()</code>	Norm of vector
<code>set_zero()</code>	Set this vector to zero.
<code>show([title, method, show, fig])</code>	Display the function graphically.

FnVector.__eq__

FnVector.__eq__(*other*)

FnVector.__getitem__

FnVector.__getitem__(*indices*)

Access values of this vector.

Parameters`indices`: int or slice

The position(s) that should be accessed

Returns`values`: scalar or *NtuplesVector*

The value(s) at the index (`indices`)

Examples

```

>>> str_3 = Ntuples(3, dtype='U6') # 6-char unicode
>>> x = str_3.element(['a', 'Hello!', '0'])
>>> print(x[0])
a
>>> print(x[1:3])
[Hello!, 0]
>>> x[1:3].space
Ntuples(2, '<U6')

```

FnVector.__setitem__

FnVector.__setitem__(indices, values)

Set values of this vector.

Parametersindices : int or slice

The position(s) that should be set

values : scalar, *array-like* or *NtuplesVector*

The value(s) that are to be assigned.

If indices is an integer, value must be scalar.

If indices is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

ReturnsNone**Examples**

```
>>> int_3 = Ntuples(3, 'int')
>>> x = int_3.element([1, 2, 3])
>>> x[0] = 5
>>> x
Ntuples(3, 'int').element([5, 2, 3])
```

Assignment from array-like structures or another vector:

```
>>> y = Ntuples(2, 'short').element([-1, 2])
>>> x[:2] = y
>>> x
Ntuples(3, 'int').element([-1, 2, 3])
>>> x[1:3] = [7, 8]
>>> x
Ntuples(3, 'int').element([-1, 7, 8])
>>> x[:] = np.array([0, 0, 0])
>>> x
Ntuples(3, 'int').element([0, 0, 0])
```

Broadcasting is also supported:

```
>>> x[1:3] = -2.
>>> x
Ntuples(3, 'int').element([0, -2, -2])
```

Array views are preserved:

```
>>> y = x[:,2] # view into x
>>> y[:] = -9
>>> print(y)
[-9, -9]
>>> print(x)
[-9, -2, -9]
```

Be aware of unsafe casts and over-/underflows, there will be warnings at maximum.

```
>>> x = Ntuples(2, 'int8').element([0, 0])
>>> maxval = 255 # maximum signed 8-bit unsigned int
>>> x[0] = maxval + 1
```

```
>>> x
Ntuples(2, 'int8').element([0, 0])
```

FnVector.asarray

`FnVector.asarray` (*start=None, stop=None, step=None, out=None*)

Extract the data of this array as a numpy array.

Parameters*start* : int, optional

Start position. None means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returns*asarray* : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'float').element([1, 2, 3])
>>> vec.asarray()
array([ 1.,  2.,  3.])
>>> vec.asarray(start=1, stop=3)
array([ 2.,  3.])
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='float')
>>> result = vec.asarray(out=out)
>>> out
array([ 1.,  2.,  3.])
>>> result is out
True
```

FnVector.assign

`FnVector.assign` (*other*)

Assign the values of *other* to self.

FnVector.conj

`FnVector.conj` (*out=None*)

The complex conjugate of this vector.

Parameters*out* : *FnVector*, optional

Vector to which the complex conjugate is written. Must be an element of this vector's space.

Returns `out : FnVector`

The complex conjugate vector. If `out` was provided, the returned object is a reference to it.

Examples

```
>>> x = Cn(3).element([5+1j, 3, 2-2j])
>>> y = x.conj(); print(y)
[(5-1j), (3-0j), (2+2j)]
```

The `out` parameter allows you to avoid a copy

```
>>> z = Cn(3).element()
>>> z_out = x.conj(out=z); print(z)
[(5-1j), (3-0j), (2+2j)]
>>> z_out is z
True
```

It can also be used for in-place conj

```
>>> x_out = x.conj(out=x); print(x)
[(5-1j), (3-0j), (2+2j)]
>>> x_out is x
True
```

FnVector.copy

`FnVector.copy()`

FnVector.dist

`FnVector.dist(other)`

Distance to other.

LinearSpace.dist

FnVector.divide

`FnVector.divide(x, y)`

Divide by other inplace.

LinearSpace.divide

FnVector.inner

`FnVector.inner(other)`

Inner product with other.

LinearSpace.inner

FnVector.lincomb

`FnVector.lincomb(a, x1, b=None, x2=None)`

Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

FnVector.multiply`FnVector.multiply(x, y)`

Multiply by other inplace.

*LinearSpace.multiply***FnVector.norm**`FnVector.norm()`

Norm of vector

*LinearSpace.norm***FnVector.set_zero**`FnVector.set_zero()`

Set this vector to zero.

*LinearSpace.zero***FnVector.show**`FnVector.show(title=None, method='scatter', show=False, fig=None, **kwargs)`

Display the function graphically.

Parameter`title` : str, optional

Set the title of the figure

method : str, optional

1d methods:

'plot' : graph plot

'scatter' : point plot

show : bool, optional

If the plot should be showed now or deferred until later.

fig : matplotlib.figure.FigureThe figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.**kwargs** : {'figsize', 'saveto', ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig` : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

See also:*odl.util.graphics.show_discrete_data* Underlying implementation`__init__(space, data)`

Initialize a new instance.

FnVectorWeighting

class odl.space.ntuples.**FnVectorWeighting**(vector, exponent=2.0, dist_using_inner=False)

Bases: odl.space.base_ntuples.FnWeightingBase

Vector weighting for *Fn*.

For exponent 2.0, a new weighted inner product with vector w is defined as

$$\langle a, b \rangle_w := b^H (w \odot a)$$

with b^H standing for transposed complex conjugate and $w \odot a$ for element-wise multiplication.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$$\|a\|_{w,\infty} := \|w \odot a\|_\infty,$$

otherwise it is

$$\|a\|_{w,p} := \|w^{1/p} \odot a\|_p.$$

Not that this definition does **not** fulfill the limit property in p , i.e.

$$\lim_{p \rightarrow \infty} \|a\|_{w,p} = \|a\|_\infty \neq \|a\|_{w,\infty}$$

unless $w = (1, \dots, 1)$.

The vector may only have positive entries, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

<i>exponent</i>	Exponent of this weighting.
<i>impl</i>	Implementation backend of this weighting.
<i>vector</i>	Weighting vector of this inner product.

FnVectorWeighting.exponent

FnVectorWeighting.**exponent**

Exponent of this weighting.

FnVectorWeighting.impl

FnVectorWeighting.**impl**

Implementation backend of this weighting.

FnVectorWeighting.vector

FnVectorWeighting.**vector**

Weighting vector of this inner product.

Methods

<i>__eq__</i> (other)	Return self == other.
<i>dist</i> (x1, x2)	Calculate the distance between two vectors.
<i>equiv</i> (other)	Test if other is an equivalent weighting.
Continued on next page	

Table 2.216 – continued from previous page

<code>inner(x1, x2)</code>	Calculate the vector weighted inner product of two vectors.
<code>norm(x)</code>	Calculate the vector-weighted norm of a vector.
<code>vector_is_valid()</code>	Test if the vector is a valid weight, i.e.

FnVectorWeighting.__eq__`FnVectorWeighting.__eq__(other)`Return `self == other`.**Returnsequals** : boolTrue if `other` is an `FnVectorWeighting` instance with **identical** vector, False otherwise.**See also:**`equiv` test for equivalent inner products**FnVectorWeighting.dist**`FnVectorWeighting.dist(x1, x2)`

Calculate the distance between two vectors.

This is the standard implementation using `norm`. Subclasses should override it for optimization purposes.**Parameters**`x1, x2` : `FnBaseVector`

Vectors whose mutual distance is calculated

Returns`dist` : float

The distance between the vectors

FnVectorWeighting.equiv`FnVectorWeighting.equiv(other)`Test if `other` is an equivalent weighting.**Returnsequivalent** : boolTrue if `other` is an `FnWeightingBase` instance with the same `FnWeightingBase.impl`, which yields the same result as this inner product for any input, False otherwise. This is checked by entry-wise comparison of matrices/vectors/constant of this inner product and `other`.**FnVectorWeighting.inner**`FnVectorWeighting.inner(x1, x2)`

Calculate the vector weighted inner product of two vectors.

Parameters`x1, x2` : `FnVector`

Vectors whose inner product is calculated

Returns`inner` : float or complex

The inner product of the two provided vectors

FnVectorWeighting.norm

`FnVectorWeighting.norm(x)`

Calculate the vector-weighted norm of a vector.

Parameters`x` : *FnVector*

Vector whose norm is calculated

Returns`norm` : float

The norm of the provided vector

FnVectorWeighting.vector_is_valid

`FnVectorWeighting.vector_is_valid()`

Test if the vector is a valid weight, i.e. positive.

`__init__(vector, exponent=2.0, dist_using_inner=False)`

Initialize a new instance.

Parameters`vector` : *array-like*, one-dim.

Weighting vector of the inner product

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined. If `matrix` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

dist_using_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y .

Can only be used if `exponent` is 2.0.

MatVecOperator

`class odl.space.ntuples.MatVecOperator(matrix, dom=None, ran=None)`

Bases: *odl.operator.operator.Operator*

Matrix multiply operator $\mathbb{F}^n \rightarrow \mathbb{F}^m$.

Attributes

<i>adjoint</i>	Adjoint operator represented by the adjoint matrix.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	Return the operator inverse.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>matrix</i>	Matrix representing this operator.
<i>matrix_issparse</i>	Whether the representing matrix is sparse or not.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

MatVecOperator.adjoint

`MatVecOperator.adjoint`

Adjoint operator represented by the adjoint matrix.

`MatVecOperator.domain`

`MatVecOperator.domain`

Set of objects on which this operator can be evaluated.

`MatVecOperator.inverse`

`MatVecOperator.inverse`

Return the operator inverse.

`RaisesOpNotImplementedError`

Since the inverse cannot be default implemented.

`MatVecOperator.is_functional`

`MatVecOperator.is_functional`

True if the this operator's range is a *Field*.

`MatVecOperator.is_linear`

`MatVecOperator.is_linear`

True if this operator is linear.

`MatVecOperator.matrix`

`MatVecOperator.matrix`

Matrix representing this operator.

`MatVecOperator.matrix_issparse`

`MatVecOperator.matrix_issparse`

Whether the representing matrix is sparse or not.

`MatVecOperator.range`

`MatVecOperator.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Raw apply method on input, writing to given output.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

`MatVecOperator.__call__`

`MatVecOperator.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

MatVecOperator._call

`MatVecOperator._call` (*x*, *out=None*)

Raw apply method on input, writing to given output.

MatVecOperator.derivative

`MatVecOperator.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*matrix*, *dom=None*, *ran=None*)

Initialize a new instance.

Parameters*matrix* : *array-like* or `scipy.sparse.spmatrix`

Matrix representing the linear operator. Its shape must be (m, n) , where n is the size of `dom` and m the size of `ran`. Its dtype must be castable to the range dtype.

dom : *Fn*, optional

Space on whose elements the matrix acts. If not provided, the domain is inferred from the matrix dtype and shape. If provided, its dtype must be castable to the range dtype.

ran : *Fn*, optional

Space to which the matrix maps. If not provided, the domain is inferred from the matrix dtype and shape.

Ntuples

class `odl.space.ntuples.Ntuples` (*size, dtype*)
Bases: `odl.space.base_ntuples.NtuplesBase`

The set of n-tuples of arbitrary type.

Attributes

<i>dtype</i>	The data type of each entry.
<i>element_type</i>	<i>NtuplesVector</i>
<i>shape</i>	The shape of this space.
<i>size</i>	The number of entries per tuple.

Ntuples.dtype

`Ntuples.dtype`
The data type of each entry.

Ntuples.element_type

`Ntuples.element_type`
NtuplesVector

Ntuples.shape

`Ntuples.shape`
The shape of this space.

Ntuples.size

`Ntuples.size`
The number of entries per tuple.

Methods

<code>__contains__(other)</code>	Return <code>other</code> in <code>self</code> .
<code>__eq__(other)</code>	Return <code>self == other</code> .
Continued on next page	

Table 2.220 – continued from previous page

<code>contains_all(other)</code>	Test if all points in <code>other</code> are contained in this set.
<code>contains_set(other)</code>	Test if <code>other</code> is a subset of this set.
<code>element([inp, data_ptr])</code>	Create a new element.
<code>one()</code>	Create a vector of ones.
<code>zero()</code>	Create a vector of zeros.

Ntuples.__contains__Ntuples.__contains__(*other*)Return `other` in `self`.**Returns**`contains` : bool

True if `other` is an `NtuplesBaseVector` instance and `other.space` is equal to this space, False otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

Ntuples.__eq__Ntuples.__eq__(*other*)Return `self == other`.**Returns**`equals` : bool

True if `other` is an instance of this space's type with the same *size* and *dtype*, otherwise False.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
```

```
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

Ntuples.contains_all

`Ntuples.contains_all` (*other*)

Test if all points in *other* are contained in this set.

This is a default implementation and should be overridden by subclasses.

Ntuples.contains_set

`Ntuples.contains_set` (*other*)

Test if *other* is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Ntuples.element

`Ntuples.element` (*inp=None, data_ptr=None*)

Create a new element.

Parameters*inp* : *array-like*, optional

Input to initialize the new element.

If *inp* is `None`, an empty element is created with no guarantee of its state (memory allocation only).

If *inp* is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied.

Return*element* : *NtuplesVector*

The new element created (from *inp*).

Notes

This method preserves “array views” of correct size and type, see the examples below.

Examples

```
>>> strings3 = Ntuples(3, dtype='U1') # 1-char strings
>>> x = strings3.element(['w', 'b', 'w'])
>>> print(x)
[w, b, w]
>>> x.space
Ntuples(3, '<U1')
```

Construction from data pointer:

```
>>> int3 = Ntuples(3, dtype='int')
>>> x = int3.element([1, 2, 3])
>>> y = int3.element(data_ptr=x.data_ptr)
>>> print(y)
[1, 2, 3]
>>> y[0] = 5
```

```
>>> print(x)
[5, 2, 3]
```

Ntuples.one

`Ntuples.one()`
Create a vector of ones.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.one()
>>> x
Cn(3).element([(1+0j), (1+0j), (1+0j)])
```

Ntuples.zero

`Ntuples.zero()`
Create a vector of zeros.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.zero()
>>> x
Cn(3).element([0j, 0j, 0j])
```

`__init__(size, dtype)`
Initialize a new instance.

Parameters
size : non-negative int
The number of entries per tuple

dtype :
The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

NtuplesVector

class `odl.space.ntuples.NtuplesVector(space, data)`
Bases: `odl.space.base_ntuples.NtuplesBaseVector`
Representation of an *Ntuples* element.

Attributes

<i>data</i>	The raw <code>numpy.ndarray</code> representing the data.
<i>data_ptr</i>	A raw pointer to the data container.
<i>dtype</i>	Length of this vector, equal to <i>space</i> size.
Continued on next page	

Table 2.221 – continued from previous page

<i>itemsize</i>	The size in bytes on one element of this type.
<i>nbytes</i>	The number of bytes this vector uses in memory.
<i>ndim</i>	Number of dimensions, always 1.
<i>shape</i>	Number of entries per axis, equals (size,) for linear storage.
<i>size</i>	Length of this vector, equal to space size.
<i>space</i>	Space to which this vector.
<i>ufunc</i>	<i>NtuplesUFuncs</i> , access to numpy style ufuncs.

NtuplesVector.data**NtuplesVector.data**

The raw numpy.ndarray representing the data.

NtuplesVector.data_ptr**NtuplesVector.data_ptr**

A raw pointer to the data container.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'int32').element([1, 2, 3])
>>> arr_type = ctypes.c_int32 * 3
>>> buffer = arr_type.from_address(vec.data_ptr)
>>> arr = np.frombuffer(buffer, dtype='int32')
>>> print(arr)
[1 2 3]
```

In-place modification via pointer:

```
>>> arr[0] = 5
>>> print(vec)
[5, 2, 3]
```

NtuplesVector.dtype**NtuplesVector.dtype**

Length of this vector, equal to space size.

NtuplesVector.itemsize**NtuplesVector.itemsize**

The size in bytes on one element of this type.

NtuplesVector.nbytes**NtuplesVector.nbytes**

The number of bytes this vector uses in memory.

NtuplesVector.ndim**NtuplesVector.ndim**

Number of dimensions, always 1.

NtuplesVector.shape**NtuplesVector.shape**

Number of entries per axis, equals (size,) for linear storage.

NtuplesVector.size**NtuplesVector.size**

Length of this vector, equal to space size.

NtuplesVector.space**NtuplesVector.space**

Space to which this vector.

NtuplesVector.ufunc**NtuplesVector.ufunc***NtuplesUFuncs*, access to numpy style ufuncs.**Notes**

These are optimized for use with ntuples and incur no overhead.

Examples

```
>>> r2 = Rn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
Rn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
Rn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
Rn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

They also support an out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
Rn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

<code>__eq__(other)</code>	Return <code>self == other</code> .
<code>__getitem__(indices)</code>	Access values of this vector.
<code>__setitem__(indices, values)</code>	Set values of this vector.
<code>asarray([start, stop, step, out])</code>	Extract the data of this array as a numpy array.
<code>copy()</code>	Create an identical (deep) copy of this vector.
<code>show([title, method, show, fig])</code>	Display the function graphically.

NtuplesVector.__eq__

`NtuplesVector.__eq__(other)`

Return `self == other`.

Returnsequals : bool

True if all entries of `other` are equal to this vector's entries, False otherwise.

Notes

Space membership is not checked, hence vectors from different spaces can be equal.

Examples

```
>>> vec1 = Ntuples(3, int).element([1, 2, 3])
>>> vec2 = Ntuples(3, int).element([-1, 2, 0])
>>> vec1 == vec2
False
>>> vec2 = Ntuples(3, int).element([1, 2, 3])
>>> vec1 == vec2
True
```

Space membership matters:

```
>>> vec2 = Ntuples(3, float).element([1, 2, 3])
>>> vec1 == vec2 or vec2 == vec1
False
```

NtuplesVector.__getitem__

`NtuplesVector.__getitem__(indices)`

Access values of this vector.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : scalar or *NtuplesVector*

The value(s) at the index (indices)

Examples

```

>>> str_3 = Ntuples(3, dtype='U6') # 6-char unicode
>>> x = str_3.element(['a', 'Hello!', '0'])
>>> print(x[0])
a
>>> print(x[1:3])
[Hello!, 0]
>>> x[1:3].space
Ntuples(2, '<U6')

```

NtuplesVector.__setitem__

NtuplesVector.__setitem__(*indices, values*)

Set values of this vector.

Parameters*indices*: int or slice

The position(s) that should be set

values: scalar, *array-like* or *NtuplesVector*

The value(s) that are to be assigned.

If *indices* is an integer, value must be scalar.

If *indices* is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

ReturnsNone

Examples

```

>>> int_3 = Ntuples(3, 'int')
>>> x = int_3.element([1, 2, 3])
>>> x[0] = 5
>>> x
Ntuples(3, 'int').element([5, 2, 3])

```

Assignment from array-like structures or another vector:

```

>>> y = Ntuples(2, 'short').element([-1, 2])
>>> x[:2] = y
>>> x
Ntuples(3, 'int').element([-1, 2, 3])
>>> x[1:3] = [7, 8]
>>> x
Ntuples(3, 'int').element([-1, 7, 8])
>>> x[:] = np.array([0, 0, 0])
>>> x
Ntuples(3, 'int').element([0, 0, 0])

```

Broadcasting is also supported:

```

>>> x[1:3] = -2.
>>> x
Ntuples(3, 'int').element([0, -2, -2])

```

Array views are preserved:

```
>>> y = x[:,2] # view into x
>>> y[:] = -9
>>> print(y)
[-9, -9]
>>> print(x)
[-9, -2, -9]
```

Be aware of unsafe casts and over-/underflows, there will be warnings at maximum.

```
>>> x = Ntuples(2, 'int8').element([0, 0])
>>> maxval = 255 # maximum signed 8-bit unsigned int
>>> x[0] = maxval + 1
>>> x
Ntuples(2, 'int8').element([0, 0])
```

NtuplesVector.asarray

`NtuplesVector.asarray(start=None, stop=None, step=None, out=None)`

Extract the data of this array as a numpy array.

Parameters`start` : int, optional

Start position. None means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returns`asarray` : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'float').element([1, 2, 3])
>>> vec.asarray()
array([ 1.,  2.,  3.])
>>> vec.asarray(start=1, stop=3)
array([ 2.,  3.])
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='float')
>>> result = vec.asarray(out=out)
>>> out
array([ 1.,  2.,  3.])
>>> result is out
True
```


NtuplesVector.copy`NtuplesVector.copy()`

Create an identical (deep) copy of this vector.

Parameters`None`**Returns**`copy : NtuplesVector`

The deep copy

Examples

```

>>> vec1 = Ntuples(3, 'int').element([1, 2, 3])
>>> vec2 = vec1.copy()
>>> vec2
Ntuples(3, 'int').element([1, 2, 3])
>>> vec1 == vec2
True
>>> vec1 is vec2
False

```

NtuplesVector.show`NtuplesVector.show(title=None, method='scatter', show=False, fig=None, **kwargs)`

Display the function graphically.

Parameter`title : str, optional`

Set the title of the figure

method : `str`, optional

1d methods:

‘plot’ : graph plot

‘scatter’ : point plot

show : `bool`, optional

If the plot should be showed now or deferred until later.

fig : `matplotlib.figure.Figure`The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common use case is that `fig` is the return value from an earlier call to this function.**kwargs** : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns`fig : matplotlib.figure.Figure`

The resulting figure. It is also shown to the user.

See also:`odl.util.graphics.show_discrete_data` Underlying implementation`__init__(space, data)`

Initialize a new instance.

Functions

<code>Cn(size[, dtype])</code>	The complex vector space C^n with vector multiplication.
<code>Rn(size[, dtype])</code>	The real vector space R^n with vector multiplication.
<code>weighted_dist(weight[, exponent, use_inner])</code>	Weighted distance on <i>Fn</i> spaces as free function.
<code>weighted_inner(weight)</code>	Weighted inner product on <i>Fn</i> spaces as free function.
<code>weighted_norm(weight[, exponent])</code>	Weighted norm on <i>Fn</i> spaces as free function.

Cn

`odl.space.ntuples.Cn(size, dtype='complex128', **kwargs)`

The complex vector space C^n with vector multiplication.

Parameters`size` : positive int

The number of dimensions of the space

dtype : object

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only complex floating-point data types are allowed.

kwargs : { 'weight', 'dist', 'norm', 'inner', 'dist_using_inner' }

See [*Fn*](#)

See also:

[*Fn*](#)-tuples over a field \mathbb{F} with arbitrary scalar data type

Rn

`odl.space.ntuples.Rn(size, dtype='float64', **kwargs)`

The real vector space R^n with vector multiplication.

Parameters`size` : positive int

The number of dimensions of the space

dtype : object

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only real floating-point data types are allowed.

kwargs : { 'weight', 'dist', 'norm', 'inner', 'dist_using_inner' }

See [*Fn*](#)

See also:

[*Fn*](#)-tuples over a field \mathbb{F} with arbitrary scalar data type

weighted_dist

`odl.space.ntuples.weighted_dist(weight, exponent=2.0, use_inner=False)`

Weighted distance on *Fn* spaces as free function.

Parameters**weight** : scalar or *array-like*

Weight of the distance. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

exponent : positive float

Exponent of the norm. If *weight* is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

use_inner : bool, optional

Calculate `dist` using the formula

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y \rangle.$$

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) *x* and *y*.

Can only be used if *exponent* is 2.0.

Returns**dist** : callable

Distance function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

FnConstWeighting, FnVectorWeighting, FnMatrixWeighting

weighted_inner

`odl.space.ntuples.weighted_inner(weight)`

Weighted inner product on *Fn* spaces as free function.

Parameters**weight** : scalar or *array-like*

Weight of the inner product. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

Returns**inner** : callable

Inner product function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

FnConstWeighting, FnVectorWeighting, FnMatrixWeighting

weighted_norm

`odl.space.ntuples.weighted_norm(weight, exponent=2.0)`

Weighted norm on *Fn* spaces as free function.

Parameters**weight** : scalar or *array-like*

Weight of the norm. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

exponent : positive float

Exponent of the norm. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

Returns`norm` : callable

Norm function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

`FnConstWeighting`, `FnVectorWeighting`, `FnMatrixWeighting`

space_utils

Utility functions for space implementations.

Functions

`vector`(array[, dtype, impl]) Create an n-tuples type vector from an array.

vector

`odl.space.space_utils.vector` (array, dtype=None, impl='numpy')

Create an n-tuples type vector from an array.

Parameters`array` : *array-like*

Array from which to create the vector. Scalars become one-dimensional vectors.

dtype : object, optional

Set the data type of the vector manually with this option. By default, the space type is inferred from the input data.

impl : {'numpy', 'cuda'}

Implementation backend for the vector

Returns`vec` : *NtuplesBaseVector*

Vector created from the input array. Its concrete type depends on the provided arguments.

Notes

This is a convenience function and not intended for use in speed-critical algorithms. It creates a NumPy array first, hence especially CUDA vectors as input result in a large speed penalty.

Examples

```
>>> vector([1, 2, 3]) # No automatic cast to float
Fn(3, 'int').element([1, 2, 3])
>>> vector([1, 2, 3], dtype=float)
Rn(3).element([1.0, 2.0, 3.0])
```

```
>>> vector([1 + 1j, 2, 3 - 2j])
Cn(3).element([(1+1j), (2+0j), (3-2j)])
```

Non-scalar types are also supported:

```
>>> vector([True, False])
Ntuples(2, 'bool').element([True, False])
```

Scalars become a one-element vector:

```
>>> vector(0.0)
Rn(1).element([0.0])
```

2.8.7 tomo

Tomography related operators and geometries.

Modules

backends

Back-ends for other libraries.

Modules

astra_cpu

Backend for ASTRA using CPU.

Functions

<code>astra_cpu_back_projector(proj_data, ...[, out])</code>	Run an ASTRA backward projection on the given data using the CPU.
<code>astra_cpu_forward_projector(vol_data, ...[, out])</code>	Run an ASTRA forward projection on the given data using the CPU.

astra_cpu_back_projector

`odl.tomo.backends.astra_cpu.astra_cpu_back_projector(proj_data, geometry, reco_space, out=None)`

Run an ASTRA backward projection on the given data using the CPU.

Parameters`proj_data` : *DiscreteLpVector*

Projection data to which the backward projector is applied

`geometry` : *Geometry*

Geometry defining the tomographic setup

`reco_space` : *DiscreteLp*

Space to which the calling operator maps

`out` : *DiscreteLpVector* or None, optional

Vector in the reconstruction space to which the result is written. If `None` creates an element in the reconstruction space `reco_space`

Returns`out` : `reco_space` element

Reconstruction data resulting from the application of the backward projector

astra_cpu_forward_projector

`odl.tomo.backends.astra_cpu.astra_cpu_forward_projector` (*vol_data*, *geometry*,
proj_space, *out=None*)

Run an ASTRA forward projection on the given data using the CPU.

Parameters`vol_data` : *DiscreteLpVector*

Volume data to which the forward projector is applied

`geometry` : *Geometry*

Geometry defining the tomographic setup

`proj_space` : *DiscreteLp*

Space to which the calling operator maps

`out` : *DiscreteLpVector*, optional

Vector in the projection space to which the result is written. If `None` creates an element in the projection space `proj_space`

Returns`out` : `proj_space` *element*

Projection data resulting from the application of the projector

astra_cuda

Backend for ASTRA using CUDA

Functions

<code>astra_cuda_back_projector</code> (<i>proj_data</i> , ..., [<i>out</i>])	Run an ASTRA backward projection on the given data using the GPU.
<code>astra_cuda_forward_projector</code> (<i>vol_data</i> , ...)	Run an ASTRA forward projection on the given data using the GPU.

astra_cuda_back_projector

`odl.tomo.backends.astra_cuda.astra_cuda_back_projector` (*proj_data*, *geometry*,
reco_space, *out=None*)

Run an ASTRA backward projection on the given data using the GPU.

Parameters`proj_data` : *DiscreteLp* element

Projection data to which the backward projector is applied

`geometry` : *Geometry*

Geometry defining the tomographic setup

`reco_space` : *DiscreteLp*

Space to which the calling operator maps

`out` : *DiscreteLpVector*, optional

Vector in the reconstruction space to which the result is written. If `None` creates an element in the reconstruction space `reco_space`

Returns`out` : `reco_space` element

Reconstruction data resulting from the application of the backward projector

astra_cuda_forward_projector

`odl.tomo.backends.astra_cuda.astra_cuda_forward_projector` (*vol_data*, *geometry*,
proj_space, *out=None*)

Run an ASTRA forward projection on the given data using the GPU.

Parameters`vol_data` : *DiscreteLpVector*

Volume data to which the projector is applied

geometry : *Geometry*

Geometry defining the tomographic setup

proj_space : *DiscreteLp*

Space to which the calling operator maps

out : *DiscreteLpVector*, optional

Vector in the projection space to which the result is written. If `None` creates an element in the projection space `proj_space`

Returns`out` : `proj_space` element

Projection data resulting from the application of the projector

astra_setup

Helper functions to prepare ASTRA algorithms.

This module contains utility functions to convert data structures from the ODL geometry representation to ASTRA's data structures, including:

- volume geometries
- projection geometries
- create vectors from geometries
- data arrays
- projectors
- algorithm configuration dictionaries

[ASTRA documentation on sourceforge.](#)

[ASTRA on github.](#)

Functions

<code>astra_algorithm</code> (<i>direction</i> , <i>ndim</i> , <i>vol_id</i> , ...)	Create an ASTRA algorithm object to run the projector.
<code>astra_conebeam_2d_geom_to_vec</code> (<i>geometry</i>)	Create vectors for ASTRA projection geometries from ODL geometry.

Continued on next page

Table 2.227 – continued from previous page

<code>astra_conebeam_3d_geom_to_vec(geometry)</code>	Create vectors for ASTRA projection geometries from ODL geometry.
<code>astra_data(astra_geom, datatype[, data, ndim])</code>	Create an ASTRA data structure.
<code>astra_parallel_3d_geom_to_vec(geometry)</code>	Create vectors for ASTRA projection geometries from ODL geometry.
<code>astra_projection_geometry(geometry)</code>	Create an ASTRA projection geometry from an ODL geometry object.
<code>astra_projector(vol_interp, astra_vol_geom, ...)</code>	Create an ASTRA projector configuration dictionary.
<code>astra_volume_geometry(discr_reco)</code>	Create an ASTRA volume geometry from the discretized domain.

astra_algorithm

`odl.tomo.backends.astra_setup.astra_algorithm(direction, ndim, vol_id, sino_id, proj_id, impl)`

Create an ASTRA algorithm object to run the projector.

Parameters`direction` : {‘forward’, ‘backward’}

Apply the forward projection if ‘forward’, otherwise the back projection

`ndim` : {2, 3}

Number of dimensions of the projector

`vol_id` : int

ASTRA ID of the volume data object

`sino_id` : int

ASTRA ID of the projection data object

`proj_id` : int

ASTRA ID of the projector

`impl` : {‘cpu’, ‘cuda’}

Implementation of the projector

Returns`id` : int

ASTRA internal ID for the new algorithm structure

astra_conebeam_2d_geom_to_vec

`odl.tomo.backends.astra_setup.astra_conebeam_2d_geom_to_vec(geometry)`

Create vectors for ASTRA projection geometries from ODL geometry.

The 2D vectors are used to create an ASTRA projection geometry for cone beam geometries (‘flat_vec’) with helical acquisition curves.

Output vectors:

Each row of vectors corresponds to a single projection, and consists of:(srcX, srcY, dX, dY, uX, uY) src : the ray source d : the center of the detector u : the vector between the centers of detector pixels 0 and 1

Parameters`geometry` : *Geometry*

The ODL geometry instance used to create the ASTRA geometry

Returns`vectors` : `numpy.ndarray`

Numpy array of shape (number of angles, 6)

astra_conebeam_3d_geom_to_vec

```
odl.tomo.backends.astra_setup.astra_conebeam_3d_geom_to_vec(geometry)
```

Create vectors for ASTRA projection geometries from ODL geometry.

The 3D vectors are used to create an ASTRA projection geometry for cone beam geometries ('cone_vec') with helical acquisition curves.

Output vectors:

Each row of vectors corresponds to a single projection, and consists of:(srcX, srcY, srcZ, dX, dY, dZ, uX, uY, uZ, vX, vY, vZ): src : the ray source d : the center of the detector u : the vector from detector pixel (0,0) to (0,1) v : the vector from detector pixel (0,0) to (1,0)

Parametersgeometry : *Geometry*

The ODL geometry instance used to create the ASTRA geometry

Returnsvectors : `numpy.ndarray`

Numpy array of shape (number of angles, 12)

astra_data

```
odl.tomo.backends.astra_setup.astra_data(astra_geom, datatype, data=None, ndim=2)
```

Create an ASTRA data structure.

Parametersastra_geom : dict

ASTRA geometry object for the data creator, must correspond to the given data type

datatype : {'volume', 'projection'}

Type of the data container

data : *DiscreteIpVector*, optional

Data for the initialization of the data structure. If `None` creates an ASTRA data object filled with zeros

ndim : {2, 3}, optional

Dimension of the data. If data is not `None`, this parameter has no effect.

Returnsid : int

ASTRA internal ID for the new data structure

astra_parallel_3d_geom_to_vec

```
odl.tomo.backends.astra_setup.astra_parallel_3d_geom_to_vec(geometry)
```

Create vectors for ASTRA projection geometries from ODL geometry.

The 3D vectors are used to create an ASTRA projection geometry for parallel beam geometries ('parallel3d_vec').

Output vectors:

Each row of vectors corresponds to a single projection, and consists of:(rayX, rayY, rayZ, dX, dY, dZ, uX, uY, uZ, vX, vY, vZ) ray : the ray direction d : the center of the detector u : the vector from detector pixel (0,0) to (0,1) v : the vector from detector pixel (0,0) to (1,0)

Parametersgeometry : *Geometry*

The ODL geometry instance used to create the ASTRA geometry

Returns`vectors` : `numpy.ndarray`

Numpy array of shape (number of angles, 12)

astra_projection_geometry

`odl.tomo.backends.astra_setup.astra_projection_geometry(geometry)`

Create an ASTRA projection geometry from an ODL geometry object.

As of ASTRA version 1.7, the length values are not required any more to be rescaled for 3D geometries and non-unit (but isotropic) voxel sizes.

Parameters`geometry` : instance of `Geometry`

The ODL geometry instance used to create the projection geometry

Returns`proj_geom` : `dict`

Dictionary defining the ASTRA projection geometry

astra_projector

`odl.tomo.backends.astra_setup.astra_projector(vol_interp, astra_vol_geom, as-
tra_proj_geom, ndim, impl)`

Create an ASTRA projector configuration dictionary.

Parameters`vol_interp` : {'nearest', 'linear'}

Interpolation type of the volume discretization

astra_vol_geom : `dict`

ASTRA volume geometry dictionary

astra_proj_geom : `dict`

ASTRA projection geometry dictionary

ndim : {2, 3}

Number of dimensions of the projector

impl : {'cpu', 'cuda'}

Implementation of the projector

Returns`proj_id` : `int`

ASTRA reference ID to the ASTRA dict with initialized 'type' key

astra_volume_geometry

`odl.tomo.backends.astra_setup.astra_volume_geometry(discr_reco)`

Create an ASTRA volume geometry from the discretized domain.

From the ASTRA documentation:

In all 3D geometries, the coordinate system is defined around the reconstruction volume. The center of the reconstruction volume is the origin, and the sides of the voxels in the volume have length 1.

All dimensions in the projection geometries are relative to this unit length.

Parameters`discr_reco` : *DiscreteLp*

Discretization of an L2 space on the reconstruction domain. It must be 2- or 3-dimensional and sampled by a regular grid.

Returns `astra_geom`: dict

The ASTRA volume geometry

Raises `NotImplementedError`

if the cell sizes are not the same in each dimension

stir_bindings

Back-end for STIR: Software for Tomographic Reconstruction

Back and forward projectors for PET.

`ForwardProjectorByBinWrapper` and `BackProjectorByBinWrapper` are general objects of STIR projectors and back-projectors, these can be used to wrap a given projector.

`stir_projector_from_file` allows users a easy way to create a `ForwardProjectorByBinWrapper` by giving file paths to the required templates.

References See the [STIR webpage](#) for more information and the [STIR doc](#) for info on the STIR classes used here.

Classes

<code>BackProjectorByBinWrapper(dom, ran, volume, ...)</code>	A back projector using STIR.
<code>ForwardProjectorByBinWrapper(dom, ran, ...)</code>	A forward projector using STIR.
<code>StirVerbosity(verbosity)</code>	Context manager setting STIR verbosity to a fixed level.

BackProjectorByBinWrapper

```
class odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper(dom, ran, volume, proj_data,
                                                                back_projector=None,
                                                                adjoint=None)
```

Bases: `odl.operator.operator.Operator`

A back projector using STIR.

Attributes

<code>adjoint</code>	The operator adjoint (abstract).
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

BackProjectorByBinWrapper.adjoint

`BackProjectorByBinWrapper.adjoint`

The operator adjoint (abstract).

Raises `OpNotImplementedError`

Since the adjoint cannot be default implemented.

BackProjectorByBinWrapper.domain

`BackProjectorByBinWrapper.domain`

Set of objects on which this operator can be evaluated.

BackProjectorByBinWrapper.inverse

`BackProjectorByBinWrapper.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

BackProjectorByBinWrapper.is_functional

`BackProjectorByBinWrapper.is_functional`

True if the this operator's range is a *Field*.

BackProjectorByBinWrapper.is_linear

`BackProjectorByBinWrapper.is_linear`

True if this operator is linear.

BackProjectorByBinWrapper.range

`BackProjectorByBinWrapper.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>__call__(projections, out)</code>	Back project.
<code>derivative(point)</code>	Return the operator derivative at point.

BackProjectorByBinWrapper.__call__

`BackProjectorByBinWrapper.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call__` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

BackProjectorByBinWrapper.__call

`BackProjectorByBinWrapper.__call` (*projections, out*)

Back project.

BackProjectorByBinWrapper.derivative

`BackProjectorByBinWrapper.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*dom, ran, volume, proj_data, back_projector=None, adjoint=None*)

Initialize a new instance.

Parameters *dom* : *DiscreteLp*

Projection space. Needs to have the same shape as `proj_data.to_array().shape()`.

ran : *DiscreteLp*

Volume of the projection. Needs to have the same shape as `volume.shape()`.

volume : `stir.FloatVoxelsOnCartesianGrid`

The stir volume to use in the forward projection

proj_data : `stir.ProjData`

The stir description of the projection.

back_projector : `stir.BackProjectorByBin`, optional

A pre-initialized back-projector.

adjoint : `ForwardProjectorByBinWrapper`, optional

A pre-initialized adjoint.

Notes

See [STIR doc](#) for info on the STIR classes.

References

ForwardProjectorByBinWrapper

```
class odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper(dom,      ran,
                                                                    volume,
                                                                    proj_data,
                                                                    projec-
                                                                    tor=None,
                                                                    ad-
                                                                    joint=None)
```

Bases: `odl.operator.operator.Operator`

A forward projector using STIR.

Uses “ForwardProjectorByBinUsingProjMatrixByBin” as a projector.

Attributes

<code>adjoint</code>	The back-projector associated with this operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator’s range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

ForwardProjectorByBinWrapper.adjoint

`ForwardProjectorByBinWrapper.adjoint`

The back-projector associated with this operator.

ForwardProjectorByBinWrapper.domain

`ForwardProjectorByBinWrapper.domain`

Set of objects on which this operator can be evaluated.

ForwardProjectorByBinWrapper.inverse

`ForwardProjectorByBinWrapper.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ForwardProjectorByBinWrapper.is_functional`ForwardProjectorByBinWrapper.is_functional`True if the this operator's range is a *Field*.**ForwardProjectorByBinWrapper.is_linear**`ForwardProjectorByBinWrapper.is_linear`

True if this operator is linear.

ForwardProjectorByBinWrapper.range`ForwardProjectorByBinWrapper.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(volume, out)</code>	Forward project a volume.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

ForwardProjectorByBinWrapper.__call__`ForwardProjectorByBinWrapper.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.**out** : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns**`out` : *Operator.range element*Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.**See also:**`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

ForwardProjectorByBinWrapper._call

`ForwardProjectorByBinWrapper._call` (*volume*, *out*)

Forward project a volume.

ForwardProjectorByBinWrapper.derivative

`ForwardProjectorByBinWrapper.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

__init__ (*dom*, *ran*, *volume*, *proj_data*, *projector=None*, *adjoint=None*)

Initialize a new instance.

Parameters*dom* : *DiscreteIp*

Volume of the projection. Needs to have the same shape as `volume.shape()`.

ran : *DiscreteIp*

Projection space. Needs to have the same shape as `proj_data.to_array().shape()`.

volume : `stir.FloatVoxelsOnCartesianGrid`

The stir volume to use in the forward projection

proj_data : `stir.ProjData`

The stir description of the projection.

projector : `stir.ForwardProjectorByBin`, optional

A pre-initialized projector.

adjoint : *BackProjectorByBinWrapper*, optional

A pre-initialized adjoint.

StirVerbosity

class `odl.tomo.backends.stir_bindings.StirVerbosity` (*verbosity*)

Bases: `object`

Context manager setting STIR verbosity to a fixed level.

Methods

```
__eq__ Return self==value.
```

```
__init__(verbosity)
```

Functions

```
stir_projector_from_file(volume_file, ...) Create a STIR projector from given template files.
```

stir_projector_from_file

```
odl.tomo.backends.stir_bindings.stir_projector_from_file(volume_file, projection_file)
```

Create a STIR projector from given template files.

Parameters`volume_file`: str

Full file path to the STIR input file containing information on the volume. This is usually a '.hv' file. For STIR reasons, a '.v' file is also needed.

`projection_file`: str

Full file path to the STIR input file with information on the projection data. This is usually a '.hs' file. For STIR reasons, a '.s' file is also needed.

Returns`projector`: *ForwardProjectorByBinWrapper*

A STIR forward projector.

geometry

Modules

conebeam

Cone beam geometries in 3 dimensions.

Classes

<i>CircularConeFlatGeometry</i> (<code>apart</code> , <code>dpart</code> , ...)	Cone beam geometry with circular source curve and flat detector.
<i>HelicalConeFlatGeometry</i> (<code>apart</code> , <code>dpart</code> , ..., ...)	Cone beam geometry with helical source curve and flat detector.

CircularConeFlatGeometry

```
class odl.tomo.geometry.conebeam.CircularConeFlatGeometry(apart, dpart, src_radius, det_radius, axis=[0, 0, 1], **kwargs)
```

Bases: *odl.tomo.geometry.conebeam.HelicalConeFlatGeometry*

Cone beam geometry with circular source curve and flat detector.

The source moves along a circle with radius `src_radius` in the plane perpendicular to a fixed `axis`. The detector reference point is opposite to the source, i.e. in the same plane on a circle with radius `det_rad` at maximum distance to the source. This implies that it lies on the line through the source point and the intersection

of the `axis` with the azimuthal plane.

The motion parameter is the 1d rotation angle parameterizing source and detector positions simultaneously.

In the standard configuration, the rotation axis is $(0, 0, 1)$, the initial source-to-detector vector is $(1, 0, 0)$, and the initial detector axes are $[(0, 1, 0), (0, 0, 1)]$.

See also:

HelicalConeFlatGeometry General case with motion in z direction

Attributes

<code>axis</code>	The normalized axis of rotation, a 3-element vector.
<code>det_grid</code>	Sampling grid of <code>det_params</code> .
<code>det_params</code>	Continuous detector parameter range, an <i>IntervalProd</i> .
<code>det_partition</code>	Partition of the detector parameter set into subsets.
<code>det_radius</code>	Detector circle radius of this geometry.
<code>detector</code>	Detector representation of this geometry.
<code>grid</code>	Joined sampling grid for motion and detector.
<code>implementation_cache</code>	Dictionary acting as a cache for this geometry.
<code>motion_grid</code>	Sampling grid of <code>motion_params</code> .
<code>motion_params</code>	Continuous motion parameter range, an <i>IntervalProd</i> .
<code>motion_partition</code>	Partition of the motion parameter set into subsets.
<code>ndim</code>	The number of dimensions of the geometry.
<code>params</code>	Joined parameter set for motion and detector.
<code>partition</code>	Joined parameter set partition for motion and detector.
<code>pitch</code>	Constant vertical distance traversed in a full rotation.
<code>pitch_offset</code>	Vertical offset at <code>angle=0</code>
<code>src_radius</code>	Source circle radius of this geometry.

CircularConeFlatGeometry.axis

`CircularConeFlatGeometry.axis`

The normalized axis of rotation, a 3-element vector.

CircularConeFlatGeometry.det_grid

`CircularConeFlatGeometry.det_grid`

Sampling grid of `det_params`.

CircularConeFlatGeometry.det_params

`CircularConeFlatGeometry.det_params`

Continuous detector parameter range, an *IntervalProd*.

CircularConeFlatGeometry.det_partition

`CircularConeFlatGeometry.det_partition`

Partition of the detector parameter set into subsets.

CircularConeFlatGeometry.det_radius

`CircularConeFlatGeometry.det_radius`

Detector circle radius of this geometry.

CircularConeFlatGeometry.detector`CircularConeFlatGeometry.detector`

Detector representation of this geometry.

CircularConeFlatGeometry.grid`CircularConeFlatGeometry.grid`

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

CircularConeFlatGeometry.implementation_cache`CircularConeFlatGeometry.implementation_cache`

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns`implementations`: dict**CircularConeFlatGeometry.motion_grid**`CircularConeFlatGeometry.motion_grid`Sampling grid of *motion_params*.**CircularConeFlatGeometry.motion_params**`CircularConeFlatGeometry.motion_params`Continuous motion parameter range, an *IntervalProd*.**CircularConeFlatGeometry.motion_partition**`CircularConeFlatGeometry.motion_partition`

Partition of the motion parameter set into subsets.

CircularConeFlatGeometry.ndim`CircularConeFlatGeometry.ndim`

The number of dimensions of the geometry.

CircularConeFlatGeometry.params`CircularConeFlatGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

CircularConeFlatGeometry.partition`CircularConeFlatGeometry.partition`

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**CircularConeFlatGeometry.pitch**`CircularConeFlatGeometry.pitch`

Constant vertical distance traversed in a full rotation.

CircularConeFlatGeometry.pitch_offset`CircularConeFlatGeometry.pitch_offset`

Vertical offset at angle=0

CircularConeFlatGeometry.src_radius`CircularConeFlatGeometry.src_radius`

Source circle radius of this geometry.

Methods

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angle)</code>	Return the detector reference point position at angle.
<code>det_to_src(mpar, dpar[, normalized])</code>	Vector pointing from a detector location to the source.
<code>rotation_matrix(angle)</code>	The detector rotation function.
<code>src_position(angle)</code>	Return the source position at angle.

CircularConeFlatGeometry.det_point_position`CircularConeFlatGeometry.det_point_position(mpar, dpar)`

The detector point position function.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns`pos` : `numpy.ndarray`, shape (*ndim*,)The source position, a *ndim*-dimensional vector**CircularConeFlatGeometry.det_refpoint**`CircularConeFlatGeometry.det_refpoint(angle)`

Return the detector reference point position at angle.

For an angle `phi`, the detector position is given by:

$$\text{ref}(\phi) = \text{det_rad} * \text{rot_matrix}(\phi) * \text{src_to_det_init} + (\text{pitch_offset} + \text{pitch} * \phi) * \text{axis}$$

where `src_to_det_init` is the initial unit vector pointing from source to detector.**Parameters**`angle` : floatRotation angle given in radians, must be contained in this geometry's *motion_params***Returns**`point` : `numpy.ndarray`, shape (3,)

Detector reference point corresponding to the given angle

See also:*rotation_matrix*

CircularConeFlatGeometry.det_to_src

`CircularConeFlatGeometry.det_to_src(mpar, dpar, normalized=True)`

Vector pointing from a detector location to the source.

A function of the motion and detector parameters.

The default implementation uses the `det_point_position` and `src_position` functions. Implementations can override this, for example if no source position is given.

Parameters`mpar` : element of motion parameters `motion_params`

Motion parameter at which to evaluate

dpar : element of detector parameters `det_params`

Detector parameter at which to evaluate

normalized : bool, optional

If True, return a normalized (unit) vector.

Returns`vec` : `numpy.ndarray`, shape (`ndim`,)

(Unit) vector pointing from the detector to the source

CircularConeFlatGeometry.rotation_matrix

`CircularConeFlatGeometry.rotation_matrix(angle)`

The detector rotation function.

Returns the matrix for rotating a vector in 3d by an angle `angle` about the rotation axis given by the property `axis` according to the right hand rule.

The matrix is computed according to [Rodrigues' rotation formula](#).

Parameters`angle` : float

The motion parameter given in radian. It must be contained in this geometry's `motion_params`.

Returns`rot_mat` : `numpy.ndarray`, shape (3, 3)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

CircularConeFlatGeometry.src_position

`CircularConeFlatGeometry.src_position(angle)`

Return the source position at angle.

For an angle `phi`, the source position is given by:

$$\text{src}(\phi) = -\text{src_rad} * \text{rot_matrix}(\phi) * \text{src_to_det_init} + (\text{pitch_offset} + \text{pitch} * \phi) * \text{axis}$$

where `src_to_det_init` is the initial unit vector pointing from source to detector.

Parameters`angle` : float

Rotation angle given in radians, must be contained in this geometry's `motion_params`

Returns`point` : `numpy.ndarray`, shape (3,)

Detector reference point corresponding to the given angle

See also:

`rotation_matrix`
`__init__` (*apart*, *dpart*, *src_radius*, *det_radius*, *axis*=[0, 0, 1], ***kwargs*)
 Initialize a new instance.

Parameters
apart : 1-dim. *RectPartition*

Partition of the angle interval

dpart : 2-dim. *RectPartition*

Partition of the detector parameter rectangle

src_radius : nonnegative float

Radius of the source circle

det_radius : nonnegative float

Radius of the detector circle

axis : *array-like*, shape (3,), optional

Fixed rotation axis, the symmetry axis of the helix

src_to_det_init : *array-like*, shape (2,), optional

Initial state of the vector pointing from source to detector reference point. The zero vector is not allowed. By default, a *perpendicular_vector* to *axis* is used.

det_init_axes : 2-tuple of *array-like* (shape (2,)), optional

Initial axes defining the detector orientation. By default, the normalized cross product of *axis* and *src_to_det_init* is used as first axis and *axis* as second.

HelicalConeFlatGeometry

class odl.tomo.geometry.conebeam.**HelicalConeFlatGeometry** (*apart*, *dpart*, *src_radius*, *det_radius*, *pitch*, *axis*=[0, 0, 1], ***kwargs*)

Bases: *odl.tomo.geometry.geometry.DivergentBeamGeometry*,
odl.tomo.geometry.geometry.AxisOrientedGeometry

Cone beam geometry with helical source curve and flat detector.

The source moves along a spiral oriented along a fixed *axis*, with radius *src_radius* in the azimuthal plane and a given *pitch*. The detector reference point is opposite to the source, i.e. in the point at distance *src_rad* + *det_rad* on the line in the azimuthal plane through the source point and *axis*.

The motion parameter is the 1d rotation angle parameterizing source and detector positions simultaneously.

In the standard configuration, the rotation axis is (0, 0, 1), the initial source-to-detector vector is (1, 0, 0), and the initial detector axes are [(0, 1, 0), (0, 0, 1)].

See also:

CircularConeFlatGeometry Case with zero pitch

Attributes

<i>axis</i>	The normalized axis of rotation, a 3-element vector.
Continued on next page	

Table 2.238 – continued from previous page

<code>det_grid</code>	Sampling grid of <code>det_params</code> .
<code>det_params</code>	Continuous detector parameter range, an <i>IntervalProd</i> .
<code>det_partition</code>	Partition of the detector parameter set into subsets.
<code>det_radius</code>	Detector circle radius of this geometry.
<code>detector</code>	Detector representation of this geometry.
<code>grid</code>	Joined sampling grid for motion and detector.
<code>implementation_cache</code>	Dictionary acting as a cache for this geometry.
<code>motion_grid</code>	Sampling grid of <code>motion_params</code> .
<code>motion_params</code>	Continuous motion parameter range, an <i>IntervalProd</i> .
<code>motion_partition</code>	Partition of the motion parameter set into subsets.
<code>ndim</code>	The number of dimensions of the geometry.
<code>params</code>	Joined parameter set for motion and detector.
<code>partition</code>	Joined parameter set partition for motion and detector.
<code>pitch</code>	Constant vertical distance traversed in a full rotation.
<code>pitch_offset</code>	Vertical offset at <code>angle=0</code>
<code>src_radius</code>	Source circle radius of this geometry.

HelicalConeFlatGeometry.axis`HelicalConeFlatGeometry.axis`

The normalized axis of rotation, a 3-element vector.

HelicalConeFlatGeometry.det_grid`HelicalConeFlatGeometry.det_grid`Sampling grid of `det_params`.**HelicalConeFlatGeometry.det_params**`HelicalConeFlatGeometry.det_params`Continuous detector parameter range, an *IntervalProd*.**HelicalConeFlatGeometry.det_partition**`HelicalConeFlatGeometry.det_partition`

Partition of the detector parameter set into subsets.

HelicalConeFlatGeometry.det_radius`HelicalConeFlatGeometry.det_radius`

Detector circle radius of this geometry.

HelicalConeFlatGeometry.detector`HelicalConeFlatGeometry.detector`

Detector representation of this geometry.

HelicalConeFlatGeometry.grid`HelicalConeFlatGeometry.grid`

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

HelicalConeFlatGeometry.implementation_cache

`HelicalConeFlatGeometry.implementation_cache`

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns`implementations` : dict

HelicalConeFlatGeometry.motion_grid

`HelicalConeFlatGeometry.motion_grid`

Sampling grid of *motion_params*.

HelicalConeFlatGeometry.motion_params

`HelicalConeFlatGeometry.motion_params`

Continuous motion parameter range, an *IntervalProd*.

HelicalConeFlatGeometry.motion_partition

`HelicalConeFlatGeometry.motion_partition`

Partition of the motion parameter set into subsets.

HelicalConeFlatGeometry.ndim

`HelicalConeFlatGeometry.ndim`

The number of dimensions of the geometry.

HelicalConeFlatGeometry.params

`HelicalConeFlatGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

HelicalConeFlatGeometry.partition

`HelicalConeFlatGeometry.partition`

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.

HelicalConeFlatGeometry.pitch

`HelicalConeFlatGeometry.pitch`

Constant vertical distance traversed in a full rotation.

HelicalConeFlatGeometry.pitch_offset

`HelicalConeFlatGeometry.pitch_offset`

Vertical offset at `angle=0`

HelicalConeFlatGeometry.src_radius

`HelicalConeFlatGeometry.src_radius`

Source circle radius of this geometry.

Methods

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angle)</code>	Return the detector reference point position at angle.
<code>det_to_src(mpar, dpar[, normalized])</code>	Vector pointing from a detector location to the source.
<code>rotation_matrix(angle)</code>	The detector rotation function.
<code>src_position(angle)</code>	Return the source position at angle.

HelicalConeFlatGeometry.det_point_position

HelicalConeFlatGeometry.**det_point_position** (*mpar*, *dpar*)

The detector point position function.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns*pos* : `numpy.ndarray`, shape (*ndim*,)

The source position, a *ndim*-dimensional vector

HelicalConeFlatGeometry.det_refpoint

HelicalConeFlatGeometry.**det_refpoint** (*angle*)

Return the detector reference point position at angle.

For an angle *phi*, the detector position is given by:

```
ref(phi) = det_rad * rot_matrix(phi) * src_to_det_init +
           (pitch_offset + pitch * phi) * axis
```

where *src_to_det_init* is the initial unit vector pointing from source to detector.

Parameters*angle* : float

Rotation angle given in radians, must be contained in this geometry's *motion_params*

Returns*point* : `numpy.ndarray`, shape (3,)

Detector reference point corresponding to the given angle

See also:

rotation_matrix

HelicalConeFlatGeometry.det_to_src

HelicalConeFlatGeometry.**det_to_src** (*mpar*, *dpar*, *normalized=True*)

Vector pointing from a detector location to the source.

A function of the motion and detector parameters.

The default implementation uses the *det_point_position* and *src_position* functions. Implementations can override this, for example if no source position is given.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

normalized : bool, optional

If True, return a normalized (unit) vector.

Returns**vec** : `numpy.ndarray`, shape (*ndim*,)

(Unit) vector pointing from the detector to the source

HelicalConeFlatGeometry.rotation_matrix

`HelicalConeFlatGeometry.rotation_matrix` (*angle*)

The detector rotation function.

Returns the matrix for rotating a vector in 3d by an angle *angle* about the rotation axis given by the property *axis* according to the right hand rule.

The matrix is computed according to [Rodrigues' rotation formula](#).

Parameters**angle** : float

The motion parameter given in radian. It must be contained in this geometry's *motion_params*.

Returns**rot_mat** : `numpy.ndarray`, shape (3, 3)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

HelicalConeFlatGeometry.src_position

`HelicalConeFlatGeometry.src_position` (*angle*)

Return the source position at angle.

For an angle *phi*, the source position is given by:

$$\text{src}(\phi) = -\text{src_rad} * \text{rot_matrix}(\phi) * \text{src_to_det_init} + (\text{pitch_offset} + \text{pitch} * \phi) * \text{axis}$$

where *src_to_det_init* is the initial unit vector pointing from source to detector.

Parameters**angle** : float

Rotation angle given in radians, must be contained in this geometry's *motion_params*

Returns**point** : `numpy.ndarray`, shape (3,)

Detector reference point corresponding to the given angle

See also:

rotation_matrix

__init__ (*apart*, *dpart*, *src_radius*, *det_radius*, *pitch*, *axis*=[0, 0, 1], ***kwargs*)

Initialize a new instance.

Parameters**apart** : 1-dim. *RectPartition*

Partition of the angle interval

dpart : 2-dim. *RectPartition*

Partition of the detector parameter rectangle

src_radius : nonnegative float

Radius of the source circle

det_radius : nonnegative float

Radius of the detector circle

pitch : float

Constant vertical distance that a point on the helix traverses when increasing the angle parameter by $2 * \pi$

axis : *array-like*, shape (3,), optional

Fixed rotation axis, the symmetry axis of the helix

src_to_det_init : *array-like*, shape (2,), optional

Initial state of the vector pointing from source to detector reference point. The zero vector is not allowed. By default, a *perpendicular_vector* to *axis* is used.

det_init_axes : 2-tuple of *array-like* (shape (2,)), optional

Initial axes defining the detector orientation. By default, the normalized cross product of *axis* and *src_to_det_init* is used as first axis and *axis* as second.

pitch_offset : float, optional

Offset along the axis at angle=0

detector

Detectors.

Classes

<i>CircleSectionDetector</i> (part, circ_rad)	A 1d detector given by a section of a circle.
<i>Detector</i> (part)	Abstract detector class.
<i>Flat1dDetector</i> (part, axis)	A 1d line detector aligned with <i>axis</i> .
<i>Flat2dDetector</i> (part, axes)	A 2d flat panel detector aligned with <i>axes</i> .
<i>FlatDetector</i> (part)	Abstract class for flat detectors in 2 and 3 dimensions.

CircleSectionDetector

class odl.tomo.geometry.detector.**CircleSectionDetector** (part, circ_rad)

Bases: odl.tomo.geometry.detector.Detector

A 1d detector given by a section of a circle.

The reference circular section is part of a circle with radius r , which is shifted by the vector $(-r, 0)$ such that the parameter value 0 results in the detector reference point $(0, 0)$.

Attributes

<i>circ_rad</i>	Circle radius of this detector.
<i>grid</i>	Sampling grid of the parameters.
<i>ndim</i>	Number of dimensions of this detector (0, 1 or 2).
<i>params</i>	Surface parameter set of this detector.
<i>partition</i>	Partition of the detector parameter set into subsets.
<i>shape</i>	Number of subsets (pixels) of the detector per axis.
<i>size</i>	Total number of pixels.

CircleSectionDetector.circ_rad`CircleSectionDetector.circ_rad`

Circle radius of this detector.

CircleSectionDetector.grid`CircleSectionDetector.grid`

Sampling grid of the parameters.

CircleSectionDetector.ndim`CircleSectionDetector.ndim`

Number of dimensions of this detector (0, 1 or 2).

CircleSectionDetector.params`CircleSectionDetector.params`

Surface parameter set of this detector.

CircleSectionDetector.partition`CircleSectionDetector.partition`

Partition of the detector parameter set into subsets.

CircleSectionDetector.shape`CircleSectionDetector.shape`

Number of subsets (pixels) of the detector per axis.

CircleSectionDetector.size`CircleSectionDetector.size`

Total number of pixels.

Methods

<code>__eq__</code>	Return self==value.
<code>surface(param)</code>	The parametrization of the detector reference surface.
<code>surface_deriv(param)</code>	The partial derivative(s) of the surface parametrization.
<code>surface_measure(param)</code>	The constant density function of the surface measure.

CircleSectionDetector.surface`CircleSectionDetector.surface(param)`

The parametrization of the detector reference surface.

Parameters*param* : element of *params*

The parameter value where to evaluate the function

CircleSectionDetector.surface_deriv

`CircleSectionDetector.surface_deriv(param)`

The partial derivative(s) of the surface parametrization.

Parameters*param* : element of *params*

The parameter value where to evaluate the function

CircleSectionDetector.surface_measure

`CircleSectionDetector.surface_measure(param)`

The constant density function of the surface measure.

Parameters*param* : element of *params*

The parameter value where to evaluate the function

Returns*measure* : float

The constant density r , equal to the length of the tangent to the detector circle at any point

`__init__(part, circ_rad)`

Initialize a new instance.

Parameters*part* : 1-dim. *RectPartition*

Partition of the parameter interval, corresponding to the angle sections along the line

circ_rad : positive float

Radius of the circle along which the detector is curved

Detector

`class odl.tomo.geometry.detector.Detector(part)`

Bases: object

Abstract detector class.

A detector is described by

- a set of parameters for surface parametrization (including sampling),
- a function mapping a surface parameter to the location of a detector point relative to its reference point,
- optionally a surface measure function.

Attributes

<i>grid</i>	Sampling grid of the parameters.
<i>ndim</i>	Number of dimensions of this detector (0, 1 or 2).
<i>params</i>	Surface parameter set of this detector.
<i>partition</i>	Partition of the detector parameter set into subsets.
<i>shape</i>	Number of subsets (pixels) of the detector per axis.
<i>size</i>	Total number of pixels.

Detector.grid`Detector.grid`

Sampling grid of the parameters.

Detector.ndim`Detector.ndim`

Number of dimensions of this detector (0, 1 or 2).

Detector.params`Detector.params`

Surface parameter set of this detector.

Detector.partition`Detector.partition`

Partition of the detector parameter set into subsets.

Detector.shape`Detector.shape`

Number of subsets (pixels) of the detector per axis.

Detector.size`Detector.size`

Total number of pixels.

Methods

<code>__eq__</code>	Return self==value.
<code>surface(param)</code>	Parametrization of the detector reference surface.
<code>surface_deriv(param)</code>	Partial derivative(s) of the surface parametrization.
<code>surface_measure(param)</code>	Density function of the surface measure.

Detector.surface`Detector.surface (param)`

Parametrization of the detector reference surface.

Parameters`param` : element of *params*

Parameter value where to evaluate the function

Returnspoint :Spatial location of the detector point corresponding to `param`**Detector.surface_deriv**`Detector.surface_deriv (param)`

Partial derivative(s) of the surface parametrization.

Parameters`param` : element of *params*

The parameter value where to evaluate the function

Returns*deriv* :

Vector (*ndim*=1) or sequence of vectors corresponding to the partial derivatives at *param*

Detector.surface_measure

`Detector.surface_measure(param)`

Density function of the surface measure.

This is the default implementation relying on the `surface_deriv` method. For *ndim* == 1, the density is given by the *Arc length*, for *ndim* == 2, it is the length of the cross product of the partial derivatives of the parametrization, see Wikipedia's *Surface area* article.

Parameters*param* : element of *params*

The parameter value where to evaluate the function

Returns*measure* : float

The density value at the given parameter

https://en.wikipedia.org/wiki/Curve#Lengths_of_curves

https://en.wikipedia.org/wiki/Surface_area

`__init__(part)`

Initialize a new instance.

Parameters*part* : *RectPartition*

Partition of the detector parameter set (pixelization). It determines dimension, parameter range and discretization.

Flat1dDetector

class `odl.tomo.geometry.detector.Flat1dDetector(part, axis)`

Bases: `odl.tomo.geometry.detector.FlatDetector`

A 1d line detector aligned with *axis*.

Attributes

<i>axis</i>	Normalized principal axis of the detector.
<i>grid</i>	Sampling grid of the parameters.
<i>ndim</i>	Number of dimensions of this detector (0, 1 or 2).
<i>normal</i>	Unit vector perpendicular to the detector.
<i>params</i>	Surface parameter set of this detector.
<i>partition</i>	Partition of the detector parameter set into subsets.
<i>shape</i>	Number of subsets (pixels) of the detector per axis.
<i>size</i>	Total number of pixels.

Flat1dDetector.axis

`Flat1dDetector.axis`

Normalized principal axis of the detector.

Flat1dDetector.grid`Flat1dDetector.grid`

Sampling grid of the parameters.

Flat1dDetector.ndim`Flat1dDetector.ndim`

Number of dimensions of this detector (0, 1 or 2).

Flat1dDetector.normal`Flat1dDetector.normal`

Unit vector perpendicular to the detector.

Its orientation is chosen such that the system axis, `normal` is right-handed.**Flat1dDetector.params**`Flat1dDetector.params`

Surface parameter set of this detector.

Flat1dDetector.partition`Flat1dDetector.partition`

Partition of the detector parameter set into subsets.

Flat1dDetector.shape`Flat1dDetector.shape`

Number of subsets (pixels) of the detector per axis.

Flat1dDetector.size`Flat1dDetector.size`

Total number of pixels.

Methods

<code>__eq__</code>	Return self==value.
<code>surface(param)</code>	The parametrization of the (1d) detector reference surface.
<code>surface_deriv([param])</code>	The derivative of the surface parametrization.
<code>surface_measure([param])</code>	The constant density function of the surface measure.

Flat1dDetector.surface`Flat1dDetector.surface` (*param*)

The parametrization of the (1d) detector reference surface.

The reference line segment is chosen to be aligned with the second coordinate axis, such that the parameter value 0 results in the reference point (0, 0).

Parameters*param* : element of *params*

The parameter value where to evaluate the function

Returns*point* : `numpy.ndarray`, shape (2,)

The point on the detector surface corresponding to the given parameters

Flat1dDetector.surface_deriv

`Flat1dDetector.surface_deriv` (*param=None*)

The derivative of the surface parametrization.

Parameters*param* : element of *params*, optional

The parameter value where to evaluate the function

Returns*derivative* : `numpy.ndarray`, shape (2,)

The constant derivative

Flat1dDetector.surface_measure

`Flat1dDetector.surface_measure` (*param=None*)

The constant density function of the surface measure.

Parameters*param* : element of *params*, optional

The parameter value where to evaluate the function

Returns*measure* : float

The constant density 1.0

__init__ (*part, axis*)

Initialize a new instance.

Parameters*part* : 1-dim. *RectPartition*

Partition of the parameter interval, corresponding to the line elements

axis : *array-like*, shape (2,)

Principal axis of the detector

Flat2dDetector

class `odl.tomo.geometry.detector.Flat2dDetector` (*part, axes*)

Bases: `odl.tomo.geometry.detector.FlatDetector`

A 2d flat panel detector aligned with *axes*.

Attributes

<i>axes</i>	Normalized principal axes of this detector as a 2-tuple.
<i>grid</i>	Sampling grid of the parameters.
<i>ndim</i>	Number of dimensions of this detector (0, 1 or 2).
<i>normal</i>	Unit vector perpendicular to this detector.
<i>params</i>	Surface parameter set of this detector.
<i>partition</i>	Partition of the detector parameter set into subsets.
<i>shape</i>	Number of subsets (pixels) of the detector per axis.
<i>size</i>	Total number of pixels.

Flat2dDetector.axes

`Flat2dDetector.axes`

Normalized principal axes of this detector as a 2-tuple.

Flat2dDetector.grid

`Flat2dDetector.grid`

Sampling grid of the parameters.

`Flat2dDetector.ndim`

`Flat2dDetector.ndim`

Number of dimensions of this detector (0, 1 or 2).

`Flat2dDetector.normal`

`Flat2dDetector.normal`

Unit vector perpendicular to this detector.

The orientation is chosen such that the triple `axes[0]`, `axes[1]`, `normal` form a right-hand system.

`Flat2dDetector.params`

`Flat2dDetector.params`

Surface parameter set of this detector.

`Flat2dDetector.partition`

`Flat2dDetector.partition`

Partition of the detector parameter set into subsets.

`Flat2dDetector.shape`

`Flat2dDetector.shape`

Number of subsets (pixels) of the detector per axis.

`Flat2dDetector.size`

`Flat2dDetector.size`

Total number of pixels.

Methods

<code>__eq__</code>	Return self==value.
<code>surface(param)</code>	Parametrization of the 2d detector reference surface.
<code>surface_deriv([param])</code>	The derivative of the surface parametrization.
<code>surface_measure([param])</code>	The constant density function of the surface measure.

`Flat2dDetector.surface`

`Flat2dDetector.surface` (*param*)

Parametrization of the 2d detector reference surface.

The reference plane segment is chosen to be aligned with the second and third coordinate axes, in this order, such that the parameter value (0, 0) results in the reference (0, 0, 0).

Parameters*param* : element of *params*

The parameter value where to evaluate the function

Returns*point* : `numpy.ndarray`, shape (3,)

The point on the detector surface corresponding to the given parameters

Flat2dDetector.surface_deriv

`Flat2dDetector.surface_deriv` (*param=None*)

The derivative of the surface parametrization.

Parameters*param* : element of *params*, optional

The parameter value where to evaluate the function

Returns*derivatives* : 2-tuple of `numpy.ndarray` (shape (3,))

The constant partial derivatives given by the detector axes

Flat2dDetector.surface_measure

`Flat2dDetector.surface_measure` (*param=None*)

The constant density function of the surface measure.

Parameters*param* : element of *params*, optional

The parameter value where to evaluate the function

Returns*measure* : float

The constant density 1.0

__init__ (*part, axes*)

Initialize a new instance.

Parameters*part* : 1-dim. *RectPartition*

Partition of the parameter interval, corresponding to the pixels

axes : 2-tuple of *array-like* (shape (3,))

Principal axes of the detector, e.g. [(0, 1, 0), (0, 0, 1)]

FlatDetector

class `odl.tomo.geometry.detector.FlatDetector` (*part*)

Bases: `odl.tomo.geometry.detector.Detector`

Abstract class for flat detectors in 2 and 3 dimensions.

Attributes

<i>grid</i>	Sampling grid of the parameters.
<i>ndim</i>	Number of dimensions of this detector (0, 1 or 2).
<i>params</i>	Surface parameter set of this detector.
<i>partition</i>	Partition of the detector parameter set into subsets.
<i>shape</i>	Number of subsets (pixels) of the detector per axis.
<i>size</i>	Total number of pixels.

FlatDetector.grid

`FlatDetector.grid`

Sampling grid of the parameters.

FlatDetector.ndim

`FlatDetector.ndim`

Number of dimensions of this detector (0, 1 or 2).

FlatDetector.params`FlatDetector.params`

Surface parameter set of this detector.

FlatDetector.partition`FlatDetector.partition`

Partition of the detector parameter set into subsets.

FlatDetector.shape`FlatDetector.shape`

Number of subsets (pixels) of the detector per axis.

FlatDetector.size`FlatDetector.size`

Total number of pixels.

Methods

<code>__eq__</code>	Return self==value.
<code>surface(param)</code>	Parametrization of the detector reference surface.
<code>surface_deriv(param)</code>	Partial derivative(s) of the surface parametrization.
<code>surface_measure([param])</code>	The constant density function of the surface measure.

FlatDetector.surface`FlatDetector.surface` (*param*)

Parametrization of the detector reference surface.

Parameters*param* : element of *params*

Parameter value where to evaluate the function

Returns*point* :Spatial location of the detector point corresponding to *param***FlatDetector.surface_deriv**`FlatDetector.surface_deriv` (*param*)

Partial derivative(s) of the surface parametrization.

Parameters*param* : element of *params*

The parameter value where to evaluate the function

Returns*deriv* :Vector (ndim=1) or sequence of vectors corresponding to the partial derivatives at *param***FlatDetector.surface_measure**`FlatDetector.surface_measure` (*param=None*)

The constant density function of the surface measure.

Parameters*param* : element of *params*, optional

The parameter value where to evaluate the function

Returnsmeasure : float

The constant density 1.0

`__init__`(part)

Initialize a new instance.

Parameterspart : *RectPartition*

Partition of the detector parameter set (pixelization). It determines dimension, parameter range and discretization.

fanbeam

Fanbeam geometries.

Classes

FanFlatGeometry(apart, dpart, src_radius, ...) Abstract 2d fan beam geometry with flat 1d detector.

FanFlatGeometry

class odl.tomo.geometry.fanbeam.**FanFlatGeometry**(apart, dpart, src_radius, det_radius, **kwargs)

Bases: *odl.tomo.geometry.geometry.DivergentBeamGeometry*

Abstract 2d fan beam geometry with flat 1d detector.

The source moves on a circle with radius `src_radius`, and the detector reference point is opposite to the source, i.e. at maximum distance, on a circle with radius `det_radius`. One of the two radii can be chosen as 0, which corresponds to a stationary source or detector, respectively.

The motion parameter is the 1d rotation angle parameterizing source and detector positions simultaneously.

In the standard configuration, the source and detector start on the first coordinate axis with vector $(1, 0)$ from source to detector, and the initial detector axis is $(0, 1)$.

Attributes

<i>det_grid</i>	Sampling grid of <i>det_params</i> .
<i>det_params</i>	Continuous detector parameter range, an <i>IntervalProd</i> .
<i>det_partition</i>	Partition of the detector parameter set into subsets.
<i>det_radius</i>	Detector circle radius of this geometry.
<i>detector</i>	Detector representation of this geometry.
<i>grid</i>	Joined sampling grid for motion and detector.
<i>implementation_cache</i>	Dictionary acting as a cache for this geometry.
<i>motion_grid</i>	Sampling grid of <i>motion_params</i> .
<i>motion_params</i>	Continuous motion parameter range, an <i>IntervalProd</i> .
<i>motion_partition</i>	Partition of the motion parameter set into subsets.
<i>ndim</i>	The number of dimensions of the geometry.
<i>params</i>	Joined parameter set for motion and detector.
<i>partition</i>	Joined parameter set partition for motion and detector.

Continued on next page

Table 2.252 – continued from previous page

<i>src_radius</i>	Source circle radius of this geometry.
<hr/>	
FanFlatGeometry.det_grid FanFlatGeometry. det_grid Sampling grid of <i>det_params</i> .	
FanFlatGeometry.det_params FanFlatGeometry. det_params Continuous detector parameter range, an <i>IntervalProd</i> .	
FanFlatGeometry.det_partition FanFlatGeometry. det_partition Partition of the detector parameter set into subsets.	
FanFlatGeometry.det_radius FanFlatGeometry. det_radius Detector circle radius of this geometry.	
FanFlatGeometry.detector FanFlatGeometry. detector Detector representation of this geometry.	
FanFlatGeometry.grid FanFlatGeometry. grid Joined sampling grid for motion and detector. By convention, the motion grid comes before the detector grid.	
FanFlatGeometry.implementation_cache FanFlatGeometry. implementation_cache Dictionary acting as a cache for this geometry. Intended for reuse of computations. Implementations that use this storage should take care of unique naming. Returns implementations : dict	
FanFlatGeometry.motion_grid FanFlatGeometry. motion_grid Sampling grid of <i>motion_params</i> .	
FanFlatGeometry.motion_params FanFlatGeometry. motion_params Continuous motion parameter range, an <i>IntervalProd</i> .	
FanFlatGeometry.motion_partition FanFlatGeometry. motion_partition Partition of the motion parameter set into subsets.	

FanFlatGeometry.ndimFanFlatGeometry.**ndim**

The number of dimensions of the geometry.

FanFlatGeometry.paramsFanFlatGeometry.**params**

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

FanFlatGeometry.partitionFanFlatGeometry.**partition**

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**FanFlatGeometry.src_radius**FanFlatGeometry.**src_radius**

Source circle radius of this geometry.

Methods

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angle)</code>	Return the detector reference point position at angle.
<code>det_to_src(mpar, dpar[, normalized])</code>	Vector pointing from a detector location to the source.
<code>rotation_matrix(angle)</code>	Return the rotation matrix for angle.
<code>src_position(angle)</code>	Return the source position at angle.

FanFlatGeometry.det_point_positionFanFlatGeometry.**det_point_position** (*mpar*, *dpar*)

The detector point position function.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns*pos* : `numpy.ndarray`, shape (*ndim*,)The source position, a *ndim*-dimensional vector**FanFlatGeometry.det_refpoint**FanFlatGeometry.**det_refpoint** (*angle*)

Return the detector reference point position at angle.

For an angle *phi*, the detector position is given by:

$$\text{ref}(\phi) = \text{det_rad} * \text{rot_matrix}(\phi) * \text{src_to_det_init}$$

where *src_to_det_init* is the initial unit vector pointing from source to detector.

Parameters`angle` : float

Rotation angle given in radians, must be contained in this geometry's `motion_params`

Returns`point` : `numpy.ndarray`, shape (2,)

Detector reference point corresponding to the given angle

See also:

`rotation_matrix`

FanFlatGeometry.det_to_src

`FanFlatGeometry.det_to_src` (`mpar`, `dpar`, `normalized=True`)

Vector pointing from a detector location to the source.

A function of the motion and detector parameters.

The default implementation uses the `det_point_position` and `src_position` functions. Implementations can override this, for example if no source position is given.

Parameters`mpar` : element of motion parameters `motion_params`

Motion parameter at which to evaluate

dpar : element of detector parameters `det_params`

Detector parameter at which to evaluate

normalized : bool, optional

If True, return a normalized (unit) vector.

Returns`vec` : `numpy.ndarray`, shape (`ndim`,)

(Unit) vector pointing from the detector to the source

FanFlatGeometry.rotation_matrix

`FanFlatGeometry.rotation_matrix` (`angle`)

Return the rotation matrix for angle.

For an angle `phi`, the matrix is given by:

```
rot(phi) = [[cos(phi), -sin(phi)],
            [sin(phi), cos(phi)]]
```

Parameters`angle` : float

Rotation angle given in radians, must be contained in this geometry's `motion_params`

Returns`rot` : `numpy.ndarray`, shape (2, 2)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system

FanFlatGeometry.src_position

FanFlatGeometry.**src_position**(*angle*)

Return the source position at angle.

For an angle ϕ , the source position is given by:

$$\text{src}(\phi) = -\text{src_rad} * \text{rot_matrix}(\phi) * \text{src_to_det_init}$$

where `src_to_det_init` is the initial unit vector pointing from source to detector.

Parameters`angle` : float

Rotation angle given in radians, must be contained in this geometry's `motion_params`

Returns`point` : `numpy.ndarray`, shape (2,)

Source position corresponding to the given angle

__init__(*apart*, *dpart*, *src_radius*, *det_radius*, ***kwargs*)

Initialize a new instance.

Parameters`apart` : 1-dim. `RectPartition`

Partition of the angle interval

dpart : 1-dim. `RectPartition`

Partition of the detector parameter interval

src_radius : nonnegative float

Radius of the source circle

det_radius : nonnegative float

Radius of the detector circle

src_to_det_init : *array-like*, shape (2,), optional

Initial state of the vector pointing from source to detector reference point. The zero vector is not allowed. Default: (1, 0).

det_init_axis : *array-like* (shape (2,)), optional

Initial axis defining the detector orientation. By default, a normalized *perpendicular_vector* to `src_to_det_init` is used.

geometry

Geometry base class.

Classes

<code>AxisOrientedGeometry</code> (axis)	Mixin class for 3d geometries oriented according to an axis.
<code>DivergentBeamGeometry</code> (ndim, motion_part, ...)	Abstract divergent beam geometry class.
<code>Geometry</code> (ndim, motion_part, detector)	Abstract geometry class.

AxisOrientedGeometry

class `odl.tomo.geometry.geometry.AxisOrientedGeometry`(*axis*)

Bases: `object`

Mixin class for 3d geometries oriented according to an axis.

Attributes

<code>axis</code>	The normalized axis of rotation, a 3-element vector.
-------------------	--

AxisOrientedGeometry.axis

`AxisOrientedGeometry.axis`

The normalized axis of rotation, a 3-element vector.

Methods

<code>__eq__</code>	Return self==value.
<code>rotation_matrix(angle)</code>	The detector rotation function.

AxisOrientedGeometry.rotation_matrix

`AxisOrientedGeometry.rotation_matrix` (*angle*)

The detector rotation function.

Returns the matrix for rotating a vector in 3d by an angle *angle* about the rotation axis given by the property *axis* according to the right hand rule.

The matrix is computed according to [Rodrigues' rotation formula](#).

Parameters*angle* : float

The motion parameter given in radian. It must be contained in this geometry's *motion_params*.

Returns*rot_mat* : `numpy.ndarray`, shape (3, 3)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

`__init__` (*axis*)

Initialize a new instance.

Parameters*axis* : 3-element *array-like*

Vector defining the fixed rotation axis after normalization

DivergentBeamGeometry

class `odl.tomo.geometry.geometry.DivergentBeamGeometry` (*ndim, motion_part, detector*)

Bases: `odl.tomo.geometry.geometry.Geometry`

Abstract divergent beam geometry class.

A divergent beam geometry is characterized by the presence of a point source.

Special cases include fan beam in 2d and cone beam in 3d.

Attributes

<code>det_grid</code>	Sampling grid of <code>det_params</code> .
<code>det_params</code>	Continuous detector parameter range, an <i>IntervalProd</i> .
<code>det_partition</code>	Partition of the detector parameter set into subsets.
<code>detector</code>	Detector representation of this geometry.
<code>grid</code>	Joined sampling grid for motion and detector.
<code>implementation_cache</code>	Dictionary acting as a cache for this geometry.
<code>motion_grid</code>	Sampling grid of <code>motion_params</code> .
<code>motion_params</code>	Continuous motion parameter range, an <i>IntervalProd</i> .
<code>motion_partition</code>	Partition of the motion parameter set into subsets.
<code>ndim</code>	The number of dimensions of the geometry.
<code>params</code>	Joined parameter set for motion and detector.
<code>partition</code>	Joined parameter set partition for motion and detector.

DivergentBeamGeometry.det_gridDivergentBeamGeometry.**det_grid**Sampling grid of `det_params`.**DivergentBeamGeometry.det_params**DivergentBeamGeometry.**det_params**Continuous detector parameter range, an *IntervalProd*.**DivergentBeamGeometry.det_partition**DivergentBeamGeometry.**det_partition**

Partition of the detector parameter set into subsets.

DivergentBeamGeometry.detectorDivergentBeamGeometry.**detector**

Detector representation of this geometry.

DivergentBeamGeometry.gridDivergentBeamGeometry.**grid**

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

DivergentBeamGeometry.implementation_cacheDivergentBeamGeometry.**implementation_cache**

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns `implementations : dict`

DivergentBeamGeometry.motion_gridDivergentBeamGeometry.**motion_grid**Sampling grid of `motion_params`.

DivergentBeamGeometry.motion_params`DivergentBeamGeometry.motion_params`Continuous motion parameter range, an *IntervalProd*.**DivergentBeamGeometry.motion_partition**`DivergentBeamGeometry.motion_partition`

Partition of the motion parameter set into subsets.

DivergentBeamGeometry.ndim`DivergentBeamGeometry.ndim`

The number of dimensions of the geometry.

DivergentBeamGeometry.params`DivergentBeamGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

DivergentBeamGeometry.partition`DivergentBeamGeometry.partition`

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**Methods**

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(mpar)</code>	The detector reference point function.
<code>det_to_src(mpar, dpar[, normalized])</code>	Vector pointing from a detector location to the source.
<code>rotation_matrix(mpar)</code>	The detector rotation function for calculating the detector reference position.
<code>src_position(mpar)</code>	The source position function.

DivergentBeamGeometry.det_point_position`DivergentBeamGeometry.det_point_position (mpar, dpar)`

The detector point position function.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns`pos` : `numpy.ndarray`, shape (*ndim*,)The source position, a *ndim*-dimensional vector**DivergentBeamGeometry.det_refpoint**`DivergentBeamGeometry.det_refpoint (mpar)`

The detector reference point function.

Parameters`mpar` : element of motion parameters

Motion parameter for which to calculate the detector reference point

Returns`point` : `numpy.ndarray`, shape (*ndim*,)

The reference point, an *ndim*-dimensional vector

DivergentBeamGeometry.det_to_src

`DivergentBeamGeometry.det_to_src` (*mpar*, *dpar*, *normalized=True*)

Vector pointing from a detector location to the source.

A function of the motion and detector parameters.

The default implementation uses the `det_point_position` and `src_position` functions. Implementations can override this, for example if no source position is given.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

normalized : `bool`, optional

If `True`, return a normalized (unit) vector.

Returns`vec` : `numpy.ndarray`, shape (*ndim*,)

(Unit) vector pointing from the detector to the source

DivergentBeamGeometry.rotation_matrix

`DivergentBeamGeometry.rotation_matrix` (*mpar*)

The detector rotation function for calculating the detector reference position.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter for which to calculate the detector reference rotation

Returns`rot` : `numpy.ndarray`, shape (*ndim*, *ndim*)

The rotation matrix mapping the standard basis vectors in the fixed (“lab”) coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

DivergentBeamGeometry.src_position

`DivergentBeamGeometry.src_position` (*mpar*)

The source position function.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter for which to calculate the source position

Returns`pos` : `numpy.ndarray`, shape (*ndim*,)

The source position, a *ndim*-dimensional vector

__init__ (*ndim*, *motion_part*, *detector*)

Initialize a new instance.

Parameters`ndim` : positive `int`

Number of dimensions of this geometry, i.e. dimensionality of the physical space in which this geometry is embedded

motion_part : *RectPartition*

Partition for the set of “motion” parameters

detector : *Detector*

The detector of this geometry

Geometry

class `odl.tomo.geometry.geometry.Geometry(ndim, motion_part, detector)`

Bases: `object`

Abstract geometry class.

A geometry is described by

- a detector,
- a set of detector motion parameters,
- a function mapping motion parameters to the location of a reference point (e.g. the center of the detector surface),
- a rotation applied to the detector surface, depending on the motion parameters,
- a mapping from the motion and surface parameters to the detector pixel direction to the source,
- optionally a mapping from the motion parameters to the source position

Attributes

<i>det_grid</i>	Sampling grid of <i>det_params</i> .
<i>det_params</i>	Continuous detector parameter range, an <i>IntervalProd</i> .
<i>det_partition</i>	Partition of the detector parameter set into subsets.
<i>detector</i>	Detector representation of this geometry.
<i>grid</i>	Joined sampling grid for motion and detector.
<i>implementation_cache</i>	Dictionary acting as a cache for this geometry.
<i>motion_grid</i>	Sampling grid of <i>motion_params</i> .
<i>motion_params</i>	Continuous motion parameter range, an <i>IntervalProd</i> .
<i>motion_partition</i>	Partition of the motion parameter set into subsets.
<i>ndim</i>	The number of dimensions of the geometry.
<i>params</i>	Joined parameter set for motion and detector.
<i>partition</i>	Joined parameter set partition for motion and detector.

Geometry.det_grid

`Geometry.det_grid`

Sampling grid of *det_params*.

Geometry.det_params

`Geometry.det_params`

Continuous detector parameter range, an *IntervalProd*.

Geometry.det_partition

Geometry.det_partition

Partition of the detector parameter set into subsets.

Geometry.detector**Geometry.detector**

Detector representation of this geometry.

Geometry.grid**Geometry.grid**

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

Geometry.implementation_cache**Geometry.implementation_cache**

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns `implementations` : dict

Geometry.motion_grid**Geometry.motion_grid**

Sampling grid of *motion_params*.

Geometry.motion_params**Geometry.motion_params**

Continuous motion parameter range, an *IntervalProd*.

Geometry.motion_partition**Geometry.motion_partition**

Partition of the motion parameter set into subsets.

Geometry.ndim**Geometry.ndim**

The number of dimensions of the geometry.

Geometry.params**Geometry.params**

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

Geometry.partition**Geometry.partition**

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.

Methods

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(mpar)</code>	The detector reference point function.
<code>det_to_src(mpar, dpar[, normalized])</code>	Vector pointing from a detector location to the source.
<code>rotation_matrix(mpar)</code>	The detector rotation function for calculating the detector reference position.

Geometry.det_point_position

`Geometry.det_point_position(mpar, dpar)`

The detector point position function.

Parameters**mpar** : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns**pos** : `numpy.ndarray`, shape (*ndim*,)

The source position, a *ndim*-dimensional vector

Geometry.det_refpoint

`Geometry.det_refpoint(mpar)`

The detector reference point function.

Parameters**mpar** : element of motion parameters

Motion parameter for which to calculate the detector reference point

Returns**point** : `numpy.ndarray`, shape (*ndim*,)

The reference point, an *ndim*-dimensional vector

Geometry.det_to_src

`Geometry.det_to_src(mpar, dpar, normalized=True)`

Vector pointing from a detector location to the source.

A function of the motion and detector parameters.

Parameters**mpar** : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

normalized : `bool`, optional

If `True`, return a normalized (unit) vector. Default: `True`

Returns**vec** : `numpy.ndarray`, shape (*ndim*,)

(Unit) vector pointing from the detector to the source

Geometry.rotation_matrix

Geometry.**rotation_matrix**(*mpar*)

The detector rotation function for calculating the detector reference position.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter for which to calculate the detector reference rotation

Returns*rot* : `numpy.ndarray`, shape (*ndim*, *ndim*)

The rotation matrix mapping the standard basis vectors in the fixed (“lab”) coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

__init__(*ndim*, *motion_part*, *detector*)

Initialize a new instance.

Parameters*ndim* : positive `int`

Number of dimensions of this geometry, i.e. dimensionality of the physical space in which this geometry is embedded

motion_part : *RectPartition*

Partition for the set of “motion” parameters

detector : *Detector*

The detector of this geometry

parallel

Parallel beam geometries.

Classes

<i>Parallel2dGeometry</i> (<i>apart</i> , <i>dpart</i> , <i>**kwargs</i>)	Parallel beam geometry in 2d.
<i>Parallel3dAxisGeometry</i> (<i>apart</i> , <i>dpart</i> [, <i>axis</i>])	Parallel beam geometry in 3d with single rotation axis.
<i>Parallel3dGeometry</i> (<i>apart</i> , <i>dpart</i> , <i>**kwargs</i>)	Parallel beam geometry in 3d.
<i>ParallelGeometry</i> (<i>ndim</i> , <i>apart</i> , <i>detector</i> , ...)	Abstract parallel beam geometry in 2 or 3 dimensions.

Parallel2dGeometry

class `odl.tomo.geometry.parallel.Parallel2dGeometry`(*apart*, *dpart*, ***kwargs*)

Bases: `odl.tomo.geometry.parallel.ParallelGeometry`

Parallel beam geometry in 2d.

The motion parameter is the counter-clockwise rotation angle around the origin, and the detector is a line detector perpendicular to the ray direction.

In the standard configuration, the detector reference point starts at (1, 0), and the initial detector axis is (0, 1).

Attributes

<i>det_grid</i>	Sampling grid of <i>det_params</i> .
Continued on next page	

Table 2.262 – continued from previous page

<i>det_params</i>	Continuous detector parameter range, an <i>IntervalProd</i> .
<i>det_partition</i>	Partition of the detector parameter set into subsets.
<i>detector</i>	Detector representation of this geometry.
<i>grid</i>	Joined sampling grid for motion and detector.
<i>implementation_cache</i>	Dictionary acting as a cache for this geometry.
<i>motion_grid</i>	Sampling grid of <i>motion_params</i> .
<i>motion_params</i>	Continuous motion parameter range, an <i>IntervalProd</i> .
<i>motion_partition</i>	Partition of the motion parameter set into subsets.
<i>ndim</i>	The number of dimensions of the geometry.
<i>params</i>	Joined parameter set for motion and detector.
<i>partition</i>	Joined parameter set partition for motion and detector.

Parallel2dGeometry.det_grid**Parallel2dGeometry.det_grid**Sampling grid of *det_params*.**Parallel2dGeometry.det_params****Parallel2dGeometry.det_params**Continuous detector parameter range, an *IntervalProd*.**Parallel2dGeometry.det_partition****Parallel2dGeometry.det_partition**

Partition of the detector parameter set into subsets.

Parallel2dGeometry.detector**Parallel2dGeometry.detector**

Detector representation of this geometry.

Parallel2dGeometry.grid**Parallel2dGeometry.grid**

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

Parallel2dGeometry.implementation_cache**Parallel2dGeometry.implementation_cache**

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns`implementations`: dict**Parallel2dGeometry.motion_grid****Parallel2dGeometry.motion_grid**Sampling grid of *motion_params*.

Parallel2dGeometry.motion_params`Parallel2dGeometry.motion_params`Continuous motion parameter range, an *IntervalProd*.**Parallel2dGeometry.motion_partition**`Parallel2dGeometry.motion_partition`

Partition of the motion parameter set into subsets.

Parallel2dGeometry.ndim`Parallel2dGeometry.ndim`

The number of dimensions of the geometry.

Parallel2dGeometry.params`Parallel2dGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

Parallel2dGeometry.partition`Parallel2dGeometry.partition`

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**Methods**

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angles)</code>	Return the position of the detector ref.
<code>det_to_src(angles, dpar[, normalized])</code>	Direction from a detector location to the source.
<code>rotation_matrix(angle)</code>	Return the rotation matrix for angle.

Parallel2dGeometry.det_point_position`Parallel2dGeometry.det_point_position(mpar, dpar)`

The detector point position function.

Parameters`mpar` : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns`pos` : `numpy.ndarray`, shape (*ndim*,)The source position, a *ndim*-dimensional vector**Parallel2dGeometry.det_refpoint**`Parallel2dGeometry.det_refpoint(angles)`Return the position of the detector ref. point at *angles*.The reference point is given by a rotation of the initial position by *angles*.

Parameters**angles** : float

Parameters describing the detector rotation, must be contained in *motion_params*.

Returns**point** : `numpy.ndarray`, shape (*ndim*,)

The reference point for the given parameters

Parallel2dGeometry.det_to_src

`Parallel2dGeometry.det_to_src` (*angles*, *dpar*, *normalized=True*)

Direction from a detector location to the source.

In parallel geometry, this function is independent of the detector parameter.

Since the (virtual) source is infinitely far away, only the normalized version is valid.

Parameters**angles** : *array-like*

Euler angles given in radians, must be contained in this geometry's *motion_params*

dpar : float

Detector parameters, must be contained in this geometry's *det_params*

normalized : bool, optional

If `True`, return the normalized version of the vector. For parallel geometry, this is the only sensible option.

Returns**vec** : `numpy.ndarray`, shape (*ndim*,)

Unit vector pointing from the detector to the source

Raises**NotImplementedError**

if `normalized=False` is given, since this case is not well defined.

Parallel2dGeometry.rotation_matrix

`Parallel2dGeometry.rotation_matrix` (*angle*)

Return the rotation matrix for angle.

For an angle *phi*, the matrix is given by:

<pre>rot(phi) = [[cos(phi), -sin(phi)], [sin(phi), cos(phi)]]</pre>
--

Parameters**angle** : float

Rotation angle given in radians, must be contained in this geometry's *motion_params*

Returns**rot** : `numpy.ndarray`, shape (2, 2)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system

__init__ (*apart*, *dpart*, ***kwargs*)

Initialize a new instance.

Parameters**apart** : 1-dim. *RectPartition*

Partition of the angle interval

dpart : 1-dim. *RectPartition*

Partition of the detector parameter interval

det_init_pos : *array-like*, shape (2,), optional

Initial position of the detector reference point. The zero vector is only allowed if `det_init_axis` is explicitly given. Default: (1, 0).

det_init_axis : *array-like* (shape (2,)), optional

Initial axis defining the detector orientation. By default, a normalized *perpendicular_vector* to `det_init_pos` is used, which is only valid if `det_init_axis` is not zero.

Parallel3dAxisGeometry

class odl.tomo.geometry.parallel.**Parallel3dAxisGeometry** (*apart*, *dpart*, *axis*=[0, 0, 1], ***kwargs*)

Bases: *odl.tomo.geometry.parallel.ParallelGeometry*, *odl.tomo.geometry.geometry.AxisOrientedGeometry*

Parallel beam geometry in 3d with single rotation axis.

The motion parameter is the rotation angle around the specified axis, and the detector is a flat 2d detector perpendicular to the ray direction.

In the standard configuration, the rotation axis is (0, 0, 1), the detector reference point starts at (1, 0, 0), and the initial detector axes are [(0, 1, 0), (0, 0, 1)].

Attributes

<i>axis</i>	The normalized axis of rotation, a 3-element vector.
<i>det_grid</i>	Sampling grid of <i>det_params</i> .
<i>det_params</i>	Continuous detector parameter range, an <i>IntervalProd</i> .
<i>det_partition</i>	Partition of the detector parameter set into subsets.
<i>detector</i>	Detector representation of this geometry.
<i>grid</i>	Joined sampling grid for motion and detector.
<i>implementation_cache</i>	Dictionary acting as a cache for this geometry.
<i>motion_grid</i>	Sampling grid of <i>motion_params</i> .
<i>motion_params</i>	Continuous motion parameter range, an <i>IntervalProd</i> .
<i>motion_partition</i>	Partition of the motion parameter set into subsets.
<i>ndim</i>	The number of dimensions of the geometry.
<i>params</i>	Joined parameter set for motion and detector.
<i>partition</i>	Joined parameter set partition for motion and detector.

Parallel3dAxisGeometry.axis

Parallel3dAxisGeometry.axis

The normalized axis of rotation, a 3-element vector.

Parallel3dAxisGeometry.det_grid

Parallel3dAxisGeometry.det_grid

Sampling grid of *det_params*.

Parallel3dAxisGeometry.det_params

`Parallel3dAxisGeometry.det_params`

Continuous detector parameter range, an *IntervalProd*.

`Parallel3dAxisGeometry.det_partition`

`Parallel3dAxisGeometry.det_partition`

Partition of the detector parameter set into subsets.

`Parallel3dAxisGeometry.detector`

`Parallel3dAxisGeometry.detector`

Detector representation of this geometry.

`Parallel3dAxisGeometry.grid`

`Parallel3dAxisGeometry.grid`

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

`Parallel3dAxisGeometry.implementation_cache`

`Parallel3dAxisGeometry.implementation_cache`

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns`implementations` : dict

`Parallel3dAxisGeometry.motion_grid`

`Parallel3dAxisGeometry.motion_grid`

Sampling grid of *motion_params*.

`Parallel3dAxisGeometry.motion_params`

`Parallel3dAxisGeometry.motion_params`

Continuous motion parameter range, an *IntervalProd*.

`Parallel3dAxisGeometry.motion_partition`

`Parallel3dAxisGeometry.motion_partition`

Partition of the motion parameter set into subsets.

`Parallel3dAxisGeometry.ndim`

`Parallel3dAxisGeometry.ndim`

The number of dimensions of the geometry.

`Parallel3dAxisGeometry.params`

`Parallel3dAxisGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

Parallel3dAxisGeometry.partitionParallel3dAxisGeometry.**partition**

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**Methods**

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angles)</code>	Return the position of the detector ref.
<code>det_to_src(angles, dpar[, normalized])</code>	Direction from a detector location to the source.
<code>rotation_matrix(angle)</code>	The detector rotation function.

Parallel3dAxisGeometry.det_point_positionParallel3dAxisGeometry.**det_point_position** (*mpar*, *dpar*)

The detector point position function.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns*pos* : numpy.ndarray, shape (*ndim*,)The source position, a *ndim*-dimensional vector**Parallel3dAxisGeometry.det_refpoint**Parallel3dAxisGeometry.**det_refpoint** (*angles*)Return the position of the detector ref. point at *angles*.The reference point is given by a rotation of the initial position by *angles*.**Parameters***angles* : floatParameters describing the detector rotation, must be contained in *motion_params*.**Returns***point* : numpy.ndarray, shape (*ndim*,)

The reference point for the given parameters

Parallel3dAxisGeometry.det_to_srcParallel3dAxisGeometry.**det_to_src** (*angles*, *dpar*, *normalized=True*)

Direction from a detector location to the source.

In parallel geometry, this function is independent of the detector parameter.

Since the (virtual) source is infinitely far away, only the normalized version is valid.

Parameters*angles* : *array-like*Euler angles given in radians, must be contained in this geometry's *motion_params***dpar** : floatDetector parameters, must be contained in this geometry's *det_params*

normalized : bool, optional

If True, return the normalized version of the vector. For parallel geometry, this is the only sensible option.

Returnsvec : numpy.ndarray, shape (ndim,)

Unit vector pointing from the detector to the source

RaisesNotImplementedError

if normalized=False is given, since this case is not well defined.

Parallel3dAxisGeometry.rotation_matrix

Parallel3dAxisGeometry.rotation_matrix (angle)

The detector rotation function.

Returns the matrix for rotating a vector in 3d by an angle angle about the rotation axis given by the property axis according to the right hand rule.

The matrix is computed according to Rodrigues' rotation formula.

Parametersangle : float

The motion parameter given in radian. It must be contained in this geometry's motion_params.

Returnsrot_mat : numpy.ndarray, shape (3, 3)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

__init__ (apart, dpart, axis=[0, 0, 1], **kwargs)

Initialize a new instance.

Parametersapart : 1-dim. RectPartition

Partition of the angle interval

dpart : 2-dim. RectPartition

Partition of the detector parameter interval

axis : array-like, shape (3,), optional

Fixed rotation axis defined by a 3-element vector

det_init_pos : array-like, shape (3,), optional

Initial position of the detector reference point. The zero vector is only allowed if det_init_axes is explicitly given. By default, a perpendicular_vector to axis is used.

det_init_axes : 2-tuple of array-like (shape (3,)), optional

Initial axes defining the detector orientation. By default, the normalized cross product of axis and det_init_pos is used as first axis and axis as second.

Parallel3dGeometry

class odl.tomo.geometry.parallel.Parallel3dGeometry (apart, dpart, **kwargs)

Bases: odl.tomo.geometry.parallel.ParallelGeometry

Parallel beam geometry in 3d.

The motion parameters are two or three Euler angles, and the detector is flat and two-dimensional.

In the standard configuration, the detector reference point starts at $(1, 0, 0)$, and the initial detector axes are $[(0, 1, 0), (0, 0, 1)]$.

Attributes

<code>det_grid</code>	Sampling grid of <code>det_params</code> .
<code>det_params</code>	Continuous detector parameter range, an <i>IntervalProd</i> .
<code>det_partition</code>	Partition of the detector parameter set into subsets.
<code>detector</code>	Detector representation of this geometry.
<code>grid</code>	Joined sampling grid for motion and detector.
<code>implementation_cache</code>	Dictionary acting as a cache for this geometry.
<code>motion_grid</code>	Sampling grid of <code>motion_params</code> .
<code>motion_params</code>	Continuous motion parameter range, an <i>IntervalProd</i> .
<code>motion_partition</code>	Partition of the motion parameter set into subsets.
<code>ndim</code>	The number of dimensions of the geometry.
<code>params</code>	Joined parameter set for motion and detector.
<code>partition</code>	Joined parameter set partition for motion and detector.

Parallel3dGeometry.det_grid

Parallel3dGeometry.**det_grid**

Sampling grid of `det_params`.

Parallel3dGeometry.det_params

Parallel3dGeometry.**det_params**

Continuous detector parameter range, an *IntervalProd*.

Parallel3dGeometry.det_partition

Parallel3dGeometry.**det_partition**

Partition of the detector parameter set into subsets.

Parallel3dGeometry.detector

Parallel3dGeometry.**detector**

Detector representation of this geometry.

Parallel3dGeometry.grid

Parallel3dGeometry.**grid**

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

Parallel3dGeometry.implementation_cache

Parallel3dGeometry.**implementation_cache**

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns `implementations`: dict

Parallel3dGeometry.motion_grid**Parallel3dGeometry.motion_grid**Sampling grid of *motion_params*.**Parallel3dGeometry.motion_params****Parallel3dGeometry.motion_params**Continuous motion parameter range, an *IntervalProd*.**Parallel3dGeometry.motion_partition****Parallel3dGeometry.motion_partition**

Partition of the motion parameter set into subsets.

Parallel3dGeometry.ndim**Parallel3dGeometry.ndim**

The number of dimensions of the geometry.

Parallel3dGeometry.params**Parallel3dGeometry.params**

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

Parallel3dGeometry.partition**Parallel3dGeometry.partition**

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**Methods**

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angles)</code>	Return the position of the detector ref.
<code>det_to_src(angles, dpar[, normalized])</code>	Direction from a detector location to the source.
<code>rotation_matrix(angles)</code>	Matrix defining the detector rotation at angles.

Parallel3dGeometry.det_point_position**Parallel3dGeometry.det_point_position** (*mpar*, *dpar*)

The detector point position function.

Parameters**mpar** : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns**pos** : `numpy.ndarray`, shape (*ndim*,)The source position, a *ndim*-dimensional vector

Parallel3dGeometry.det_refpoint`Parallel3dGeometry.det_refpoint` (*angles*)

Return the position of the detector ref. point at angles.

The reference point is given by a rotation of the initial position by angles.

Parameters*angles* : float

Parameters describing the detector rotation, must be contained in *motion_params*.

Returns*point* : `numpy.ndarray`, shape (*ndim*,)

The reference point for the given parameters

Parallel3dGeometry.det_to_src`Parallel3dGeometry.det_to_src` (*angles*, *dpar*, *normalized=True*)

Direction from a detector location to the source.

In parallel geometry, this function is independent of the detector parameter.

Since the (virtual) source is infinitely far away, only the normalized version is valid.

Parameters*angles* : *array-like*

Euler angles given in radians, must be contained in this geometry's *motion_params*

dpar : float

Detector parameters, must be contained in this geometry's *det_params*

normalized : bool, optional

If True, return the normalized version of the vector. For parallel geometry, this is the only sensible option.

Returns*vec* : `numpy.ndarray`, shape (*ndim*,)

Unit vector pointing from the detector to the source

Raises`NotImplementedError`

if *normalized=False* is given, since this case is not well defined.

Parallel3dGeometry.rotation_matrix`Parallel3dGeometry.rotation_matrix` (*angles*)

Matrix defining the detector rotation at angles.

Parameters*angles* : *array-like*

Angles in radians defining the rotation, must be contained in this geometry's *motion_params*

Returns*rot* : `numpy.ndarray`, shape (3, 3)

The rotation matrix mapping the standard basis vectors in the fixed ("lab") coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

`__init__` (*apart*, *dpart*, ***kwargs*)

Initialize a new instance.

Parameters*apart* : 2- or 3-dim. *RectPartition*

Partition of the angle parameter set

dpart : 2-dim. *RectPartition*

Partition of the detector parameter interval

det_init_pos : *array-like*, shape (3,), optional

Initial position of the detector reference point. The zero vector is only allowed if `det_init_axes` is explicitly given. Default: (1, 0, 0)

det_init_axes : 2-tuple of *array-like* (shape (3,)), optional

Initial axes defining the detector orientation. By default, a normalized *perpendicular_vector* to `det_init_pos` is taken as first axis, and the normalized cross product of these two as second.

ParallelGeometry

class `odl.tomo.geometry.parallel.ParallelGeometry` (*ndim, apart, detector, det_init_pos*)

Bases: `odl.tomo.geometry.geometry.Geometry`

Abstract parallel beam geometry in 2 or 3 dimensions.

Parallel geometries are characterized by a virtual source at infinity, such that a unit vector from a detector point towards the source (*det_to_src*) is independent of the location on the detector.

Attributes

<i>det_grid</i>	Sampling grid of <i>det_params</i> .
<i>det_params</i>	Continuous detector parameter range, an <i>IntervalProd</i> .
<i>det_partition</i>	Partition of the detector parameter set into subsets.
<i>detector</i>	Detector representation of this geometry.
<i>grid</i>	Joined sampling grid for motion and detector.
<i>implementation_cache</i>	Dictionary acting as a cache for this geometry.
<i>motion_grid</i>	Sampling grid of <i>motion_params</i> .
<i>motion_params</i>	Continuous motion parameter range, an <i>IntervalProd</i> .
<i>motion_partition</i>	Partition of the motion parameter set into subsets.
<i>ndim</i>	The number of dimensions of the geometry.
<i>params</i>	Joined parameter set for motion and detector.
<i>partition</i>	Joined parameter set partition for motion and detector.

ParallelGeometry.det_grid

`ParallelGeometry.det_grid`

Sampling grid of *det_params*.

ParallelGeometry.det_params

`ParallelGeometry.det_params`

Continuous detector parameter range, an *IntervalProd*.

ParallelGeometry.det_partition

`ParallelGeometry.det_partition`

Partition of the detector parameter set into subsets.

ParallelGeometry.detector

`ParallelGeometry.detector`

Detector representation of this geometry.

ParallelGeometry.grid`ParallelGeometry.grid`

Joined sampling grid for motion and detector.

By convention, the motion grid comes before the detector grid.

ParallelGeometry.implementation_cache`ParallelGeometry.implementation_cache`

Dictionary acting as a cache for this geometry.

Intended for reuse of computations. Implementations that use this storage should take care of unique naming.

Returns`implementations`: dict**ParallelGeometry.motion_grid**`ParallelGeometry.motion_grid`Sampling grid of *motion_params*.**ParallelGeometry.motion_params**`ParallelGeometry.motion_params`Continuous motion parameter range, an *IntervalProd*.**ParallelGeometry.motion_partition**`ParallelGeometry.motion_partition`

Partition of the motion parameter set into subsets.

ParallelGeometry.ndim`ParallelGeometry.ndim`

The number of dimensions of the geometry.

ParallelGeometry.params`ParallelGeometry.params`

Joined parameter set for motion and detector.

By convention, the motion parameters come before the detector parameters.

ParallelGeometry.partition`ParallelGeometry.partition`

Joined parameter set partition for motion and detector.

Returns a *RectPartition* with the detector partition inserted after the motion partition.**Methods**

<code>__eq__</code>	Return self==value.
<code>det_point_position(mpar, dpar)</code>	The detector point position function.
<code>det_refpoint(angles)</code>	Return the position of the detector ref.
<code>det_to_src(angles, dpar[, normalized])</code>	Direction from a detector location to the source.
<code>rotation_matrix(mpar)</code>	The detector rotation function for calculating the detector reference position.

ParallelGeometry.det_point_position

`ParallelGeometry.det_point_position` (*mpar*, *dpar*)

The detector point position function.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter at which to evaluate

dpar : element of detector parameters *det_params*

Detector parameter at which to evaluate

Returns*pos* : `numpy.ndarray`, shape (*ndim*,)

The source position, a *ndim*-dimensional vector

ParallelGeometry.det_refpoint

`ParallelGeometry.det_refpoint` (*angles*)

Return the position of the detector ref. point at angles.

The reference point is given by a rotation of the initial position by angles.

Parameters*angles* : float

Parameters describing the detector rotation, must be contained in *motion_params*.

Returns*point* : `numpy.ndarray`, shape (*ndim*,)

The reference point for the given parameters

ParallelGeometry.det_to_src

`ParallelGeometry.det_to_src` (*angles*, *dpar*, *normalized=True*)

Direction from a detector location to the source.

In parallel geometry, this function is independent of the detector parameter.

Since the (virtual) source is infinitely far away, only the normalized version is valid.

Parameters*angles* : *array-like*

Euler angles given in radians, must be contained in this geometry's *motion_params*

dpar : float

Detector parameters, must be contained in this geometry's *det_params*

normalized : bool, optional

If `True`, return the normalized version of the vector. For parallel geometry, this is the only sensible option.

Returns*vec* : `numpy.ndarray`, shape (*ndim*,)

Unit vector pointing from the detector to the source

Raises`NotImplementedError`

if `normalized=False` is given, since this case is not well defined.

ParallelGeometry.rotation_matrix

ParallelGeometry.**rotation_matrix**(*mpar*)

The detector rotation function for calculating the detector reference position.

Parameters*mpar* : element of motion parameters *motion_params*

Motion parameter for which to calculate the detector reference rotation

Returns*rot* : `numpy.ndarray`, shape (*ndim*, *ndim*)

The rotation matrix mapping the standard basis vectors in the fixed (“lab”) coordinate system to the basis vectors of the local coordinate system of the detector reference point, expressed in the fixed system.

___**init**___(*ndim*, *apart*, *detector*, *det_init_pos*)

Initialize a new instance.

Parameters*ndim* : {2, 3}

Number of dimensions of this geometry, i.e. dimensionality of the physical space in which this geometry is embedded

apart : *RectPartition*

Partition of the angle set

detector : *Detector*

The detector to use in this geometry

det_init_pos : *array-like*

Initial position of the detector reference point. The zero vector is not allowed.

operators**Modules****ray_trafo**

Ray transforms.

Classes

<i>RayBackProjection</i> (<i>discr_range</i> , <i>geometry</i> [, <i>impl</i>])	The adjoint of the discrete Ray transform between L^p spaces.
<i>RayTransform</i> (<i>discr_domain</i> , <i>geometry</i> [, <i>impl</i>])	The discrete Ray transform between L^p spaces.

RayBackProjection

class odl.tomo.operators.ray_trafo.**RayBackProjection**(*discr_range*, *geometry*,
impl=‘astra_cpu’, ***kwargs*)

Bases: *odl.operator.operator.Operator*

The adjoint of the discrete Ray transform between L^p spaces.

Attributes

<code>adjoint</code>	Return the adjoint operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>geometry</code>	Geometry of this operator.
<code>impl</code>	Implementation back-end for evaluation of this operator.
<code>inverse</code>	Return the operator inverse.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

RayBackProjection.adjoint`RayBackProjection.adjoint`

Return the adjoint operator.

RayBackProjection.domain`RayBackProjection.domain`

Set of objects on which this operator can be evaluated.

RayBackProjection.geometry`RayBackProjection.geometry`

Geometry of this operator.

RayBackProjection.impl`RayBackProjection.impl`

Implementation back-end for evaluation of this operator.

RayBackProjection.inverse`RayBackProjection.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

RayBackProjection.is_functional`RayBackProjection.is_functional`True if the this operator's range is a *Field*.**RayBackProjection.is_linear**`RayBackProjection.is_linear`

True if this operator is linear.

RayBackProjection.range`RayBackProjection.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x[, out])</code>	Apply the operator to <code>x</code> and store the result in <code>out</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

RayBackProjection.__call__

`RayBackProjection.__call__(x, out=None, **kwargs)`

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

RayBackProjection._call`RayBackProjection._call(x, out=None)`

Apply the operator to `x` and store the result in `out`.

Parameters`x` : *DiscreteLpVector*

Element in the domain of the operator which is back-projected

out : *DiscreteLpVector*, optional

Element in the reconstruction space to which the result is written. If `None` an element in the range of the operator is created.

Returns`out` : *DiscreteLpVector*

Returns an element in the projection space

RayBackProjection.derivative`RayBackProjection.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__(discr_range, geometry, impl='astra_cpu', **kwargs)`

Initialize a new instance.

Parameters`discr_range` : *DiscreteLp*

Reconstruction space, the range of the back-projector

geometry : *Geometry*

The geometry of the transform, contains information about the operator domain

impl : {'astra_cpu', 'astra_cuda'}, optional

Implementation back-end for the transform. Supported back-ends: 'astra_cpu': ASTRA toolbox using CPU, only 2D 'astra_cuda': ASTRA toolbox, using CUDA, 2D or 3D

interp : {'nearest', 'linear'}

Interpolation type for the discretization of the operator range. Default: 'nearest'

RayTransform`class odl.tomo.operators.ray_trafo.RayTransform(discr_domain, geometry, impl='astra_cpu', **kwargs)`

Bases: *odl.operator.operator.Operator*

The discrete Ray transform between L^p spaces.

Attributes

<i>adjoint</i>	Return the adjoint operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>geometry</i>	Geometry of this operator.
<i>impl</i>	Implementation back-end for evaluation of this operator.
<i>inverse</i>	Return the operator inverse.
Continued on next page	

Table 2.273 – continued from previous page

<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.

RayTransform.adjoint`RayTransform.adjoint`

Return the adjoint operator.

RayTransform.domain`RayTransform.domain`

Set of objects on which this operator can be evaluated.

RayTransform.geometry`RayTransform.geometry`

Geometry of this operator.

RayTransform.impl`RayTransform.impl`

Implementation back-end for evaluation of this operator.

RayTransform.inverse`RayTransform.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

RayTransform.is_functional`RayTransform.is_functional`True if the this operator's range is a *Field*.**RayTransform.is_linear**`RayTransform.is_linear`

True if this operator is linear.

RayTransform.range`RayTransform.range`

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>__call__(x[, out])</code>	Apply the operator to <code>x</code> and store the result in <code>out</code> .
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

RayTransform.__call__

RayTransform.__call__(x, out=None, **kwargs)

Return self(x[, out, **kwargs]).

Implementation of the call pattern `op(x)` with the private `__call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `__call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

RayTransform._call

RayTransform.__call__(x, out=None)

Apply the operator to `x` and store the result in `out`.

Parameters`x` : *DiscreteLpVector*

Element in the domain of the operator to be forward projected

out : *DiscreteLpVector*, optional

Vector in the projection space to which the result is written. If `None` creates an element in the range of the operator.

Returns`out` : *DiscreteLpVector*

Returns an element in the projection space

RayTransform.derivative

`RayTransform.derivative` (*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__` (*discr_domain*, *geometry*, *impl*='astra_cpu', ***kwargs*)

Initialize a new instance.

Parameters`discr_domain` : *DiscreteLp*

Discretized space, the domain of the forward projector

geometry : *Geometry*

Geometry of the transform, containing information about the operator range

impl : {'astra_cpu', 'astra_cuda'}, optional

Implementation back-end for the transform. Supported back-ends: 'astra_cpu': ASTRA toolbox using CPU, only 2D 'astra_cuda': ASTRA toolbox, using CUDA, 2D or 3D

interp : {'nearest', 'linear'}

Interpolation type for the discretization of the operator range. Default: 'nearest'

util

Modules

utility

Functions

<code>angles_from_matrix</code> (<i>rot_matrix</i>)	
<code>axis_rotation</code> (<i>axis</i> , <i>angle</i> , <i>vectors</i>)	Rotate a vector or an array of vectors around an axis in 3d.
<code>axis_rotation_matrix</code> (<i>axis</i> , <i>angle</i>)	Matrix of the rotation around an axis in 3d.
<code>euler_matrix</code> (<i>*angles</i>)	Rotation matrix in 2 and 3 dimensions.
<code>is_rotation_matrix</code> (<i>mat</i> [, <i>show_diff</i>])	
<code>perpendicular_vector</code> (<i>vec</i>)	Return a vector perpendicular to <i>vec</i> .
<code>to_lab_sys</code> (<i>vec_in_local_coords</i> , <i>local_sys</i>)	
<code>to_local_sys</code> (<i>vec_in_lab_coords</i> , <i>local_sys</i>)	

angles_from_matrix

`odl.tomo.util.utility.angles_from_matrix` (*rot_matrix*)

axis_rotation

`odl.tomo.util.utility.axis_rotation(axis, angle, vectors)`

Rotate a vector or an array of vectors around an axis in 3d.

The rotation is computed by *Rodriguez' rotation formula*.

Parameters`axis` : array-like, shape (3,)

The rotation axis, assumed to be a unit vector

angle : float

The rotation angle

vectors : array-like, shape (3,) or (N, 3)

The vector(s) to be rotated

Returns`rot_vec` : `numpy.ndarray`

The rotated vector(s)

https://en.wikipedia.org/wiki/Rodrigues'_rotation_formula

axis_rotation_matrix

`odl.tomo.util.utility.axis_rotation_matrix(axis, angle)`

Matrix of the rotation around an axis in 3d.

The matrix is computed according to *Rodriguez' rotation formula*.

Parameters`axis` : array-like, shape (3,)

The rotation axis, assumed to be a unit vector

angle : float

The rotation angle

Returns`mat` : `numpy.ndarray`, shape (3, 3)

The axis rotation matrix

https://en.wikipedia.org/wiki/Rodrigues'_rotation_formula

euler_matrix

`odl.tomo.util.utility.euler_matrix(*angles)`

Rotation matrix in 2 and 3 dimensions.

Compute the Euler rotation matrix from angles given in radians. Its rows represent the canonical unit vectors as seen from the rotated system while the columns are the rotated unit vectors as seen from the canonical system.

Parameters`angle1,...,angleN` : float

One angle results in a (2x2) matrix representing a counter-clockwise rotation. Two or three angles result in a (3x3) matrix and are interpreted as Euler angles of a 3d rotation according to the 'ZXZ' rotation order, see the Wikipedia article *Euler angles*.

Returns`mat` : `numpy.ndarray`, shape (2, 2) or (3, 3)

The rotation matrix

https://en.wikipedia.org/wiki/Euler_angles#Rotation_matrix

is_rotation_matrix

`odl.tomo.util.utility.is_rotation_matrix(mat, show_diff=False)`

perpendicular_vector

`odl.tomo.util.utility.perpendicular_vector(vec)`

Return a vector perpendicular to `vec`.

Parameters`vec`: *array-like*

Vector of arbitrary length

Returns`perp_vec`: `numpy.ndarray`

Array of same size such that `<vec, perp_vec> == 0`

Examples

Works in 2d:

```
>>> perpendicular_vector([1, 0])
array([ 0.,  1.])
>>> perpendicular_vector([0, 1])
array([-1.,  0.] )
```

And in 3d:

```
>>> perpendicular_vector([1, 0, 0])
array([ 0.,  1.,  0.])
>>> perpendicular_vector([0, 1, 0])
array([-1.,  0.,  0.])
>>> perpendicular_vector([0, 0, 1])
array([ 1.,  0.,  0.] )
```

to_lab_sys

`odl.tomo.util.utility.to_lab_sys(vec_in_local_coords, local_sys)`

to_local_sys

`odl.tomo.util.utility.to_local_sys(vec_in_lab_coords, local_sys)`

2.8.8 trafos

Function transformations based on ODL.

Modules**fourier**

Discretized Fourier transform on L^p spaces.

Classes

<code>DiscreteFourierTransform(dom[, ran, axes, ...])</code>	Plain forward DFT, only evaluating the trigonometric sum.
<code>DiscreteFourierTransformInverse(ran[, dom, ...])</code>	Plain backward DFT, only evaluating the trigonometric sum.
<code>FourierTransform(dom[, ran, impl])</code>	Discretized Fourier transform between discrete L^p spaces.
<code>FourierTransformInverse(ran[, dom, impl])</code>	Inverse of the discretized Fourier transform between L^p spaces.

DiscreteFourierTransform

class `odl.trafos.fourier.DiscreteFourierTransform` (*dom, ran=None, axes=None, sign='-', halfcomplex=False, impl='numpy'*)

Bases: `odl.operator.operator.Operator`

Plain forward DFT, only evaluating the trigonometric sum.

This operator calculates the forward DFT:

```
f_hat[k] = sum_j( f[j] * exp(-+ 1j*2*pi * j*k/N) )
```

without any further shifting or scaling compensation. See the [Numpy FFT documentation](#), the [pyfftw API documentation](#) or [What FFTW really computes](#) for further information.

See also:

`numpy.fft.fftn` n -dimensional FFT routine

`numpy.fft.rfftn` n -dimensional half-complex FFT

`pyfftw.callapply` an FFTW transform

References

Attributes

<code>adjoint</code>	Adjoint transform, equal to the inverse.
<code>axes</code>	Axes along the FT is calculated by this operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>halfcomplex</code>	Return True if the last transform axis is halved.
<code>impl</code>	Backend for the FFT implementation.
<code>inverse</code>	Inverse Fourier transform.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.
<code>sign</code>	Sign of the complex exponent in the transform.

DiscreteFourierTransform.adjoint

`DiscreteFourierTransform.adjoint`

Adjoint transform, equal to the inverse.

DiscreteFourierTransform.axes

`DiscreteFourierTransform.axes`

Axes along the FT is calculated by this operator.

DiscreteFourierTransform.domain

DiscreteFourierTransform.domain
Set of objects on which this operator can be evaluated.

DiscreteFourierTransform.halfcomplex
DiscreteFourierTransform.halfcomplex
Return True if the last transform axis is halved.

DiscreteFourierTransform.impl
DiscreteFourierTransform.impl
Backend for the FFT implementation.

DiscreteFourierTransform.inverse
DiscreteFourierTransform.inverse
Inverse Fourier transform.

DiscreteFourierTransform.is_functional
DiscreteFourierTransform.is_functional
True if the this operator's range is a *Field*.

DiscreteFourierTransform.is_linear
DiscreteFourierTransform.is_linear
True if this operator is linear.

DiscreteFourierTransform.range
DiscreteFourierTransform.range
Set in which the result of an evaluation of this operator lies.

DiscreteFourierTransform.sign
DiscreteFourierTransform.sign
Sign of the complex exponent in the transform.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x, out, **kwargs)</code>	Implement <code>self(x, out[, **kwargs])</code> .
<code>clear_fftw_plan()</code>	Delete the FFTW plan of this transform.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .
<code>init_fftw_plan([planning_effort])</code>	Initialize the FFTW plan for this transform for later use.

DiscreteFourierTransform.__call__
DiscreteFourierTransform.__call__(`x`, `out=None`, `**kwargs`)
Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain* element-like

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

DiscreteFourierTransform._call

DiscreteFourierTransform._call(*x*, *out*, ***kwargs*)

Implement `self(x, out[, **kwargs])`.

Parameters *x* : domain *element*

Discretized function to be transformed

out : range *element*

Element to which the output is written

See also:

`pyfftw_call` Call pyfftw backend directly

Notes

See the `pyfftw_call` function for `**kwargs` options. The parameters `axes` and `halfcomplex` cannot be overridden.

DiscreteFourierTransform.clear_fftw_plan

`DiscreteFourierTransform.clear_fftw_plan()`

Delete the FFTW plan of this transform.

DiscreteFourierTransform.derivative

`DiscreteFourierTransform.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

DiscreteFourierTransform.init_fftw_plan

`DiscreteFourierTransform.init_fftw_plan(planning_effort='measure', **kwargs)`

Initialize the FFTW plan for this transform for later use.

If the implementation of this operator is not 'pyfftw', this method has no effect.

Parameters`planning_effort` : { 'estimate', 'measure', 'patient', 'exhaustive' }

Flag for the amount of effort put into finding an optimal FFTW plan. See the [FFTW doc on planner flags](#).

planning_timelimit : float, optional

Limit planning time to roughly this amount of seconds. Default: None (no limit)

threads : int, optional

Number of threads to use. Default: 1

__init__ (`dom`, `ran=None`, `axes=None`, `sign='-'`, `halfcomplex=False`, `impl='numpy'`)

Initialize a new instance.

Parameters`dom` : `DiscreteLp`

Domain of the Fourier transform. If its `DiscreteLp.exponent` is equal to 2.0, this operator has an adjoint which is equal to the inverse.

ran : `DiscreteLp`, optional

Range of the Fourier transform. If not given, the range is determined from `dom` and the other parameters as a `discr_sequence_space` with exponent $p / (p - 1)$ (read as 'inf' for $p=1$ and 1 for $p='inf'$).

axes : sequence of int, optional

Dimensions in which a transform is to be calculated. None means all axes.

sign : { '-', '+' }, optional

Sign of the complex exponent. Default: '-'

halfcomplex : bool, optional

If `True`, calculate only the negative frequency part along the last axis in `axes` for real input. This reduces the size of the range to `floor(N[i]/2) + 1` in this axis `i`, where `N` is the shape of the input arrays. Otherwise, calculate the full complex FFT. If `dom_dtype` is a complex type, this option has no effect.

impl : { 'numpy', 'pyfftw' }

Backend for the FFT implementation. The 'pyfftw' backend is faster but requires the `pyfftw` package.

Examples

Complex-to-complex (default) transforms have the same grids in domain and range:

```
>>> domain = discr_sequence_space((2, 4))
>>> fft = DiscreteFourierTransform(domain)
>>> fft.domain.shape
(2, 4)
>>> fft.range.shape
(2, 4)
```

Real-to-complex transforms have a range grid with shape `n // 2 + 1` in the last tranform axis:

```
>>> domain = discr_sequence_space((2, 3, 4), dtype='float')
>>> axes = (0, 1)
>>> fft = DiscreteFourierTransform(
...     domain, halfcomplex=True, axes=axes)
>>> fft.range.shape      # shortened in the second axis
(2, 2, 4)
>>> fft.domain.shape
(2, 3, 4)
```

DiscreteFourierTransformInverse

class `odl.trafos.fourier.DiscreteFourierTransformInverse` (*ran, dom=None, axes=None, sign='+', halfcomplex=False, impl='numpy'*)

Bases: `odl.trafos.fourier.DiscreteFourierTransform`

Plain backward DFT, only evaluating the trigonometric sum.

This operator calculates the inverse DFT:

```
f[k] = 1/prod(N) * sum_j( f_hat[j] * exp(+ 1j*2*pi * j*k/N) )
```

without any further shifting or scaling compensation. See the [Numpy FFT documentation](#), the [pyfftw API documentation](#) or [What FFTW really computes](#) for further information.

See also:

`numpy.fft.ifftn` n-dimensional inverse FFT routine

`numpy.fft.irfftn` n-dimensional half-complex inverse FFT

`pyfftw.call` apply an FFTW transform

References

Attributes

<i>adjoint</i>	Adjoint transform, equal to the inverse.
<i>axes</i>	Axes along the FT is calculated by this operator.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>halfcomplex</i>	Return True if the last transform axis is halved.
<i>impl</i>	Backend for the FFT implementation.
<i>inverse</i>	Inverse Fourier transform.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>range</i>	Set in which the result of an evaluation of this operator lies.
<i>sign</i>	Sign of the complex exponent in the transform.

DiscreteFourierTransformInverse.adjoint

`DiscreteFourierTransformInverse.adjoint`

Adjoint transform, equal to the inverse.

DiscreteFourierTransformInverse.axes

`DiscreteFourierTransformInverse.axes`

Axes along the FT is calculated by this operator.

DiscreteFourierTransformInverse.domain

`DiscreteFourierTransformInverse.domain`

Set of objects on which this operator can be evaluated.

DiscreteFourierTransformInverse.halfcomplex

`DiscreteFourierTransformInverse.halfcomplex`

Return True if the last transform axis is halved.

DiscreteFourierTransformInverse.impl

`DiscreteFourierTransformInverse.impl`

Backend for the FFT implementation.

DiscreteFourierTransformInverse.inverse

`DiscreteFourierTransformInverse.inverse`

Inverse Fourier transform.

DiscreteFourierTransformInverse.is_functional

`DiscreteFourierTransformInverse.is_functional`

True if the this operator's range is a *Field*.

DiscreteFourierTransformInverse.is_linear

`DiscreteFourierTransformInverse.is_linear`

True if this operator is linear.

DiscreteFourierTransformInverse.rangeDiscreteFourierTransformInverse.**range**

Set in which the result of an evaluation of this operator lies.

DiscreteFourierTransformInverse.signDiscreteFourierTransformInverse.**sign**

Sign of the complex exponent in the transform.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x, out, **kwargs)</code>	Implement <code>self(x, out[, **kwargs])</code> .
<code>clear_fftw_plan()</code>	Delete the FFTW plan of this transform.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .
<code>init_fftw_plan([planning_effort])</code>	Initialize the FFTW plan for this transform for later use.

DiscreteFourierTransformInverse.__call__DiscreteFourierTransformInverse.**__call__**(*x*, *out=None*, ***kwargs*)Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters***x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optionalPassed on to the underlying implementation in `_call`**Returns***out* : *Operator.range element*

Result of the operator evaluation. If *out* was provided, the returned object is a reference to it.

See also:`_call` Implementation of the method**Examples**

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

DiscreteFourierTransformInverse._call

DiscreteFourierTransformInverse._call(*x*, *out*, ***kwargs*)

Implement self(*x*, *out*[, ***kwargs*]).

Parameters*xs* : domain *element*

Discretized function to be transformed

out : range *element*

Element to which the output is written

See also:

[*pyfftw_call*](#) Call pyfftw backend directly

Notes

See the [*pyfftw_call*](#) function for ***kwargs* options. The parameters *axes* and *halfcomplex* cannot be overridden.

DiscreteFourierTransformInverse.clear_fftw_plan

DiscreteFourierTransformInverse.clear_fftw_plan()

Delete the FFTW plan of this transform.

DiscreteFourierTransformInverse.derivative

DiscreteFourierTransformInverse.derivative(*point*)

Return the operator derivative at *point*.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

DiscreteFourierTransformInverse.init_fftw_plan

DiscreteFourierTransformInverse.init_fftw_plan(*planning_effort*=*'measure'*, ***kwargs*)

Initialize the FFTW plan for this transform for later use.

If the implementation of this operator is not 'pyfftw', this method has no effect.

Parameters*planning_effort* : {'estimate', 'measure', 'patient', 'exhaustive'}

Flag for the amount of effort put into finding an optimal FFTW plan. See the [FFTW doc on planner flags](#).

planning_timelimit : float, optional

Limit planning time to roughly this amount of seconds. Default: None (no limit)

threads : int, optional

Number of threads to use. Default: 1

__init__(*ran, dom=None, axes=None, sign='+', halfcomplex=False, impl='numpy'*)

Initialize a new instance.

Parameters*ran* : *DiscreteLp*

Range of the inverse Fourier transform. If its *DiscreteLp.exponent* is equal to 2.0, this operator has an adjoint which is equal to the inverse.

dom : *DiscreteLp*, optional

Domain of the inverse Fourier transform. If not given, the domain is determined from *ran* and the other parameters as a *discr_sequence_space* with exponent $p / (p - 1)$ (read as 'inf' for $p=1$ and 1 for $p='inf'$).

axes : sequence of int, optional

Dimensions in which a transform is to be calculated. None means all axes.

sign : {'-', '+'}, optional

Sign of the complex exponent. Default: '-'

halfcomplex : bool, optional

If True, interpret the last axis in *axes* as the negative frequency part of the transform of a real signal and calculate a "half-complex-to-real" inverse FFT. In this case, the domain has by default the shape $\text{floor}(N[i]/2) + 1$ in this axis *i*. Otherwise, domain and range have the same shape. If *ran* is a complex space, this option has no effect.

impl : {'numpy', 'pyfftw'}

Backend for the FFT implementation. The 'pyfftw' backend is faster but requires the pyfftw package.

Examples

Complex-to-complex (default) transforms have the same grids in domain and range:

```
>>> range_ = discr_sequence_space((2, 4))
>>> ifft = DiscreteFourierTransformInverse(range_)
>>> ifft.domain.shape
(2, 4)
>>> ifft.range.shape
(2, 4)
```

Complex-to-real transforms have a domain grid with shape $n // 2 + 1$ in the last tranform axis:

```
>>> range_ = discr_sequence_space((2, 3, 4), dtype='float')
>>> axes = (0, 1)
>>> ifft = DiscreteFourierTransformInverse(
...     range_, halfcomplex=True, axes=axes)
>>> ifft.domain.shape    # shortened in the second axis
(2, 2, 4)
>>> ifft.range.shape
(2, 3, 4)
```


FourierTransform

class odl.trafos.fourier.**FourierTransform**(*dom*, *ran*=None, *impl*='numpy', ***kwargs*)

Bases: `odl.operator.operator.Operator`

Discretized Fourier transform between discrete L^p spaces.

This operator is the discretized variant of the continuous [Fourier Transform](#) between Lebesgue L^p spaces. It applies a three-step procedure consisting of a pre-processing step of the data, an FFT evaluation and a post-processing step. Pre- and post-processing account for the shift and scaling of the real-space and Fourier-space grids.

The sign convention ('-' vs. '+') can be changed with the `sign` parameter.

See also:

`dft_preprocess_data`, `pyfftw_call`, `dft_postprocess_data`

Attributes

<code>adjoint</code>	The adjoint Fourier transform.
<code>axes</code>	Axes along the FT is calculated by this operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>halfcomplex</code>	Return True if the last transform axis is halved.
<code>impl</code>	Backend for the FFT implementation.
<code>inverse</code>	The inverse Fourier transform.
<code>is_functional</code>	True if the this operator's range is a <i>Field</i> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.
<code>shifts</code>	Return the boolean list indicating shifting per axis.
<code>sign</code>	Sign of the complex exponent in the transform.

FourierTransform.adjoint

`FourierTransform.adjoint`

The adjoint Fourier transform.

FourierTransform.axes

`FourierTransform.axes`

Axes along the FT is calculated by this operator.

FourierTransform.domain

`FourierTransform.domain`

Set of objects on which this operator can be evaluated.

FourierTransform.halfcomplex

`FourierTransform.halfcomplex`

Return True if the last transform axis is halved.

FourierTransform.impl

`FourierTransform.impl`

Backend for the FFT implementation.

FourierTransform.inverse`FourierTransform.inverse`

The inverse Fourier transform.

FourierTransform.is_functional`FourierTransform.is_functional`True if the this operator's range is a *Field*.**FourierTransform.is_linear**`FourierTransform.is_linear`

True if this operator is linear.

FourierTransform.range`FourierTransform.range`

Set in which the result of an evaluation of this operator lies.

FourierTransform.shifts`FourierTransform.shifts`

Return the boolean list indicating shifting per axis.

FourierTransform.sign`FourierTransform.sign`

Sign of the complex exponent in the transform.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x, out, **kwargs)</code>	Implement <code>self(x, out[, **kwargs])</code> .
<code>clear_fftw_plan()</code>	Delete the FFTW plan of this transform.
<code>clear_temporaries()</code>	Set the temporaries to None.
<code>create_temporaries([r, f])</code>	Allocate and store reusable temporaries.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .
<code>init_fftw_plan([planning_effort])</code>	Initialize the FFTW plan for this transform for later use.

FourierTransform.__call__`FourierTransform.__call__(x, out=None, **kwargs)`Return `self(x[, out, **kwargs])`.Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.**Parameters**`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns `out` : `Operator.range element`

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

FourierTransform._call

`FourierTransform._call(x, out, **kwargs)`
 Implement `self(x, out[, **kwargs])`.

Parameters `x` : domain `element`

Discretized function to be transformed

out : range `element`

Element to which the output is written

See also:

`pyfftw_call` Call pyfftw backend directly

Notes

See the `pyfftw_call` function for `**kwargs` options. The parameters `axes` and `halfcomplex` cannot be overridden.

FourierTransform.clear_fftw_plan

`FourierTransform.clear_fftw_plan()`
 Delete the FFTW plan of this transform.

FourierTransform.clear_temporaries`FourierTransform.clear_temporaries()`

Set the temporaries to None.

FourierTransform.create_temporaries`FourierTransform.create_temporaries(r=True, f=True)`

Allocate and store reusable temporaries.

Existing temporaries are overwritten.

Parameters `r` : bool, optional

Create temporary for the real space

`f` : bool, optional

Create temporary for the frequency space

See also:

`clear_temporaries`

`clear_fftw_plan` can also hold references to the temporaries

Notes

To save memory, clear the temporaries when the transform is no longer used.

FourierTransform.derivative`FourierTransform.derivative(point)`

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

FourierTransform.init_fftw_plan`FourierTransform.init_fftw_plan(planning_effort='measure', **kwargs)`

Initialize the FFTW plan for this transform for later use.

If the implementation of this operator is not 'pyfftw', this method has no effect.

Parameters `planning_effort` : {'estimate', 'measure', 'patient', 'exhaustive'}

Flag for the amount of effort put into finding an optimal FFTW plan. See the [FFTW doc on planner flags](#).

planning_timelimit : float, optional

Limit planning time to roughly this amount of seconds. Default: None (no limit)

threads : int, optional

Number of threads to use. Default: 1

See also:

`clear_fftw_plan`

Notes

To save memory, clear the plan when the transform is no longer used (the plan stores 2 arrays).

`__init__` (*dom*, *ran*=None, *impl*='numpy', ***kwargs*)

Initialize a new instance.

Parameters*dom* : *DiscreteLp*

Domain of the Fourier transform. If the *DiscreteLp.exponent* of *dom* and *ran* are equal to 2.0, this operator has an adjoint which is equal to its inverse.

ran : *DiscreteLp*, optional

Range of the Fourier transform. If not given, the range is determined from *dom* and the other parameters. The exponent is chosen to be the conjugate $p / (p - 1)$, which reads as 'inf' for $p=1$ and 1 for $p=\text{'inf'}$.

impl : {'numpy', 'pyfftw'}

Backend for the FFT implementation. The 'pyfftw' backend is faster but requires the *pyfftw* package.

axes : sequence of int, optional

Dimensions along which to take the transform. Default: all axes

sign : {'-', '+'}, optional

Sign of the complex exponent. Default: '-'

halfcomplex : bool, optional

If True, calculate only the negative frequency part along the last axis for real input. If False, calculate the full complex FFT. For complex *dom*, it has no effect. Default: True

shift : bool or sequence of bool, optional

If True, the reciprocal grid is shifted by half a stride in the negative direction. With a boolean sequence, this option is applied separately to each axis. If a sequence is provided, it must have the same length as *axes* if supplied. Note that this must be set to True in the halved axis in half-complex transforms. Default: True

Other Parameters*tmp_r* : *DiscreteLpVector* or *numpy.ndarray*

Temporary for calculations in the real space (domain of this transform). It is shared with the inverse.

Variants using this: R2C, R2HC, C2R (inverse)

tmp_f : *DiscreteLpVector* or *numpy.ndarray*

Temporary for calculations in the frequency (reciprocal) space. It is shared with the inverse.

Variants using this: R2C, C2R (inverse), HC2R (inverse)

Notes

- The transform variants are:

- C2C: complex-to-complex. The default variant, one-to-one and unitary.

–**R2C**: real-to-complex. This variant has no adjoint, and the inverse may suffer from information loss since the result is cast to real.

–**R2HC**: real-to-halfcomplex. This variant stores only a half-space of frequencies and is guaranteed to be one-to-one (invertible).

- The `Operator.range` of this operator always has the `ComplexNumbers` as `LinearSpace.field`, i.e. if the field of `dom` is the `RealNumbers`, this operator has no `Operator.adjoint`.

FourierTransformInverse

```
class odl.trafos.fourier.FourierTransformInverse(ran, dom=None, impl='numpy',
**kwargs)
```

Bases: `odl.trafos.fourier.FourierTransform`

Inverse of the discretized Fourier transform between L^p spaces.

This operator is the exact inverse of the `FourierTransform`, and **not** a discretization of the Fourier integral with “+” sign in the complex exponent. For the latter, use the `sign` parameter of the forward transform.

See also:

`FourierTransform`

Attributes

<code>adjoint</code>	The adjoint Fourier transform.
<code>axes</code>	Axes along the FT is calculated by this operator.
<code>domain</code>	Set of objects on which this operator can be evaluated.
<code>halfcomplex</code>	Return <code>True</code> if the last transform axis is halved.
<code>impl</code>	Backend for the FFT implementation.
<code>inverse</code>	Inverse of the inverse, the forward FT.
<code>is_functional</code>	True if the this operator’s range is a <code>Field</code> .
<code>is_linear</code>	True if this operator is linear.
<code>range</code>	Set in which the result of an evaluation of this operator lies.
<code>shifts</code>	Return the boolean list indicating shifting per axis.
<code>sign</code>	Sign of the complex exponent in the transform.

FourierTransformInverse.adjoint

`FourierTransformInverse.adjoint`

The adjoint Fourier transform.

FourierTransformInverse.axes

`FourierTransformInverse.axes`

Axes along the FT is calculated by this operator.

FourierTransformInverse.domain

`FourierTransformInverse.domain`

Set of objects on which this operator can be evaluated.

FourierTransformInverse.halfcomplex

FourierTransformInverse.halfcomplex
Return True if the last transform axis is halved.

FourierTransformInverse.impl
FourierTransformInverse.impl
Backend for the FFT implementation.

FourierTransformInverse.inverse
FourierTransformInverse.inverse
Inverse of the inverse, the forward FT.

FourierTransformInverse.is_functional
FourierTransformInverse.is_functional
True if the this operator's range is a *Field*.

FourierTransformInverse.is_linear
FourierTransformInverse.is_linear
True if this operator is linear.

FourierTransformInverse.range
FourierTransformInverse.range
Set in which the result of an evaluation of this operator lies.

FourierTransformInverse.shifts
FourierTransformInverse.shifts
Return the boolean list indicating shifting per axis.

FourierTransformInverse.sign
FourierTransformInverse.sign
Sign of the complex exponent in the transform.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x, out, **kwargs)</code>	Implement <code>self(x, out[, **kwargs])</code> .
<code>clear_fftw_plan()</code>	Delete the FFTW plan of this transform.
<code>clear_temporaries()</code>	Set the temporaries to None.
<code>create_temporaries([r, f])</code>	Allocate and store reusable temporaries.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .
<code>init_fftw_plan([planning_effort])</code>	Initialize the FFTW plan for this transform for later use.

FourierTransformInverse.__call__
FourierTransformInverse.__call__ (*x*, *out=None*, ***kwargs*)
Return `self(x[, out, **kwargs])`.
Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters`x` : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns`out` : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

[`_call`](#) Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

FourierTransformInverse._call

`FourierTransformInverse._call(x, out, **kwargs)`

Implement `self(x, out[, **kwargs])`.

Parameters`x` : domain *element*

Discretized function to be transformed

out : range *element*

Element to which the output is written

See also:

[`pyfftw_call`](#) Call pyfftw backend directly

Notes

See the `pyfftw_call` function for `**kwargs` options. The parameters `axes` and `halfcomplex` cannot be overridden.

FourierTransformInverse.clear_fftw_plan

`FourierTransformInverse.clear_fftw_plan()`

Delete the FFTW plan of this transform.

FourierTransformInverse.clear_temporaries

`FourierTransformInverse.clear_temporaries()`

Set the temporaries to None.

FourierTransformInverse.create_temporaries

`FourierTransformInverse.create_temporaries(r=True, f=True)`

Allocate and store reusable temporaries.

Existing temporaries are overwritten.

Parameters `r` : bool, optional

Create temporary for the real space

f : bool, optional

Create temporary for the frequency space

See also:

`clear_temporaries`

`clear_fftw_plan` can also hold references to the temporaries

Notes

To save memory, clear the temporaries when the transform is no longer used.

FourierTransformInverse.derivative

`FourierTransformInverse.derivative(point)`

Return the operator derivative at `point`.

Raises `OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

FourierTransformInverse.init_fftw_plan

`FourierTransformInverse.init_fftw_plan(planning_effort='measure', **kwargs)`

Initialize the FFTW plan for this transform for later use.

If the implementation of this operator is not 'pyfftw', this method has no effect.

Parameters `planning_effort` : {'estimate', 'measure', 'patient', 'exhaustive'}

Flag for the amount of effort put into finding an optimal FFTW plan. See the [FFTW doc on planner flags](#).

planning_timelimit : float, optional

Limit planning time to roughly this amount of seconds. Default: None (no limit)

threads : int, optional

Number of threads to use. Default: 1

See also:

`clear_fftw_plan`

Notes

To save memory, clear the plan when the transform is no longer used (the plan stores 2 arrays).

`__init__(ran, dom=None, impl='numpy', **kwargs)`

Parameters
ran : *DiscreteLp*

Range of the inverse Fourier transform. If the *DiscreteLp.exponent* of **dom** and **ran** are equal to 2.0, this operator has an adjoint which is equal to its inverse.

dom : *DiscreteLp*, optional

Domain of the inverse Fourier transform. If not given, the domain is determined from **ran** and the other parameters. The exponent is chosen to be the conjugate $p / (p - 1)$, which reads as 'inf' for $p=1$ and 1 for $p='inf'$.

impl : {'numpy', 'pyfftw'}

Backend for the FFT implementation. The 'pyfftw' backend is faster but requires the pyfftw package.

axes : sequence of int, optional

Dimensions along which to take the transform. Default: all axes

sign : {'-', '+'}, optional

Sign of the complex exponent. Default: '+'

halfcomplex : bool, optional

If True, calculate only the negative frequency part along the last axis for real input. If False, calculate the full complex FFT. For complex **dom**, it has no effect. Default: True

shift : bool or sequence of bool, optional

If True, the reciprocal grid is shifted by half a stride in the negative direction. With a boolean sequence, this option is applied separately to each axis. If a sequence is provided, it must have the same length as **axes** if supplied. Note that this must be set to True in the halved axis in half-complex transforms. Default: True

Other Parameters
tmp_r : *DiscreteLpVector* or `numpy.ndarray`

Temporary for calculations in the real space (range of this transform). It is shared with the inverse.

Variants using this: C2R, R2C (forward), R2HC (forward)

tmp_f : *DiscreteLpVector* or `numpy.ndarray`

Temporary for calculations in the frequency (reciprocal) space. It is shared with the inverse.

Variants using this: C2R, HC2R, R2C (forward)

Notes

- The transform variants are:
 - C2C**: complex-to-complex. The default variant, one-to-one and unitary.
 - C2R**: complex-to-real. This variant has no adjoint and may suffer from information loss since the result is cast to real.
 - HC2R**: halfcomplex-to-real. This variant interprets input as a signal on a half-space of frequencies. It is guaranteed to be one-to-one (invertible).
- The `Operator.domain` of this operator always has the `ComplexNumbers` as `LinearSpace.field`, i.e. if the field of `ran` is the `RealNumbers`, this operator has no `Operator.adjoint`.

Functions

<code>dft_postprocess_data(arr, real_grid, ..., ...)</code>	Post-process the Fourier-space data after DFT.
<code>dft_preprocess_data(arr[, shift, axes, ...])</code>	Pre-process the real-space data before DFT.
<code>inverse_reciprocal(grid, x0[, axes, ...])</code>	Return the inverse reciprocal of the given regular grid.
<code>pyfftw_call(array_in, array_out[, ...])</code>	Calculate the DFT with pyfftw.
<code>reciprocal(grid[, shift, axes, halfcomplex])</code>	Return the reciprocal of the given regular grid.
<code>reciprocal_space(space[, axes, halfcomplex, ...])</code>	Return the range of the Fourier transform on <code>space</code> .

dft_postprocess_data

`odl.trafos.fourier.dft_postprocess_data(arr, real_grid, recip_grid, shifts, axes, interp, sign='-', op='multiply', out=None)`

Post-process the Fourier-space data after DFT.

This function multiplies the given data with the separable function:

$$q(\mathbf{x}) = \exp(\pm 1j * \text{dot}(\mathbf{x}[0], \mathbf{x}_i)) * s * \text{phi_hat}(\mathbf{x}_i_\text{bar})$$

where `x[0]` and `s` are the minimum point and the stride of the real-space grid, respectively, and `phi_hat(x_i_bar)` is the FT of the interpolation kernel. The sign of the exponent depends on the choice of `sign`. Note that for `op='divide'` the multiplication with `s * phi_hat(x_i_bar)` is replaced by a division with the same array.

In discretized form on the reciprocal grid, the exponential part of this function becomes an array:

$$q[k] = \exp(\pm 1j * \text{dot}(\mathbf{x}[0], \mathbf{x}_i[k]))$$

and the arguments `x_i_bar` to the interpolation kernel are the normalized frequencies:

```
for 'shift=True' : x_i_bar[k] = -pi + pi * (2*k) / N
for 'shift=False': x_i_bar[k] = -pi + pi * (2*k+1) / N
```

See [Pre+2007], Section 13.9 “Computing Fourier Integrals Using the FFT” for a similar approach.

Parameters`arr` : *array-like*

Array to be pre-processed. An array with real data type is converted to its complex counterpart.

real_grid : *RegularGrid*

Real space grid in the transform

recip_grid : *RegularGrid*

Reciprocal grid in the transform

shifts : sequence of bool

If `True`, the grid is shifted by half a stride in the negative direction in the corresponding axes. The sequence must have the same length as `axes`.

axes : sequence of int

Dimensions along which to take the transform. The sequence must have the same length as `shifts`.

interp : str or sequence of str

Interpolation scheme used in the real-space

sign : {'-', '+'}, optional

Sign of the complex exponent

op : {'multiply', 'divide'}

Operation to perform with the stride times the interpolation kernel FT

out : `numpy.ndarray`, optional

Array in which the result is stored. If `out` is `arr`, an in-place modification is performed.

Returns`out` : `numpy.ndarray`

Result of the post-processing. If `out` was given, the returned object is a reference to it.

dft_preprocess_data

`odl.trafos.fourier.dft_preprocess_data(arr, shift=True, axes=None, sign='-', out=None)`
Pre-process the real-space data before DFT.

This function multiplies the given data with the separable function:

$$p(x) = \exp(\pm 1j * \text{dot}(x - x[0], xi[0]))$$

where `x[0]` and `xi[0]` are the minimum coordinates of the real-space and reciprocal grids, respectively. The sign of the exponent depends on the choice of `sign`. In discretized form, this function becomes an array:

$$p[k] = \exp(\pm 1j * k * s * xi[0])$$

If the reciprocal grid is not shifted, i.e. symmetric around 0, it is `xi[0] = pi/s * (-1 + 1/N)`, hence:

$$p[k] = \exp(\pm 1j * pi * k * (1 - 1/N))$$

For a shifted grid, we have `xi[0] = -pi/s`, thus the array is given by:

$$p[k] = (-1)**k$$

Parameters`sarr` : *array-like*

Array to be pre-processed. If its data type is a real non-floating type, it is converted to 'float64'.

shift : `bool` or sequence of `bool`, optional

If `True`, the grid is shifted by half a stride in the negative direction. With a sequence, this option is applied separately on each axis.

axes : sequence of `int`, optional

Dimensions in which to calculate the reciprocal. The sequence must have the same length as `shift` if the latter is given as a sequence. `None` means all axes in `dfunc`.

sign : {'-', '+'}, optional

Sign of the complex exponent

out : `numpy.ndarray`, optional

Array in which the result is stored. If `out` is `arr`, an in-place modification is performed. For real data type, this is only possible for `shift=True` since the factors are complex otherwise.

Returns`out` : `numpy.ndarray`

Result of the pre-processing. If `out` was given, the returned object is a reference to it.

Notes

If `out` is not specified, the data type of the returned array is the same as that of `arr` except when `arr` has real data type and `shift` is not `True`. In this case, the return type is the complex counterpart of `arr.dtype`.

inverse_reciprocal

```
odl.trafos.fourier.inverse_reciprocal(grid, x0, axes=None, halfcomplex=False,
                                     halfcx_parity='even')
```

Return the inverse reciprocal of the given regular grid.

Given a reciprocal grid:

```
xi[j] = xi[0] + j * sigma,
```

with a multi-index `j = (j[0], ..., j[d-1])` in the range `0 <= j < M`, this function calculates the original grid:

```
x[k] = x[0] + k * s
```

by using a provided `x[0]` and calculating the stride `s`.

If the reciprocal grid is interpreted as coming from a usual complex-to-complex FFT, it is `N == M`, and the stride is:

```
s = 2*pi / (sigma * N)
```

For a reciprocal grid from a real-to-complex (half-complex) FFT, it is `M[i] = floor(N[i]/2) + 1` in the last transform axis `i`. To resolve the ambiguity regarding the parity of `N[i]`, the it must be specified if the output shape should be even or odd, resulting in:

```
odd : N[i] = 2 * M[i] - 1
even: N[i] = 2 * M[i] - 2
```

The output stride is calculated with this `N` as above in this case.

Parameters`grid` : *RegularGrid*

Original sampling grid

x0 : array-like

Minimal point of the inverse reciprocal grid

axes : sequence of `int`, optional

Dimensions in which to calculate the reciprocal. The sequence must have the same length as `shift` if the latter is given as a sequence. `None` means all axes in `grid`.

halfcomplex : `bool`, optional

If `True`, interpret the given grid as the reciprocal as used in a half-complex FFT (see above). Otherwise, the grid is regarded as being used in a complex-to-complex transform.

halfcx_parity : {'even', 'odd'}

Use this parity for the shape of the returned grid in the last axis of `axes` in the case `halfcomplex=True`

Returns`sirecip` : *RegularGrid*

The inverse reciprocal grid

`pyfftw_call`

`odl.trafos.fourier.pyfftw_call` (*array_in*, *array_out*, *direction='forward'*, *axes=None*, *halfcomplex=False*, ***kwargs*)

Calculate the DFT with `pyfftw`.

The discrete Fourier (forward) transform calculates the sum:

```
f_hat[k] = sum_j( f[j] * exp(-2*pi*1j * j*k/N) )
```

where the summation is taken over all indices `j = (j[0], ..., j[d-1])` in the range `0 <= j < N` (component-wise), with `N` being the shape of the input array.

The output indices `k` lie in the same range, except for half-complex transforms, where the last axis `i` in `axes` is shortened to `0 <= k[i] < floor(N[i]/2) + 1`.

In the backward transform, sign of the the exponential argument is flipped.

Parameters`array_in` : `numpy.ndarray`

Array to be transformed

array_out : `numpy.ndarray`

Output array storing the transformed values, may be aligned with `array_in`.

direction : {'forward', 'backward'}

Direction of the transform

axes : sequence of `int`, optional

Dimensions along which to take the transform. `None` means using all axis and is equivalent to `np.arange(ndim)`.

halfcomplex : `bool`, optional

If `True`, calculate only the negative frequency part along the last axis. If `False`, calculate the full complex FFT. This option can only be used with real input data.

Returns`sfftplan` : `pyfftw.FFTW`

The plan object created from the input arguments. It can be reused for transforms of the same size with the same data types. Note that reuse only gives a speedup if the initial plan used a planner flag other than `'estimate'`. If `fftw_plan` was specified, the returned object is a reference to it.

Other Parameters`sfftplan` : `pyfftw.FFTW`, optional

Use this plan instead of calculating a new one. If specified, the options `planning_effort`, `planning_timelimit` and `threads` have no effect.

planning_effort : `{'estimate', 'measure', 'patient', 'exhaustive'}`

Flag for the amount of effort put into finding an optimal FFTW plan. See the [FFTW doc on planner flags](#). Default: `'estimate'`.

planning_timelimit : `float`, optional

Limit planning time to roughly this amount of seconds. Default: `None` (no limit)

threads : `int`, optional

Number of threads to use. Default: Number of CPUs if the number of data points is larger than 1000, else 1.

normalise_idft : `bool`, optional

If `True`, the backward transform is normalized by $1 / N$, where N is the total number of points in `array_in[axes]`. This ensures that the IDFT is the true inverse of the forward DFT. Default: `False`

import_wisdom : filename or file handle, optional

File to load FFTW wisdom from. If the file does not exist, it is ignored.

export_wisdom : filename or file handle, optional

File to append the accumulated FFTW wisdom to

Notes

- The planning and direction flags can also be specified as capitalized and prepended by `'FFTW_'`, i.e. in the original FFTW form.
- For a `halfcomplex` forward transform, the arrays must fulfill `array_out.shape[axes[-1]] == array_in.shape[axes[-1]] // 2 + 1`, and vice versa for backward transforms.
- All planning schemes except `'estimate'` require an internal copy of the input array but are often several times faster after the first call (measuring results are cached). Typically, `'measure'` is a good compromise. If you cannot afford the copy, use `'estimate'`.
- If a plan is provided via the `fftw_plan` parameter, no copy is needed internally.

reciprocal

`odl.trafos.fourier.reciprocal(grid, shift=True, axes=None, halfcomplex=False)`

Return the reciprocal of the given regular grid.

This function calculates the reciprocal (Fourier/frequency space) grid for a given regular grid defined by the nodes:

$$x[k] = x[0] + k * s,$$

where $k = (k[0], \dots, k[d-1])$ is a d -dimensional index in the range $0 \leq k < N$ (component-wise). The multi-index N is the shape of the input grid. This grid's reciprocal is then given by the nodes:

$$xi[j] = xi[0] + j * sigma,$$

with the reciprocal grid stride $\sigma = 2\pi / (s * N)$. The minimum frequency $xi[0]$ can in principle be chosen freely, but usually it is chosen in a such a way that the reciprocal grid is centered around zero. For this, there are two possibilities:

1. Make the grid point-symmetric around 0.
2. Make the grid “almost” point-symmetric around zero by shifting it to the left by half a reciprocal stride.

In the first case, the minimum frequency (per axis) is given as:

$$xi_1[0] = -\pi/s + \pi/(s*n) = -\pi/s + \sigma/2.$$

For the second case, it is:

$$xi_1[0] = -\pi / s.$$

Note that the zero frequency is contained in case 1 for an odd number of points, while for an even size, the second option guarantees that 0 is contained.

If a real-to-complex (half-complex) transform is to be computed, the reciprocal grid has the shape $M[i] = \text{floor}(N[i]/2) + 1$ in the last transform axis i .

Parameters**grid** : *RegularGrid*

Original sampling grid

shift : `bool` or sequence of `bool`, optional

If `True`, the grid is shifted by half a stride in the negative direction. With a sequence, this option is applied separately on each axis.

axes : sequence of `int`, optional

Dimensions in which to calculate the reciprocal. The sequence must have the same length as `shift` if the latter is given as a sequence. `None` means all axes in `grid`.

halfcomplex : `bool`, optional

If `True`, return the half of the grid with last coordinate less than zero. This is related to the fact that for real-valued functions, the other half is the mirrored complex conjugate of the given half and therefore needs not be stored.

Returns**recip** : *RegularGrid*

The reciprocal grid

reciprocal_space

`odl.trafos.fourier.reciprocal_space(space, axes=None, halfcomplex=False, shift=True, **kwargs)`

Return the range of the Fourier transform on `space`.

Parameters`space` : *DiscreteLp*

Real space whose reciprocal is calculated. It must be uniformly discretized.

axes : sequence of `int`, optional

Dimensions along which the Fourier transform is taken. Default: all axes

halfcomplex : `bool`, optional

If `True`, take only the negative frequency part along the last axis for. If `False`, use the full frequency space. This option can only be used if `space` is a space of real-valued functions. Default: `False`

shift : `bool` or sequence of `bool`, optional

If `True`, the reciprocal grid is shifted by half a stride in the negative direction. With a boolean sequence, this option is applied separately to each axis. If a sequence is provided, it must have the same length as `axes` if supplied. Note that this must be set to `True` in the halved axis in half-complex transforms. Default: `True`

exponent : `float`, optional

Create a space with this exponent. By default, the conjugate exponent $q = p / (p - 1)$ of the exponent of `space` is used, where $q = \text{inf}$ for $p = 1$ and vice versa.

dtype : optional

Complex data type of the reciprocal space. By default, the complex counterpart of `space.dtype` is used.

Returns`rspace` : *DiscreteLp*

Reciprocal of the input `space`. If `halfcomplex=True`, the upper end of the domain (where the half space ends) is chosen to coincide with the grid node.

wavelet

Discrete wavelet transformation on L2 spaces.

Classes

<code>WaveletTransform(dom, nscales, wbasis, mode)</code>	Discrete wavelet trafo between discrete L2 spaces.
<code>WaveletTransformInverse(ran, nscales, ...)</code>	Discrete inverse wavelet trafo between discrete L2 spaces.

WaveletTransform

class `odl.trafos.wavelet.WaveletTransform(dom, nscales, wbasis, mode)`

Bases: `odl.operator.operator.Operator`

Discrete wavelet trafo between discrete L2 spaces.

Attributes

<i>adjoint</i>	The adjoint wavelet transform.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	The inverse wavelet transform.
<i>is_biorthogonal</i>	Whether or not the wavelet basis is bi-orthogonal.
<i>is_functional</i>	True if the this operator's range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>is_orthogonal</i>	Whether or not the wavelet basis is orthogonal.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

WaveletTransform.adjoint

WaveletTransform.**adjoint**

The adjoint wavelet transform.

WaveletTransform.domain

WaveletTransform.**domain**

Set of objects on which this operator can be evaluated.

WaveletTransform.inverse

WaveletTransform.**inverse**

The inverse wavelet transform.

WaveletTransform.is_biorthogonal

WaveletTransform.**is_biorthogonal**

Whether or not the wavelet basis is bi-orthogonal.

WaveletTransform.is_functional

WaveletTransform.**is_functional**

True if the this operator's range is a *Field*.

WaveletTransform.is_linear

WaveletTransform.**is_linear**

True if this operator is linear.

WaveletTransform.is_orthogonal

WaveletTransform.**is_orthogonal**

Whether or not the wavelet basis is orthogonal.

WaveletTransform.range

WaveletTransform.**range**

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(x)</code>	Compute the discrete wavelet transform.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

WaveletTransform.__call__

WaveletTransform.**__call__**(*x*, *out=None*, ***kwargs*)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters*x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns*out* : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`_call` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

WaveletTransform._call`WaveletTransform._call(x)`

Compute the discrete wavelet transform.

Parameters`x` : *DiscreteLpVector***Returns**`sarr` : `numpy.ndarray`

Flattened and concatenated coefficient array The length of the array depends on the size of input image to be transformed and on the chosen wavelet basis.

WaveletTransform.derivative`WaveletTransform.derivative(point)`Return the operator derivative at `point`.**Raises**`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

`__init__(dom, nscales, wbasis, mode)`

Initialize a new instance.

Parameters`dom` : *DiscreteLp*Domain of the wavelet transform (the “image domain”). The exponent p of the discrete L^p space must be equal to 2.0.**nscales** : `int`Number of scales in the coefficient list. The maximum number of usable scales can be determined by `pywt.dwt_max_level`. For more information see the corresponding [documentation of PyWavelets](#).**wbasis** : `{str, pywt.Wavelet}`If a string is given, converts to a `pywt.Wavelet`. Describes properties of a selected wavelet basis. See [PyWavelet documentation](#)

Possible wavelet families are:

Haar (`haar`)Daubechies (`db`)Symlets (`sym`)Coiflets (`coif`)Biorthogonal (`bior`)Reverse biorthogonal (`rbio`)Discrete FIR approximation of Meyer wavelet (`dmey`)**mode** : `str`Signal extention modes as defined by `pywt.MODES.modes`
<http://www.pybytes.com/pywavelets/ref/signal-extension-modes.html>

Possible extension modes are:

‘zpd’: zero-padding – signal is extended by adding zero samples

‘cpd’: constant padding – border values are replicated

‘sym’: symmetric padding – signal extension by mirroring samples

‘ppd’: periodic padding – signal is treated as a periodic one

‘sp1’: smooth padding – signal is extended according to the first derivatives calculated on the edges (straight line)

‘per’: periodization – like periodic-padding but gives the smallest possible number of decomposition coefficients.

Examples

```
>>> import odl, pywt
>>> wbasis = pywt.Wavelet('db1')
>>> discr_domain = odl.uniform_discr([0, 0], [1, 1], (16, 16))
>>> op = WaveletTransform(discr_domain, nscales=1,
...                       wbasis=wbasis, mode='per')
>>> op.is_biorthogonal
True
```

WaveletTransformInverse

class odl.trafos.wavelet.**WaveletTransformInverse** (*ran, nscales, wbasis, mode*)

Bases: *odl.operator.operator.Operator*

Discrete inverse wavelet trafo between discrete L2 spaces.

Attributes

<i>adjoint</i>	The adjoint wavelet transform.
<i>domain</i>	Set of objects on which this operator can be evaluated.
<i>inverse</i>	The inverse wavelet transform.
<i>is_biorthogonal</i>	Whether or not the wavelet basis is bi-orthogonal.
<i>is_functional</i>	True if the this operator’s range is a <i>Field</i> .
<i>is_linear</i>	True if this operator is linear.
<i>is_orthogonal</i>	Whether or not the wavelet basis is orthogonal.
<i>range</i>	Set in which the result of an evaluation of this operator lies.

WaveletTransformInverse.adjoint

WaveletTransformInverse.**adjoint**

The adjoint wavelet transform.

WaveletTransformInverse.domain

WaveletTransformInverse.**domain**

Set of objects on which this operator can be evaluated.

WaveletTransformInverse.inverse

WaveletTransformInverse.**inverse**

The inverse wavelet transform.

WaveletTransformInverse.is_biorthogonal

WaveletTransformInverse.is_biorthogonal

Whether or not the wavelet basis is bi-orthogonal.

WaveletTransformInverse.is_functional

WaveletTransformInverse.is_functional

True if the this operator's range is a *Field*.

WaveletTransformInverse.is_linear

WaveletTransformInverse.is_linear

True if this operator is linear.

WaveletTransformInverse.is_orthogonal

WaveletTransformInverse.is_orthogonal

Whether or not the wavelet basis is orthogonal.

WaveletTransformInverse.range

WaveletTransformInverse.range

Set in which the result of an evaluation of this operator lies.

Methods

<code>__call__(x[, out])</code>	Return <code>self(x[, out, **kwargs])</code> .
<code>__eq__</code>	Return <code>self==value</code> .
<code>_call(coeff)</code>	Compute the discrete 1D, 2D or 3D inverse wavelet transform.
<code>derivative(point)</code>	Return the operator derivative at <code>point</code> .

WaveletTransformInverse.__call__

WaveletTransformInverse.__call__ (*x*, *out=None*, ***kwargs*)

Return `self(x[, out, **kwargs])`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parameters*x* : *Operator.domain element-like*

An object which can be converted into an element of this operator's domain with the `self.domain.element` method. The operator is applied to this object, which is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range element*, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

kwargs : Further arguments to the function, optional

Passed on to the underlying implementation in `_call`

Returns*out* : *Operator.range element*

Result of the operator evaluation. If `out` was provided, the returned object is a reference to it.

See also:

`__call__` Implementation of the method

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

WaveletTransformInverse.__call

WaveletTransformInverse.`__call__`(*coeff*)

Compute the discrete 1D, 2D or 3D inverse wavelet transform.

Parameters*coeff* : *DiscreteLpVector*

Returns*sarr* : *DiscreteLpVector*

WaveletTransformInverse.derivative

WaveletTransformInverse.`derivative`(*point*)

Return the operator derivative at *point*.

Raises`OpNotImplementedError`

If the operator is not linear, the derivative cannot be default implemented.

`__init__`(*ran*, *nscales*, *wbasis*, *mode*)

Initialize a new instance.

Parameters*dom* : *DiscreteLp*

Domain of the wavelet transform (the “image domain”). The exponent p of the discrete L^p space must be equal to 2.0.

nscales : `int`

Number of scales in the coefficient list. The maximum number of usable scales can be determined by `pywt.dwt_max_level`. For more information see the corresponding [documentation of PyWavelets](#).

wbasis : `pywt.Wavelet`

Describes properties of a selected wavelet basis. See PyWavelet [documentation](#)

Possible wavelet families are:

Haar (`haar`)

Daubechies (`db`)

Symlets (`sym`)

Coiflets (`coif`)

Biorthogonal (`bior`)

Reverse biorthogonal (`rbio`)

Discrete FIR approximation of Meyer wavelet (`dmey`)

mode : str

Signal extension modes as defined by `pywt.MODES.modes`
<http://www.pybytes.com/pywavelets/ref/signal-extension-modes.html>

Possible extension modes are:

‘zpd’: zero-padding – signal is extended by adding zero samples

‘cpd’: constant padding – border values are replicated

‘sym’: symmetric padding – signal extension by mirroring samples

‘ppd’: periodic padding – signal is treated as a periodic one

‘spl’: smooth padding – signal is extended according to the first derivatives calculated on the edges (straight line)

‘per’: periodization – like periodic-padding but gives the smallest possible number of decomposition coefficients.

Functions

<code>array_to_pywt_coeff(coeff, size_list)</code>	Convert a flat array into a <code>pywt</code> coefficient list.
<code>coeff_size_list(shape, nscales, wbasis, mode)</code>	Construct a size list from given wavelet coefficients.
<code>pywt_coeff_to_array(coeff, size_list)</code>	Convert a Pywavelets coefficient list into a flat array.
<code>wavelet_decomposition3d(x, wbasis, mode, nscales)</code>	Discrete 3D multiresolution wavelet decomposition.
<code>wavelet_reconstruction3d(coeff_list, wbasis, ...)</code>	Discrete 3D multiresolution wavelet reconstruction

`array_to_pywt_coeff`

`odl.trafos.wavelet.array_to_pywt_coeff(coeff, size_list)`

Convert a flat array into a `pywt` coefficient list.

For multilevel 1D, 2D and 3D discrete wavelet transforms.

Parameters`coeff` : *DiscreteLpVector*

A flat coefficient vector containing the approximation, and detail coefficients in the following order [aaaN, aadN, adaN, addN, daaN, dadN, ddaN, dddN, ... aad1, ada1, add1, daa1, dad1, dda1, ddd1]

size_list : list

A list of coefficient sizes such that,

size_list[0] = size of approximation coefficients at the coarsest level,

size_list[1] = size of the detailed details at the coarsest level,

size_list[N] = size of the detailed coefficients at the finest level,

`size_list[N+1]` = size of original image,

`N` = the number of scaling levels

Returns`coeff` : ordered list

Coefficient are organized in the list in the following way:

In 1D:

`[aN, (dN), ... (d1)]`

The abbreviations refer to

`a` = approximation,

`d` = detail,

In 2D:

`[aaN, (adN, daN, ddN), ... (ad1, da1, dd1)]`

The abbreviations refer to

`aa` = approx. on 1st dim, approx. on 2nd dim (approximation),

`ad` = approx. on 1st dim, detail on 2nd dim (horizontal),

`da` = detail on 1st dim, approx. on 2nd dim (vertical),

`dd` = detail on 1st dim, detail on 2nd dim (diagonal),

In 3D:

`[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ...
(aad1, ada1, add1, daa1, dad1, dda1, ddd1)]`

The abbreviations refer to

`aaa` = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`aad` = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`ada` = approx. on 1st dim, detail on 3rd dim, approx. on 3rd dim,

`add` = approx. on 1st dim, detail on 3rd dim, detail on 3rd dim,

`daa` = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`dad` = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`dda` = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

`ddd` = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

`N` refers to the number of scaling levels

`coeff_size_list`

`odl.trafos.wavelet.coeff_size_list(shape, nscales, wbasis, mode)`

Construct a size list from given wavelet coefficients.

Related to 1D, 2D and 3D multidimensional wavelet transforms that utilize [PyWavelets](#).

Parameters`shape` : tuple

Number of pixels/voxels in the image. Its length must be 1, 2 or 3.

nscales : int

Number of scales in the multidimensional wavelet transform. This parameter is checked against the maximum number of scales returned by `pywt.dwt_max_level`. For more information see the [PyWavelets documentation on the maximum level of scales](#).

wbasis : `pywt.Wavelet`

Selected wavelet basis. For more information see the [PyWavelets documentation on wavelet bases](#).

mode : str

Signal extension mode. Possible extension modes are

‘zpd’: zero-padding – signal is extended by adding zero samples

‘cpd’: constant padding – border values are replicated

‘sym’: symmetric padding – signal extension by mirroring samples

‘ppd’: periodic padding – signal is treated as a periodic one

‘spl’: smooth padding – signal is extended according to the first derivatives calculated on the edges (straight line)

‘per’: periodization – like periodic-padding but gives the smallest possible number of decomposition coefficients.

Returnssize_list : list

A list containing the sizes of the wavelet (approximation and detail) coefficients at different scaling levels:

`size_list[0]` = size of approximation coefficients at the coarsest level

`size_list[1]` = size of the detail coefficients at the coarsest level

...

`size_list[N]` = size of the detail coefficients at the finest level

`size_list[N+1]` = size of the original image

`N` = number of scaling levels = `nscales`

pywt_coeff_to_array

`odl.trafos.wavelet.pywt_coeff_to_array(coeff, size_list)`

Convert a Pywavelets coefficient list into a flat array.

Related to 1D, 2D and 3D multilevel discrete wavelet transforms.

Parameterscoeff : ordered list

Coefficient are organized in the list in the following way:

In 1D:

`[aN, (dN), ..., (d1)]`

The abbreviations refer to

a = approximation,

d = detail

In 2D:

```
[aaN, (adN, daN, ddN), ..., (ad1, da1, dd1)]
```

The abbreviations refer to

aa = approx. on 1st dim, approx. on 2nd dim (approximation),

ad = approx. on 1st dim, detail on 2nd dim (horizontal),

da = detail on 1st dim, approx. on 2nd dim (vertical),

dd = detail on 1st dim, detail on 2nd dim (diagonal),

In 3D:

```
[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ...  
(aad1, ada1, add1, daa1, dad1, dda1, ddd1)]
```

The abbreviations refer to

aaa = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

aad = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

ada = approx. on 1st dim, detail on 3rd dim, approx. on 3rd dim,

add = approx. on 1st dim, detail on 3rd dim, detail on 3rd dim,

daa = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

dad = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

dda = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

ddd = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

N refers to the number of scaling levels

size_list : list

A list containing the sizes of the wavelet (approximation and detail) coefficients at different scaling levels.

size_list[0] = size of approximation coefficients at the coarsest level,

size_list[1] = size of the detailed coefficients at the coarsest level,

size_list[N] = size of the detailed coefficients at the finest level,

size_list[N+1] = size of original image,

N = the number of scaling levels

Returns : `numpy.ndarray`

Flattened and concatenated coefficient array The length of the array depends on the size of input image to be transformed and on the chosen wavelet basis.

wavelet_decomposition3d

`odl.trafos.wavelet.wavelet_decomposition3d(x, wbasis, mode, nscales)`

Discrete 3D multiresolution wavelet decomposition.

Compute the discrete 3D multiresolution wavelet decomposition at the given level (nscales) for a given 3D image. Utilizes a [n-dimensional PyWavelet](#) function `pywt.dwtN`.

Parameters`x : DiscreteLpVector`

wbasis : `pywt.Wavelet`

Selected wavelet basis. For more information see the [PyWavelets documentation on wavelet bases](#).

mode : `str`

Signal extension mode. For possible extensions see the [Pywavelets documentation on signal extension modes](#).

n scales : `int`

Number of scales in the coefficient list.

Returns`coeff_list : list`

A list of coefficient organized in the following way ``[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ... (aad1, ada1, add1, daa1, dad1, dda1, ddd1)]``.

The abbreviations refer to

aaa = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

aad = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

ada = approx. on 1st dim, detail on 3rd dim, approx. on 3rd dim,

add = approx. on 1st dim, detail on 3rd dim, detail on 3rd dim,

daa = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

dad = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

dda = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

ddd = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

N = the number of scaling levels

wavelet_reconstruction3d

`odl.trafos.wavelet.wavelet_reconstruction3d(coeff_list, wbasis, mode, nscales)`

Discrete 3D multiresolution wavelet reconstruction

Compute a discrete 3D multiresolution wavelet reconstruction from a given wavelet coefficient list. Utilizes a [PyWavelet](#) function `pywt.dwtN`

Parameters`coeff_list : list`

A list of wavelet approximation and detail coefficients organized in the following way ``[caaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ... (aad1, ada1, add1, daa1, dad1, dda1, ddd1)]``.

The abbreviations refer to

aaa = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

aad = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

ada = approx. on 1st dim, detail on 3rd dim, approx. on 3rd dim,

add = approx. on 1st dim, detail on 3rd dim, detail on 3rd dim,

daa = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

dad = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

dda = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

ddd = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

N = the number of scaling levels

wbasis : `_pywt.Wavelet`

Describes properties of a selected wavelet basis. For more information see [PyWavelet documentation](#)

mode : `str`

Signal extension mode. For possible extensions see the [signal extension modes](#) of Py-Wavelets.

n scales : `int`

Number of scales in the coefficient list.

Returns : `numpy.ndarray`.

A wavelet reconstruction.

2.8.9 util

Utility library for ODL, only for internal use.

Modules

graphics

Functions for graphical output.

Functions

`show_discrete_data(values, grid[, title, ...])` Display a discrete 1d or 2d function.

show_discrete_data

`odl.util.graphics.show_discrete_data(values, grid, title=None, method='', show=False, fig=None, **kwargs)`

Display a discrete 1d or 2d function.

Parameters**values** : `numpy.ndarray`

The values to visualize

grid : `RegularGrid`

Grid of the values

title : `str`, optional

Set the title of the figure

method : str, optional

1d methods:

‘plot’ : graph plot

‘scatter’ : scattered 2d points (2nd axis <-> value)

2d methods:

‘imshow’ : image plot with coloring according to value, including a colorbar.

‘scatter’ : cloud of scattered 3d points (3rd axis <-> value)

‘wireframe’, ‘plot_wireframe’ : surface plot

show : bool, optional

If the plot should be showed now or deferred until later

fig : matplotlib.figure.Figure

The figure to show in. Expected to be of same “style”, as the figure given by this function. The most common usecase is that fig is the return value from an earlier call to this function.

interp : {‘nearest’, ‘linear’}

Interpolation method to use.

axis_labels : str

Axis labels, default: [‘x’, ‘y’]

kwargs : {‘figsize’, ‘saveto’, ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

Returns**fig** : matplotlib.figure.Figure

The resulting figure. It is also shown to the user.

colorbar : matplotlib.colorbar.Colorbar

The colorbar

See also:

`matplotlib.pyplot.plot`Show graph plot

`matplotlib.pyplot.imshow`Show data as image

`matplotlib.pyplot.scatter`Show scattered 3d points

numerics

Numerical helper functions for convenience or speed.

Functions

<code>apply_on_boundary</code> (array, func[, only_once, ...])	Apply a function of the boundary of an n-dimensional array.
<code>fast_1d_tensor_mult</code> (ndarr, onedim_arrs[, ...])	Fast multiplication of an n-dim array with an outer product.

apply_on_boundary

`odl.util.numerics.apply_on_boundary(array, func, only_once=True, which_boundaries=None, axis_order=None, out=None)`

Apply a function of the boundary of an n-dimensional array.

All other values are preserved as is.

Parameters
array : array-like

Modify the boundary of this array

func : callable or sequence

If a single function is given, assign `array[slice] = func(array[slice])` on the boundary slices, e.g. use `lambda x: x / 2` to divide values by 2. A sequence of functions is applied per axis separately. It must have length `array.ndim` and may consist of one function or a 2-tuple of functions per axis. `None` entries in a sequence cause the axis (side) to be skipped.

only_once : bool, optional

If `True`, ensure that each boundary point appears in exactly one slice. If `func` is a list of functions, the `axis_order` determines which functions are applied to nodes which appear in multiple slices, according to the principle “first-come, first-served”.

which_boundaries : sequence, optional

If provided, this sequence determines per axis whether to apply the function at the boundaries in each axis. The entry in each axis may consist in a single bool or a 2-tuple of bool. In the latter case, the first tuple entry decides for the left, the second for the right boundary. The length of the sequence must be `array.ndim`. `None` is interpreted as ‘all boundaries’.

axis_order : sequence of int, optional

Permutation of `range(array.ndim)` defining the order in which to process the axes. If combined with `only_once` and a function list, this determines which function is evaluated in the points that are potentially processed multiple times.

out : `numpy.ndarray`, optional

Location in which to store the result, can be the same as `array`. Default: copy of `array`

Examples

```
>>> import numpy as np
>>> arr = np.ones((3, 3))
>>> apply_on_boundary(arr, lambda x: x / 2)
array([[ 0.5,  0.5,  0.5],
       [ 0.5,  1. ,  0.5],
       [ 0.5,  0.5,  0.5]])
```

If called with `only_once=False`, applies function repeatedly

```
>>> apply_on_boundary(arr, lambda x: x / 2, only_once=False)
array([[ 0.25,  0.5 ,  0.25],
       [ 0.5 ,  1. ,  0.5 ],
       [ 0.25,  0.5 ,  0.25]])
```

```
>>> apply_on_boundary(arr, lambda x: x / 2, only_once=True,
...                    which_boundaries=((True, False), True))
array([[ 0.5,  0.5,  0.5],
       [ 0.5,  1. ,  0.5],
       [ 0.5,  1. ,  0.5]])
```

Also accepts out parameter:

```
>>> out = np.empty_like(arr)
>>> result = apply_on_boundary(arr, lambda x: x / 2, out=out)
>>> result
array([[ 0.5,  0.5,  0.5],
       [ 0.5,  1. ,  0.5],
       [ 0.5,  0.5,  0.5]])
>>> result is out
True
```

fast_1d_tensor_mult

`odl.util.numerics.fast_1d_tensor_mult(ndarr, onedim_arrs, axes=None, out=None)`

Fast multiplication of an n-dim array with an outer product.

This method implements the multiplication of an n-dimensional array with an outer product of one-dimensional arrays, e.g.:

```
a = np.ones((10, 10, 10))
x = np.random.rand(10)
a *= x[:, None, None] * x[None, :, None] * x[None, None, :]
```

Basically, there are two ways to do such an operation:

- 1.First calculate the factor on the right-hand side and do one “big” multiplication; or
- 2.Multiply by one factor at a time.

The procedure of building up the large factor in the first method is relatively cheap if the number of 1d arrays is smaller than the number of dimensions. For exactly n vectors, the second method is faster, although it loops over the array a n times.

This implementation combines the two ideas into a hybrid scheme:

- If there are less 1d arrays than dimensions, choose 1.
- Otherwise, calculate the factor array for n-1 arrays and multiply it to the large array. Finally, multiply with the last 1d array.

The advantage of this approach is that it is memory-friendly and loops over the big array only twice.

Parameters`ndarr` : *array-like*

Array to multiply to

onedim_arrs : sequence of array-like

One-dimensional arrays to be multiplied with `ndarr`. The sequence may not be longer than `ndarr.ndim`.

axes : sequence of `int`, optional

Take the 1d transform along these axes. `None` corresponds to the last `len(onedim_arrs)` axes, in ascending order.

out : `numpy.ndarray`, optional

Array in which the result is stored

Returns**out** : `numpy.ndarray`

Result of the modification. If `out` was given, the returned object is a reference to it.

phantom

Some useful phantoms, mostly for tomography tests.

Functions

<code>cuboid</code> (discr_space, begin, end)	Rectangular cuboid.
<code>derenzo_sources</code> (space)	Create the PET/SPECT Derenzo sources phantom.
<code>ellipse_phantom_2d</code> (space, ellipses)	Create an ellipse phantom in 2d space.
<code>ellipse_phantom_3d</code> (space, ellipses)	Create an ellipse phantom in 3d space.
<code>indicate_proj_axis</code> (discr_space[, ...])	Phantom indicating along which axis it is projected.
<code>phantom</code> (space, ellipses)	Return a phantom given by ellipses.
<code>shepp_logan</code> (space[, modified])	Create a Shepp-Logan phantom.
<code>submarine_phantom</code> (discr[, smooth, taper])	Return a ‘submarine’ phantom consisting in an ellipsoid and a box.

cuboid

`odl.util.phantom.cuboid`(discr_space, begin, end)

Rectangular cuboid.

Parameters**discr_space** : *Discretization*

Discretized space in which the phantom is supposed to be created

begin : array-like or float in [0, 1]

The lower left corner of the cuboid within the space grid relative to the extend of the grid

end : array-like or float in [0, 1]

The upper right corner of the cuboid within the space grid relative to the extend of the grid

Returns**phantom** : *LinearSpaceVector*

Returns an element in `discr_space`

Examples

```
>>> import odl
>>> space = odl.uniform_discr(0, 1, 6, dtype='float32')
>>> print(cuboid(space, 0.5, 1))
[0.0, 0.0, 0.0, 1.0, 1.0, 1.0]
>>> space = odl.uniform_discr([0, 0], [1, 1], [4, 6], dtype='float32')
>>> print(cuboid(space, [0.25, 0], [0.75, 0.5]))
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
```

```
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0],  
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0],  
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
```

derenzo_sources

`odl.util.phantom.derenzo_sources(space)`

Create the PET/SPECT Derenzo sources phantom.

The Derenzo phantom contains a series of circles of decreasing size.

In 3d the phantom is simply the 2d phantom extended in the z direction as cylinders.

ellipse_phantom_2d

`odl.util.phantom.ellipse_phantom_2d(space, ellipses)`

Create an ellipse phantom in 2d space.

Parameters**space** : *DiscreteLp*

The space the phantom should be generated in.

ellipses : list of lists

Each row should contain: 'value', 'axis_1', 'axis_2', 'center_x', 'center_y', 'rotation'

The ellipses should be contained the rectangle [-1, -1] x [1, 1].

Returns**phantom** : *DiscreteLpVector*

The phantom

ellipse_phantom_3d

`odl.util.phantom.ellipse_phantom_3d(space, ellipses)`

Create an ellipse phantom in 3d space.

Parameters**space** : *DiscreteLp*

The space the phantom should be generated in.

ellipses : list of lists

Each row should contain: 'value', 'axis_1', 'axis_2', 'axis_2', 'center_x', 'center_y', 'center_z', 'rotation_phi', 'rotation_theta', 'rotation_psi' The ellipses should be contained the rectangle [-1, -1, -1] x [1, 1, 1].

Returns**phantom** : *DiscreteLpVector*

The phantom

indicate_proj_axis

`odl.util.phantom.indicate_proj_axis(discr_space, scale_structures=0.5)`

Phantom indicating along which axis it is projected.

The number (n) of rectangles in a parallel-beam projection along a main axis (0, 1, or 2) indicates the projection to be along the (n-1)the dimension.

Parameters`discr_space` : *Discretization*

Discretized space in which the phantom is supposed to be created

`scale_structures` : positive float in (0, 1]

Scales objects (cube, cuboids)

Returns`phantom` : *LinearSpaceVector*

Returns an element in `discr_space`

Examples

```
>>> import odl
>>> space = odl.uniform_discr([0] * 3, [1] * 3, [8, 8, 8])
>>> phan = indicate_proj_axis(space).asarray()
>>> print(np.sum(phan, 0))
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  3.  3.  0.  0.  0.]
 [ 0.  0.  0.  3.  3.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
>>> print(np.sum(phan, 1))
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  2.  2.  0.  0.  0.]
 [ 0.  0.  0.  2.  2.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
>>> print(np.sum(phan, 2))
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  2.  2.  0.  0.  0.]
 [ 0.  0.  0.  2.  2.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  2.  0.  0.  0.]
 [ 0.  0.  0.  2.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

phantom

`odl.util.phantom.phantom(space, ellipses)`

Return a phantom given by ellipses.

shepp_logan

`odl.util.phantom.shepp_logan(space, modified=False)`

Create a Shepp-Logan phantom.

The standard Shepp-Logan phantom in 2 or 3 dimensions.

References

Wikipedia : https://en.wikipedia.org/wiki/Shepp%E2%80%93Logan_phantom

submarine_phantom

`odl.util.phantom.submarine_phantom(discr, smooth=True, taper=20.0)`

Return a ‘submarine’ phantom consisting in an ellipsoid and a box.

This phantom is used in [Okt2015] for shape-based reconstruction.

Parameters`discr` : *DiscreteLp*

Discretized space in which the phantom is supposed to be created

smooth : `bool`, optional

If `True`, the boundaries are smoothed out. Otherwise, the function steps from 0 to 1 at the boundaries.

taper : `float`, optional

Tapering parameter for the boundary smoothing. Larger values mean faster taper, i.e. sharper boundaries.

Returns`phantom` : *DiscreteLpVector*

testutils

Utilities for internal use.

Classes

<i>FailCounter</i> (<code>[err_msg]</code>)	Used to count the number of failures of something
<i>ProgressBar</i> (<code>[text]</code>)	A simple command-line progress bar.
<i>ProgressRange</i> (<code>text, n</code>)	Simple range sequence with progress bar output.
<i>Timer</i> (<code>[name]</code>)	A timer context manager.

FailCounter

class `odl.util.testutils.FailCounter` (`err_msg=None`)

Bases: `object`

Used to count the number of failures of something

Usage:

```
with FailCounter() as counter:
    # Do stuff

    counter.fail()
```

When done, it prints

```
*** FAILED 1 TEST CASE(S) ***
```

Methods

<code>__eq__</code>	Return self==value.
<code>fail([string])</code>	Add failure with reason as string.

FailCounter.fail

`FailCounter.fail` (*string=None*)
 Add failure with reason as string.
`__init__` (*err_msg=None*)

ProgressBar

`class odl.util.testutils.ProgressBar` (*text='progress', *njobs*)

Bases: object

A simple command-line progress bar.

Usage:

```
>>> progress = ProgressBar('Reading data', 10)
```

Reading data: [] Starting

```
>>> progress.update(4) #halfway, zero indexing
```

Reading data: [#####] 50.0%

Multi-indices, from slowest to fastest:

```
>>> progress = ProgressBar('Reading data', 10, 10)
```

Reading data: [] Starting

```
>>> progress.update(9, 8)
```

Reading data: [#####] 99.0%

Supports simply calling update, which moves the counter forward:

```
>>> progress = ProgressBar('Reading data', 10, 10)
```

Reading data: [] Starting

```
>>> progress.update()
```

Reading data: [] 1.0%

Methods

<code>__eq__</code>	Return self==value.
<code>start()</code>	Print the initial bar.
<code>update(*indices)</code>	Update the bar according to indices.

ProgressBar.start

`ProgressBar.start()`
Print the initial bar.

ProgressBar.update

`ProgressBar.update(*indices)`
Update the bar according to indices.
`__init__(text='progress', *njobs)`
Initialize a new instance.

ProgressRange

`class odl.util.testutils.ProgressRange(text, n)`
Bases: object
Simple range sequence with progress bar output.

Methods

`__eq__` Return self==value.

`__init__(text, n)`
Initialize a new instance.

Timer

`class odl.util.testutils.Timer(name=None)`
Bases: object
A timer context manager.
Usage:

```
with Timer('name'):  
    # Do stuff
```

Prints the time stuff took to execute.

Methods

`__eq__` Return self==value.

`__init__(name=None)`

Functions

`all_almost_equal(iter1, iter2[, places])` True if all elements in a and b are almost equal.

Continued on next page

Table 2.300 – continued from previous page

<code>all_almost_equal_array(v1, v2, places)</code>	
<code>all_equal(iter1, iter2)</code>	True if all elements in a and b are equal.
<code>almost_equal(a, b[, places])</code>	True if scalars a and b are almost equal.
<code>is_subdict(subdict, dictionary)</code>	True if all items of subdict are in dictionary.
<code>skip_if_no_benchmark(function)</code>	Trivial decorator used if pytest marks are not available.
<code>skip_if_no_cuda(function)</code>	Trivial decorator used if pytest marks are not available.
<code>skip_if_no_largescale(function)</code>	Trivial decorator used if pytest marks are not available.
<code>skip_if_no_pyfftw(function)</code>	Trivial decorator used if pytest marks are not available.
<code>skip_if_no_pywavelets(function)</code>	Trivial decorator used if pytest marks are not available.
<code>timeit(arg)</code>	A timer decorator.

all_almost_equal

`odl.util.testutils.all_almost_equal(iter1, iter2, places=None)`
 True if all elements in a and b are almost equal.

all_almost_equal_array

`odl.util.testutils.all_almost_equal_array(v1, v2, places)`

all_equal

`odl.util.testutils.all_equal(iter1, iter2)`
 True if all elements in a and b are equal.

almost_equal

`odl.util.testutils.almost_equal(a, b, places=None)`
 True if scalars a and b are almost equal.

is_subdict

`odl.util.testutils.is_subdict(subdict, dictionary)`
 True if all items of subdict are in dictionary.

skip_if_no_benchmark

`odl.util.testutils.skip_if_no_benchmark(function)`
 Trivial decorator used if pytest marks are not available.

skip_if_no_cuda

`odl.util.testutils.skip_if_no_cuda(function)`
 Trivial decorator used if pytest marks are not available.

skip_if_no_largescale

`odl.util.testutils.skip_if_no_largescale` (*function*)
 Trivial decorator used if pytest marks are not available.

skip_if_no_pyfftw

`odl.util.testutils.skip_if_no_pyfftw` (*function*)
 Trivial decorator used if pytest marks are not available.

skip_if_no_pywavelets

`odl.util.testutils.skip_if_no_pywavelets` (*function*)
 Trivial decorator used if pytest marks are not available.

timeit

`odl.util.testutils.timeit` (*arg*)
 A timer decorator.

Usage:

```
@timeit
def myfunction(...):
    ...

@timeit('info string')
def myfunction(...):
    ...
```

ufuncs

UFuncs for ODL vectors.

These functions are internal and should only be used as methods on *NtuplesBaseVector* type spaces.

See `numpy.ufuncs` for more information.

Notes

The default implementation of these methods make heavy use of the `NtuplesBaseVector.__array__` to extract a `numpy.ndarray` from the vector, and then apply a ufunc to it. Afterwards, `NtuplesBaseVector.__array_wrap__` is used to re-wrap the data into the appropriate space.

Classes

<code>CudaNtuplesUFuncs(vector)</code>	UFuncs for <i>CudaNtuplesVector</i> objects.
<code>DiscreteLpUFuncs(vector)</code>	UFuncs for <i>DiscreteLpVector</i> objects.
<code>NtuplesBaseUFuncs(vector)</code>	UFuncs for <i>NtuplesBaseVector</i> objects.

Continued on next page

Table 2.301 – continued from previous page

<i>NtuplesUFuncs</i> (vector)	UFuncs for <i>NtuplesVector</i> objects.
<i>ProductSpaceUFuncs</i> (vector)	UFuncs for <i>ProductSpaceVector</i> objects.

CudaNtuplesUFuncs

class odl.util.ufuncs.**CudaNtuplesUFuncs** (vector)

Bases: odl.util.ufuncs.NtuplesBaseUFuncs

UFuncs for *CudaNtuplesVector* objects.

Internal object, should not be created except in *CudaNtuplesVector*.

Methods

<code>__eq__</code>	Return self==value.
<code>absolute([out])</code>	Calculate the absolute value element-wise.
<code>add(x2[, out])</code>	Add arguments element-wise.
<code>arccos([out])</code>	Trigonometric inverse cosine, element-wise.
<code>arccosh([out])</code>	Inverse hyperbolic cosine, element-wise.
<code>arcsin([out])</code>	Inverse sine, element-wise.
<code>arcsinh([out])</code>	Inverse hyperbolic sine element-wise.
<code>arctan([out])</code>	Trigonometric inverse tangent, element-wise.
<code>arctan2(x2[, out])</code>	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
<code>arctanh([out])</code>	Inverse hyperbolic tangent element-wise.
<code>bitwise_and(x2[, out])</code>	Compute the bit-wise AND of two arrays element-wise.
<code>bitwise_or(x2[, out])</code>	Compute the bit-wise OR of two arrays element-wise.
<code>bitwise_xor(x2[, out])</code>	Compute the bit-wise XOR of two arrays element-wise.
<code>ceil([out])</code>	Return the ceiling of the input, element-wise.
<code>conj([out])</code>	Return the complex conjugate, element-wise.
<code>copysign(x2[, out])</code>	Change the sign of x1 to that of x2, element-wise.
<code>cos([out])</code>	Cosine element-wise.
<code>cosh([out])</code>	Hyperbolic cosine, element-wise.
<code>deg2rad([out])</code>	Convert angles from degrees to radians.
<code>divide(x2[, out])</code>	Returns a true division of the inputs, element-wise.
<code>equal(x2[, out])</code>	Return (x1 == x2) element-wise.
<code>exp([out])</code>	Calculate the exponential of all elements in the input array.
<code>exp2([out])</code>	Calculate 2^{**p} for all p in the input array.
<code>expm1([out])</code>	Calculate $\exp(x) - 1$ for all elements in the array.
<code>floor([out])</code>	Return the floor of the input, element-wise.
<code>floor_divide(x2[, out])</code>	Return the largest integer smaller or equal to the division of the
<code>fmax(x2[, out])</code>	Element-wise maximum of array elements.
<code>fmin(x2[, out])</code>	Element-wise minimum of array elements.
<code>fmod(x2[, out])</code>	Return the element-wise remainder of division.
<code>greater(x2[, out])</code>	Return the truth value of (x1 > x2) element-wise.
<code>greater_equal(x2[, out])</code>	Return the truth value of (x1 >= x2) element-wise.
<code>hypot(x2[, out])</code>	Given the “legs” of a right triangle, return its hypotenuse.
<code>invert([out])</code>	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<code>isfinite([out])</code>	Test element-wise for finiteness (not infinity or not Not a Number).
<code>isinf([out])</code>	Test element-wise for positive or negative infinity.
<code>isnan([out])</code>	Test element-wise for NaN and return result as a boolean array.

Continued on next page

Table 2.302 – continued from previous page

<code>left_shift(x2[, out])</code>	Shift the bits of an integer to the left.
<code>less(x2[, out])</code>	Return the truth value of $(x1 < x2)$ element-wise.
<code>less_equal(x2[, out])</code>	Return the truth value of $(x1 \leq x2)$ element-wise.
<code>log([out])</code>	Natural logarithm, element-wise.
<code>log10([out])</code>	Return the base 10 logarithm of the input array, element-wise.
<code>log1p([out])</code>	Return the natural logarithm of one plus the input array, element-wise.
<code>log2([out])</code>	Base-2 logarithm of x .
<code>logaddexp(x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs.
<code>logaddexp2(x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs in base-2.
<code>logical_and(x2[, out])</code>	Compute the truth value of $x1$ AND $x2$ element-wise.
<code>logical_not([out])</code>	Compute the truth value of NOT x element-wise.
<code>logical_or(x2[, out])</code>	Compute the truth value of $x1$ OR $x2$ element-wise.
<code>logical_xor(x2[, out])</code>	Compute the truth value of $x1$ XOR $x2$, element-wise.
<code>max()</code>	Maximum value in array.
<code>maximum(x2[, out])</code>	Element-wise maximum of array elements.
<code>min()</code>	Minimum value in array.
<code>minimum(x2[, out])</code>	Element-wise minimum of array elements.
<code>mod(x2[, out])</code>	Return element-wise remainder of division.
<code>modf([out1, out2])</code>	Return the fractional and integral parts of an array, element-wise.
<code>multiply(x2[, out])</code>	Multiply arguments element-wise.
<code>negative([out])</code>	Numerical negative, element-wise.
<code>not_equal(x2[, out])</code>	Return $(x1 \neq x2)$ element-wise.
<code>power(x2[, out])</code>	First array elements raised to powers from second array, element-wise.
<code>prod()</code>	Product of array elements.
<code>rad2deg([out])</code>	Convert angles from radians to degrees.
<code>reciprocal([out])</code>	Return the reciprocal of the argument, element-wise.
<code>remainder(x2[, out])</code>	Return element-wise remainder of division.
<code>right_shift(x2[, out])</code>	Shift the bits of an integer to the right.
<code>rint([out])</code>	Round elements of the array to the nearest integer.
<code>sign([out])</code>	Returns an element-wise indication of the sign of a number.
<code>signbit([out])</code>	Returns element-wise True where signbit is set (less than zero).
<code>sin([out])</code>	Trigonometric sine, element-wise.
<code>sinh([out])</code>	Hyperbolic sine, element-wise.
<code>sqrt([out])</code>	Return the positive square-root of an array, element-wise.
<code>square([out])</code>	Return the element-wise square of the input.
<code>subtract(x2[, out])</code>	Subtract arguments, element-wise.
<code>sum()</code>	Sum of array elements.
<code>tan([out])</code>	Compute tangent element-wise.
<code>tanh([out])</code>	Compute hyperbolic tangent element-wise.
<code>true_divide(x2[, out])</code>	Returns a true division of the inputs, element-wise.
<code>trunc([out])</code>	Return the truncated value of the input, element-wise.

CudaNtuplesUFuncs.absolute

`CudaNtuplesUFuncs.absolute` (*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

CudaNtuplesUFuncs.add

`CudaNtuplesUFuncs.add(x2, out=None)`

Add arguments element-wise.

See also:

`numpy.add`

CudaNtuplesUFuncs.arccos

`CudaNtuplesUFuncs.arccos(out=None)`

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

CudaNtuplesUFuncs.arccosh

`CudaNtuplesUFuncs.arccosh(out=None)`

Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

CudaNtuplesUFuncs.arcsin

`CudaNtuplesUFuncs.arcsin(out=None)`

Inverse sine, element-wise.

See also:

`numpy.arcsin`

CudaNtuplesUFuncs.arcsinh

`CudaNtuplesUFuncs.arcsinh(out=None)`

Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

CudaNtuplesUFuncs.arctan

`CudaNtuplesUFuncs.arctan(out=None)`

Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

CudaNtuplesUFuncs.arctan2

`CudaNtuplesUFuncs.arctan2(x2, out=None)`

Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

See also:

`numpy.arctan2`

CudaNtuplesUFuncs.arctanh

CudaNtuplesUFuncs.**arctanh** (*out=None*)

Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

CudaNtuplesUFuncs.bitwise_and

CudaNtuplesUFuncs.**bitwise_and** (*x2, out=None*)

Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

CudaNtuplesUFuncs.bitwise_or

CudaNtuplesUFuncs.**bitwise_or** (*x2, out=None*)

Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

CudaNtuplesUFuncs.bitwise_xor

CudaNtuplesUFuncs.**bitwise_xor** (*x2, out=None*)

Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

CudaNtuplesUFuncs.ceil

CudaNtuplesUFuncs.**ceil** (*out=None*)

Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

CudaNtuplesUFuncs.conj

CudaNtuplesUFuncs.**conj** (*out=None*)

Return the complex conjugate, element-wise.

See also:

`numpy.conj`

CudaNtuplesUFuncs.copysign

CudaNtuplesUFuncs.**copysign** (*x2, out=None*)

Change the sign of *x1* to that of *x2*, element-wise.

See also:

`numpy.copysign`

CudaNtuplesUFuncs.cos

CudaNtuplesUFuncs.**cos** (*out=None*)

Cosine element-wise.

See also:

`numpy.cos`

CudaNtuplesUFuncs.cosh

CudaNtuplesUFuncs.**cosh** (*out=None*)

Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

CudaNtuplesUFuncs.deg2rad

CudaNtuplesUFuncs.**deg2rad** (*out=None*)

Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

CudaNtuplesUFuncs.divide

CudaNtuplesUFuncs.**divide** (*x2, out=None*)

Returns a true division of the inputs, element-wise.

See also:

`numpy.divide`

CudaNtuplesUFuncs.equal

CudaNtuplesUFuncs.**equal** (*x2, out=None*)

Return $(x1 == x2)$ element-wise.

See also:

`numpy.equal`

CudaNtuplesUFuncs.exp

CudaNtuplesUFuncs.**exp** (*out=None*)

Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

CudaNtuplesUFuncs.exp2

CudaNtuplesUFuncs.**exp2** (*out=None*)

Calculate 2^{*p} for all p in the input array.

See also:

`numpy.exp2`

CudaNtuplesUFuncs.expm1

`CudaNtuplesUFuncs.expm1` (*out=None*)

Calculate $\exp(x) - 1$ for all elements in the array.

See also:

`numpy.expm1`

CudaNtuplesUFuncs.floor

`CudaNtuplesUFuncs.floor` (*out=None*)

Return the floor of the input, element-wise.

See also:

`numpy.floor`

CudaNtuplesUFuncs.floor_divide

`CudaNtuplesUFuncs.floor_divide` (*x2, out=None*)

Return the largest integer smaller or equal to the division of the

See also:

`numpy.floor_divide`

CudaNtuplesUFuncs.fmax

`CudaNtuplesUFuncs.fmax` (*x2, out=None*)

Element-wise maximum of array elements.

See also:

`numpy.fmax`

CudaNtuplesUFuncs.fmin

`CudaNtuplesUFuncs.fmin` (*x2, out=None*)

Element-wise minimum of array elements.

See also:

`numpy.fmin`

CudaNtuplesUFuncs.fmod

`CudaNtuplesUFuncs.fmod` (*x2, out=None*)

Return the element-wise remainder of division.

See also:

`numpy.fmod`

CudaNtuplesUFuncs.greater

`CudaNtuplesUFuncs.greater` (*x2, out=None*)

Return the truth value of ($x1 > x2$) element-wise.

See also:

`numpy.greater`

CudaNtuplesUFuncs.greater_equal

`CudaNtuplesUFuncs.greater_equal` (*x2*, *out=None*)

Return the truth value of (*x1* >= *x2*) element-wise.

See also:

`numpy.greater_equal`

CudaNtuplesUFuncs.hypot

`CudaNtuplesUFuncs.hypot` (*x2*, *out=None*)

Given the “legs” of a right triangle, return its hypotenuse.

See also:

`numpy.hypot`

CudaNtuplesUFuncs.invert

`CudaNtuplesUFuncs.invert` (*out=None*)

Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

`numpy.invert`

CudaNtuplesUFuncs.isfinite

`CudaNtuplesUFuncs.isfinite` (*out=None*)

Test element-wise for finiteness (not infinity or not Not a Number).

See also:

`numpy.isfinite`

CudaNtuplesUFuncs.isinf

`CudaNtuplesUFuncs.isinf` (*out=None*)

Test element-wise for positive or negative infinity.

See also:

`numpy.isinf`

CudaNtuplesUFuncs.isnan

`CudaNtuplesUFuncs.isnan` (*out=None*)

Test element-wise for NaN and return result as a boolean array.

See also:

`numpy.isnan`

CudaNtuplesUFuncs.left_shift

`CudaNtuplesUFuncs.left_shift` (*x2*, *out=None*)

Shift the bits of an integer to the left.

See also:

`numpy.left_shift`

CudaNtuplesUFuncs.less

`CudaNtuplesUFuncs.less(x2, out=None)`

Return the truth value of $(x1 < x2)$ element-wise.

See also:

`numpy.less`

CudaNtuplesUFuncs.less_equal

`CudaNtuplesUFuncs.less_equal(x2, out=None)`

Return the truth value of $(x1 \leq x2)$ element-wise.

See also:

`numpy.less_equal`

CudaNtuplesUFuncs.log

`CudaNtuplesUFuncs.log(out=None)`

Natural logarithm, element-wise.

See also:

`numpy.log`

CudaNtuplesUFuncs.log10

`CudaNtuplesUFuncs.log10(out=None)`

Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

CudaNtuplesUFuncs.log1p

`CudaNtuplesUFuncs.log1p(out=None)`

Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

CudaNtuplesUFuncs.log2

`CudaNtuplesUFuncs.log2(out=None)`

Base-2 logarithm of x .

See also:

`numpy.log2`

CudaNtuplesUFuncs.logaddexp

`CudaNtuplesUFuncs.logaddexp(x2, out=None)`

Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

CudaNtuplesUFuncs.logaddexp2

CudaNtuplesUFuncs.**logaddexp2** (*x2*, *out=None*)

Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

CudaNtuplesUFuncs.logical_and

CudaNtuplesUFuncs.**logical_and** (*x2*, *out=None*)

Compute the truth value of x1 AND x2 element-wise.

See also:

`numpy.logical_and`

CudaNtuplesUFuncs.logical_not

CudaNtuplesUFuncs.**logical_not** (*out=None*)

Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

CudaNtuplesUFuncs.logical_or

CudaNtuplesUFuncs.**logical_or** (*x2*, *out=None*)

Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

CudaNtuplesUFuncs.logical_xor

CudaNtuplesUFuncs.**logical_xor** (*x2*, *out=None*)

Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

CudaNtuplesUFuncs.max

CudaNtuplesUFuncs.**max** ()

Maximum value in array.

See also:

`numpy.amax`

CudaNtuplesUFuncs.maximum

CudaNtuplesUFuncs.**maximum** (*x2*, *out=None*)

Element-wise maximum of array elements.

See also:

`numpy.maximum`

CudaNtuplesUFuncs.min

`CudaNtuplesUFuncs.min()`
Minimum value in array.

See also:

`numpy.amin`

CudaNtuplesUFuncs.minimum

`CudaNtuplesUFuncs.minimum(x2, out=None)`
Element-wise minimum of array elements.

See also:

`numpy.minimum`

CudaNtuplesUFuncs.mod

`CudaNtuplesUFuncs.mod(x2, out=None)`
Return element-wise remainder of division.

See also:

`numpy.mod`

CudaNtuplesUFuncs.modf

`CudaNtuplesUFuncs.modf(out1=None, out2=None)`
Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

CudaNtuplesUFuncs.multiply

`CudaNtuplesUFuncs.multiply(x2, out=None)`
Multiply arguments element-wise.

See also:

`numpy.multiply`

CudaNtuplesUFuncs.negative

`CudaNtuplesUFuncs.negative(out=None)`
Numerical negative, element-wise.

See also:

`numpy.negative`

CudaNtuplesUFuncs.not_equal

`CudaNtuplesUFuncs.not_equal(x2, out=None)`
Return $(x1 \neq x2)$ element-wise.

See also:

`numpy.not_equal`

CudaNtuplesUFuncs.power

CudaNtuplesUFuncs.**power**(x2, out=None)

First array elements raised to powers from second array, element-wise.

See also:

numpy.power

CudaNtuplesUFuncs.prod

CudaNtuplesUFuncs.**prod**()

Product of array elements.

See also:

numpy.prod

CudaNtuplesUFuncs.rad2deg

CudaNtuplesUFuncs.**rad2deg**(out=None)

Convert angles from radians to degrees.

See also:

numpy.rad2deg

CudaNtuplesUFuncs.reciprocal

CudaNtuplesUFuncs.**reciprocal**(out=None)

Return the reciprocal of the argument, element-wise.

See also:

numpy.reciprocal

CudaNtuplesUFuncs.remainder

CudaNtuplesUFuncs.**remainder**(x2, out=None)

Return element-wise remainder of division.

See also:

numpy.remainder

CudaNtuplesUFuncs.right_shift

CudaNtuplesUFuncs.**right_shift**(x2, out=None)

Shift the bits of an integer to the right.

See also:

numpy.right_shift

CudaNtuplesUFuncs rint

CudaNtuplesUFuncs.**rint**(out=None)

Round elements of the array to the nearest integer.

See also:

numpy.rint

CudaNtuplesUFuncs.sign

`CudaNtuplesUFuncs.sign` (*out=None*)

Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

CudaNtuplesUFuncs.signbit

`CudaNtuplesUFuncs.signbit` (*out=None*)

Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

CudaNtuplesUFuncs.sin

`CudaNtuplesUFuncs.sin` (*out=None*)

Trigonometric sine, element-wise.

See also:

`numpy.sin`

CudaNtuplesUFuncs.sinh

`CudaNtuplesUFuncs.sinh` (*out=None*)

Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

CudaNtuplesUFuncs.sqrt

`CudaNtuplesUFuncs.sqrt` (*out=None*)

Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

CudaNtuplesUFuncs.square

`CudaNtuplesUFuncs.square` (*out=None*)

Return the element-wise square of the input.

See also:

`numpy.square`

CudaNtuplesUFuncs.subtract

`CudaNtuplesUFuncs.subtract` (*x2, out=None*)

Subtract arguments, element-wise.

See also:

`numpy.subtract`

CudaNtuplesUFuncs.sum

CudaNtuplesUFuncs.**sum**()
Sum of array elements.

See also:

`numpy.sum`

CudaNtuplesUFuncs.tan

CudaNtuplesUFuncs.**tan**(*out=None*)
Compute tangent element-wise.

See also:

`numpy.tan`

CudaNtuplesUFuncs.tanh

CudaNtuplesUFuncs.**tanh**(*out=None*)
Compute hyperbolic tangent element-wise.

See also:

`numpy.tanh`

CudaNtuplesUFuncs.true_divide

CudaNtuplesUFuncs.**true_divide**(*x2, out=None*)
Returns a true division of the inputs, element-wise.

See also:

`numpy.true_divide`

CudaNtuplesUFuncs.trunc

CudaNtuplesUFuncs.**trunc**(*out=None*)
Return the truncated value of the input, element-wise.

See also:

`numpy.trunc`

__init__(*vector*)

Create ufunc wrapper for vector.

DiscreteLpUFuncs

class odl.util.ufuncs.**DiscreteLpUFuncs**(*vector*)

Bases: `odl.util.ufuncs.NtuplesBaseUFuncs`

UFuncs for *DiscreteLpVector* objects.

Internal object, should not be created except in *DiscreteLpVector*.

Methods

<code>__eq__</code>	Return self==value.
<code>absolute</code> ([<i>out</i>])	Calculate the absolute value element-wise.
Continued on next page	

Table 2.303 – continued from previous page

<i>add</i> (x2[, out])	Add arguments element-wise.
<i>arccos</i> ([out])	Trigonometric inverse cosine, element-wise.
<i>arccosh</i> ([out])	Inverse hyperbolic cosine, element-wise.
<i>arcsin</i> ([out])	Inverse sine, element-wise.
<i>arcsinh</i> ([out])	Inverse hyperbolic sine element-wise.
<i>arctan</i> ([out])	Trigonometric inverse tangent, element-wise.
<i>arctan2</i> (x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
<i>arctanh</i> ([out])	Inverse hyperbolic tangent element-wise.
<i>bitwise_and</i> (x2[, out])	Compute the bit-wise AND of two arrays element-wise.
<i>bitwise_or</i> (x2[, out])	Compute the bit-wise OR of two arrays element-wise.
<i>bitwise_xor</i> (x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
<i>ceil</i> ([out])	Return the ceiling of the input, element-wise.
<i>conj</i> ([out])	Return the complex conjugate, element-wise.
<i>copysign</i> (x2[, out])	Change the sign of x1 to that of x2, element-wise.
<i>cos</i> ([out])	Cosine element-wise.
<i>cosh</i> ([out])	Hyperbolic cosine, element-wise.
<i>deg2rad</i> ([out])	Convert angles from degrees to radians.
<i>divide</i> (x2[, out])	Returns a true division of the inputs, element-wise.
<i>equal</i> (x2[, out])	Return (x1 == x2) element-wise.
<i>exp</i> ([out])	Calculate the exponential of all elements in the input array.
<i>exp2</i> ([out])	Calculate 2^{**p} for all p in the input array.
<i>expm1</i> ([out])	Calculate $\exp(x) - 1$ for all elements in the array.
<i>floor</i> ([out])	Return the floor of the input, element-wise.
<i>floor_divide</i> (x2[, out])	Return the largest integer smaller or equal to the division of the
<i>fmax</i> (x2[, out])	Element-wise maximum of array elements.
<i>fmin</i> (x2[, out])	Element-wise minimum of array elements.
<i>fmod</i> (x2[, out])	Return the element-wise remainder of division.
<i>greater</i> (x2[, out])	Return the truth value of (x1 > x2) element-wise.
<i>greater_equal</i> (x2[, out])	Return the truth value of (x1 >= x2) element-wise.
<i>hypot</i> (x2[, out])	Given the “legs” of a right triangle, return its hypotenuse.
<i>invert</i> ([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<i>isfinite</i> ([out])	Test element-wise for finiteness (not infinity or not Not a Number).
<i>isinf</i> ([out])	Test element-wise for positive or negative infinity.
<i>isnan</i> ([out])	Test element-wise for NaN and return result as a boolean array.
<i>left_shift</i> (x2[, out])	Shift the bits of an integer to the left.
<i>less</i> (x2[, out])	Return the truth value of (x1 < x2) element-wise.
<i>less_equal</i> (x2[, out])	Return the truth value of (x1 <= x2) element-wise.
<i>log</i> ([out])	Natural logarithm, element-wise.
<i>log10</i> ([out])	Return the base 10 logarithm of the input array, element-wise.
<i>log1p</i> ([out])	Return the natural logarithm of one plus the input array, element-wise.
<i>log2</i> ([out])	Base-2 logarithm of x.
<i>logaddexp</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs.
<i>logaddexp2</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
<i>logical_and</i> (x2[, out])	Compute the truth value of x1 AND x2 element-wise.
<i>logical_not</i> ([out])	Compute the truth value of NOT x element-wise.
<i>logical_or</i> (x2[, out])	Compute the truth value of x1 OR x2 element-wise.
<i>logical_xor</i> (x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
<i>max</i> ()	Maximum value in array.
<i>maximum</i> (x2[, out])	Element-wise maximum of array elements.
<i>min</i> ()	Minimum value in array.

Continued on next page

Table 2.303 – continued from previous page

<i>minimum</i> (x2[, out])	Element-wise minimum of array elements.
<i>mod</i> (x2[, out])	Return element-wise remainder of division.
<i>modf</i> ([out1, out2])	Return the fractional and integral parts of an array, element-wise.
<i>multiply</i> (x2[, out])	Multiply arguments element-wise.
<i>negative</i> ([out])	Numerical negative, element-wise.
<i>not_equal</i> (x2[, out])	Return (x1 != x2) element-wise.
<i>power</i> (x2[, out])	First array elements raised to powers from second array, element-wise.
<i>prod</i> ()	Product of array elements.
<i>rad2deg</i> ([out])	Convert angles from radians to degrees.
<i>reciprocal</i> ([out])	Return the reciprocal of the argument, element-wise.
<i>remainder</i> (x2[, out])	Return element-wise remainder of division.
<i>right_shift</i> (x2[, out])	Shift the bits of an integer to the right.
<i>rint</i> ([out])	Round elements of the array to the nearest integer.
<i>sign</i> ([out])	Returns an element-wise indication of the sign of a number.
<i>signbit</i> ([out])	Returns element-wise True where signbit is set (less than zero).
<i>sin</i> ([out])	Trigonometric sine, element-wise.
<i>sinh</i> ([out])	Hyperbolic sine, element-wise.
<i>sqrt</i> ([out])	Return the positive square-root of an array, element-wise.
<i>square</i> ([out])	Return the element-wise square of the input.
<i>subtract</i> (x2[, out])	Subtract arguments, element-wise.
<i>sum</i> ()	Sum of array elements.
<i>tan</i> ([out])	Compute tangent element-wise.
<i>tanh</i> ([out])	Compute hyperbolic tangent element-wise.
<i>true_divide</i> (x2[, out])	Returns a true division of the inputs, element-wise.
<i>trunc</i> ([out])	Return the truncated value of the input, element-wise.

DiscreteLpUFuncs.absolute

`DiscreteLpUFuncs.absolute` (*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

DiscreteLpUFuncs.add

`DiscreteLpUFuncs.add` (*x2, out=None*)

Add arguments element-wise.

See also:

`numpy.add`

DiscreteLpUFuncs.arccos

`DiscreteLpUFuncs.arccos` (*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

DiscreteLpUFuncs.arccosh

`DiscreteLpUFuncs.arccosh` (*out=None*)
Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

DiscreteLpUFuncs.arcsin

`DiscreteLpUFuncs.arcsin` (*out=None*)
Inverse sine, element-wise.

See also:

`numpy.arcsin`

DiscreteLpUFuncs.arcsinh

`DiscreteLpUFuncs.arcsinh` (*out=None*)
Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

DiscreteLpUFuncs.arctan

`DiscreteLpUFuncs.arctan` (*out=None*)
Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

DiscreteLpUFuncs.arctan2

`DiscreteLpUFuncs.arctan2` (*x2, out=None*)
Element-wise arc tangent of x_1/x_2 choosing the quadrant correctly.

See also:

`numpy.arctan2`

DiscreteLpUFuncs.arctanh

`DiscreteLpUFuncs.arctanh` (*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

DiscreteLpUFuncs.bitwise_and

`DiscreteLpUFuncs.bitwise_and` (*x2, out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

DiscreteLpUFuncs.bitwise_or

`DiscreteLpUFuncs.bitwise_or` (*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

DiscreteLpUFuncs.bitwise_xor

`DiscreteLpUFuncs.bitwise_xor` (*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

DiscreteLpUFuncs.ceil

`DiscreteLpUFuncs.ceil` (*out=None*)
Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

DiscreteLpUFuncs.conj

`DiscreteLpUFuncs.conj` (*out=None*)
Return the complex conjugate, element-wise.

See also:

`numpy.conj`

DiscreteLpUFuncs.copysign

`DiscreteLpUFuncs.copysign` (*x2*, *out=None*)
Change the sign of *x1* to that of *x2*, element-wise.

See also:

`numpy.copysign`

DiscreteLpUFuncs.cos

`DiscreteLpUFuncs.cos` (*out=None*)
Cosine element-wise.

See also:

`numpy.cos`

DiscreteLpUFuncs.cosh

`DiscreteLpUFuncs.cosh` (*out=None*)
Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

DiscreteLpUFuncs.deg2rad

`DiscreteLpUFuncs.deg2rad(out=None)`

Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

DiscreteLpUFuncs.divide

`DiscreteLpUFuncs.divide(x2, out=None)`

Returns a true division of the inputs, element-wise.

See also:

`numpy.divide`

DiscreteLpUFuncs.equal

`DiscreteLpUFuncs.equal(x2, out=None)`

Return $(x1 == x2)$ element-wise.

See also:

`numpy.equal`

DiscreteLpUFuncs.exp

`DiscreteLpUFuncs.exp(out=None)`

Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

DiscreteLpUFuncs.exp2

`DiscreteLpUFuncs.exp2(out=None)`

Calculate 2^{**p} for all p in the input array.

See also:

`numpy.exp2`

DiscreteLpUFuncs.expm1

`DiscreteLpUFuncs.expm1(out=None)`

Calculate $\exp(x) - 1$ for all elements in the array.

See also:

`numpy.expm1`

DiscreteLpUFuncs.floor

`DiscreteLpUFuncs.floor(out=None)`

Return the floor of the input, element-wise.

See also:

`numpy.floor`

DiscreteLpUFuncs.floor_divide

`DiscreteLpUFuncs.floor_divide(x2, out=None)`

Return the largest integer smaller or equal to the division of the

See also:

`numpy.floor_divide`

DiscreteLpUFuncs.fmax

`DiscreteLpUFuncs.fmax(x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.fmax`

DiscreteLpUFuncs.fmin

`DiscreteLpUFuncs.fmin(x2, out=None)`

Element-wise minimum of array elements.

See also:

`numpy.fmin`

DiscreteLpUFuncs.fmod

`DiscreteLpUFuncs.fmod(x2, out=None)`

Return the element-wise remainder of division.

See also:

`numpy.fmod`

DiscreteLpUFuncs.greater

`DiscreteLpUFuncs.greater(x2, out=None)`

Return the truth value of ($x_1 > x_2$) element-wise.

See also:

`numpy.greater`

DiscreteLpUFuncs.greater_equal

`DiscreteLpUFuncs.greater_equal(x2, out=None)`

Return the truth value of ($x_1 \geq x_2$) element-wise.

See also:

`numpy.greater_equal`

DiscreteLpUFuncs.hypot

`DiscreteLpUFuncs.hypot(x2, out=None)`

Given the “legs” of a right triangle, return its hypotenuse.

See also:

`numpy.hypot`

DiscreteLpUFuncs.invert

`DiscreteLpUFuncs.invert (out=None)`

Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

`numpy.invert`

DiscreteLpUFuncs.isfinite

`DiscreteLpUFuncs.isfinite (out=None)`

Test element-wise for finiteness (not infinity or not Not a Number).

See also:

`numpy.isfinite`

DiscreteLpUFuncs.isinf

`DiscreteLpUFuncs.isinf (out=None)`

Test element-wise for positive or negative infinity.

See also:

`numpy.isinf`

DiscreteLpUFuncs.isnan

`DiscreteLpUFuncs.isnan (out=None)`

Test element-wise for NaN and return result as a boolean array.

See also:

`numpy.isnan`

DiscreteLpUFuncs.left_shift

`DiscreteLpUFuncs.left_shift (x2, out=None)`

Shift the bits of an integer to the left.

See also:

`numpy.left_shift`

DiscreteLpUFuncs.less

`DiscreteLpUFuncs.less (x2, out=None)`

Return the truth value of ($x_1 < x_2$) element-wise.

See also:

`numpy.less`

DiscreteLpUFuncs.less_equal

`DiscreteLpUFuncs.less_equal (x2, out=None)`

Return the truth value of ($x_1 \leq x_2$) element-wise.

See also:

`numpy.less_equal`

DiscreteLpUFuncs.log

`DiscreteLpUFuncs.log` (*out=None*)

Natural logarithm, element-wise.

See also:

`numpy.log`

DiscreteLpUFuncs.log10

`DiscreteLpUFuncs.log10` (*out=None*)

Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

DiscreteLpUFuncs.log1p

`DiscreteLpUFuncs.log1p` (*out=None*)

Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

DiscreteLpUFuncs.log2

`DiscreteLpUFuncs.log2` (*out=None*)

Base-2 logarithm of *x*.

See also:

`numpy.log2`

DiscreteLpUFuncs.logaddexp

`DiscreteLpUFuncs.logaddexp` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

DiscreteLpUFuncs.logaddexp2

`DiscreteLpUFuncs.logaddexp2` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

DiscreteLpUFuncs.logical_and

`DiscreteLpUFuncs.logical_and` (*x2, out=None*)

Compute the truth value of *x1* AND *x2* element-wise.

See also:

`numpy.logical_and`

DiscreteLpUFuncs.logical_not

`DiscreteLpUFuncs.logical_not (out=None)`

Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

DiscreteLpUFuncs.logical_or

`DiscreteLpUFuncs.logical_or (x2, out=None)`

Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

DiscreteLpUFuncs.logical_xor

`DiscreteLpUFuncs.logical_xor (x2, out=None)`

Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

DiscreteLpUFuncs.max

`DiscreteLpUFuncs.max ()`

Maximum value in array.

See also:

`numpy.amax`

DiscreteLpUFuncs.maximum

`DiscreteLpUFuncs.maximum (x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.maximum`

DiscreteLpUFuncs.min

`DiscreteLpUFuncs.min ()`

Minimum value in array.

See also:

`numpy.amin`

DiscreteLpUFuncs.minimum

`DiscreteLpUFuncs.minimum (x2, out=None)`

Element-wise minimum of array elements.

See also:

`numpy.minimum`

DiscreteLpUFuncs.mod

`DiscreteLpUFuncs.mod(x2, out=None)`
Return element-wise remainder of division.

See also:

`numpy.mod`

DiscreteLpUFuncs.modf

`DiscreteLpUFuncs.modf(out1=None, out2=None)`
Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

DiscreteLpUFuncs.multiply

`DiscreteLpUFuncs.multiply(x2, out=None)`
Multiply arguments element-wise.

See also:

`numpy.multiply`

DiscreteLpUFuncs.negative

`DiscreteLpUFuncs.negative(out=None)`
Numerical negative, element-wise.

See also:

`numpy.negative`

DiscreteLpUFuncs.not_equal

`DiscreteLpUFuncs.not_equal(x2, out=None)`
Return $(x1 \neq x2)$ element-wise.

See also:

`numpy.not_equal`

DiscreteLpUFuncs.power

`DiscreteLpUFuncs.power(x2, out=None)`
First array elements raised to powers from second array, element-wise.

See also:

`numpy.power`

DiscreteLpUFuncs.prod

`DiscreteLpUFuncs.prod()`
Product of array elements.

See also:

`numpy.prod`

DiscreteLpUFuncs.rad2deg

`DiscreteLpUFuncs.rad2deg (out=None)`

Convert angles from radians to degrees.

See also:

`numpy.rad2deg`

DiscreteLpUFuncs.reciprocal

`DiscreteLpUFuncs.reciprocal (out=None)`

Return the reciprocal of the argument, element-wise.

See also:

`numpy.reciprocal`

DiscreteLpUFuncs.remainder

`DiscreteLpUFuncs.remainder (x2, out=None)`

Return element-wise remainder of division.

See also:

`numpy.remainder`

DiscreteLpUFuncs.right_shift

`DiscreteLpUFuncs.right_shift (x2, out=None)`

Shift the bits of an integer to the right.

See also:

`numpy.right_shift`

DiscreteLpUFuncs rint

`DiscreteLpUFuncs.rint (out=None)`

Round elements of the array to the nearest integer.

See also:

`numpy.rint`

DiscreteLpUFuncs.sign

`DiscreteLpUFuncs.sign (out=None)`

Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

DiscreteLpUFuncs.signbit

`DiscreteLpUFuncs.signbit (out=None)`

Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

DiscreteLpUFuncs.sin

`DiscreteLpUFuncs.sin` (*out=None*)
Trigonometric sine, element-wise.

See also:

`numpy.sin`

DiscreteLpUFuncs.sinh

`DiscreteLpUFuncs.sinh` (*out=None*)
Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

DiscreteLpUFuncs.sqrt

`DiscreteLpUFuncs.sqrt` (*out=None*)
Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

DiscreteLpUFuncs.square

`DiscreteLpUFuncs.square` (*out=None*)
Return the element-wise square of the input.

See also:

`numpy.square`

DiscreteLpUFuncs.subtract

`DiscreteLpUFuncs.subtract` (*x2, out=None*)
Subtract arguments, element-wise.

See also:

`numpy.subtract`

DiscreteLpUFuncs.sum

`DiscreteLpUFuncs.sum` ()
Sum of array elements.

See also:

`numpy.sum`

DiscreteLpUFuncs.tan

`DiscreteLpUFuncs.tan` (*out=None*)
Compute tangent element-wise.

See also:

`numpy.tan`

DiscreteLpUFuncs.tanh`DiscreteLpUFuncs.tanh (out=None)`

Compute hyperbolic tangent element-wise.

See also:`numpy.tanh`**DiscreteLpUFuncs.true_divide**`DiscreteLpUFuncs.true_divide (x2, out=None)`

Returns a true division of the inputs, element-wise.

See also:`numpy.true_divide`**DiscreteLpUFuncs.trunc**`DiscreteLpUFuncs.trunc (out=None)`

Return the truncated value of the input, element-wise.

See also:`numpy.trunc``__init__ (vector)`

Create ufunc wrapper for vector.

NtuplesBaseUFuncs`class odl.util.ufuncs.NtuplesBaseUFuncs (vector)`Bases: `object`UFuncs for *NtuplesBaseVector* objects.Internal object, should not be created except in *NtuplesBaseVector*.**Methods**

<code>__eq__</code>	Return self==value.
<code>absolute([out])</code>	Calculate the absolute value element-wise.
<code>add(x2[, out])</code>	Add arguments element-wise.
<code>arccos([out])</code>	Trigonometric inverse cosine, element-wise.
<code>arccosh([out])</code>	Inverse hyperbolic cosine, element-wise.
<code>arcsin([out])</code>	Inverse sine, element-wise.
<code>arcsinh([out])</code>	Inverse hyperbolic sine element-wise.
<code>arctan([out])</code>	Trigonometric inverse tangent, element-wise.
<code>arctan2(x2[, out])</code>	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
<code>arctanh([out])</code>	Inverse hyperbolic tangent element-wise.
<code>bitwise_and(x2[, out])</code>	Compute the bit-wise AND of two arrays element-wise.
<code>bitwise_or(x2[, out])</code>	Compute the bit-wise OR of two arrays element-wise.
<code>bitwise_xor(x2[, out])</code>	Compute the bit-wise XOR of two arrays element-wise.
<code>ceil([out])</code>	Return the ceiling of the input, element-wise.
<code>conj([out])</code>	Return the complex conjugate, element-wise.
<code>copysign(x2[, out])</code>	Change the sign of x1 to that of x2, element-wise.
<code>cos([out])</code>	Cosine element-wise.

Continued on next page

Table 2.304 – continued from previous page

<i>cosh</i> ([out])	Hyperbolic cosine, element-wise.
<i>deg2rad</i> ([out])	Convert angles from degrees to radians.
<i>divide</i> (x2[, out])	Returns a true division of the inputs, element-wise.
<i>equal</i> (x2[, out])	Return (x1 == x2) element-wise.
<i>exp</i> ([out])	Calculate the exponential of all elements in the input array.
<i>exp2</i> ([out])	Calculate 2^{**p} for all p in the input array.
<i>expm1</i> ([out])	Calculate $\exp(x) - 1$ for all elements in the array.
<i>floor</i> ([out])	Return the floor of the input, element-wise.
<i>floor_divide</i> (x2[, out])	Return the largest integer smaller or equal to the division of the
<i>fmax</i> (x2[, out])	Element-wise maximum of array elements.
<i>fmin</i> (x2[, out])	Element-wise minimum of array elements.
<i>fmod</i> (x2[, out])	Return the element-wise remainder of division.
<i>greater</i> (x2[, out])	Return the truth value of (x1 > x2) element-wise.
<i>greater_equal</i> (x2[, out])	Return the truth value of (x1 >= x2) element-wise.
<i>hypot</i> (x2[, out])	Given the “legs” of a right triangle, return its hypotenuse.
<i>invert</i> ([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<i>isfinite</i> ([out])	Test element-wise for finiteness (not infinity or not Not a Number).
<i>isinf</i> ([out])	Test element-wise for positive or negative infinity.
<i>isnan</i> ([out])	Test element-wise for NaN and return result as a boolean array.
<i>left_shift</i> (x2[, out])	Shift the bits of an integer to the left.
<i>less</i> (x2[, out])	Return the truth value of (x1 < x2) element-wise.
<i>less_equal</i> (x2[, out])	Return the truth value of (x1 <= x2) element-wise.
<i>log</i> ([out])	Natural logarithm, element-wise.
<i>log10</i> ([out])	Return the base 10 logarithm of the input array, element-wise.
<i>log1p</i> ([out])	Return the natural logarithm of one plus the input array, element-wise.
<i>log2</i> ([out])	Base-2 logarithm of x.
<i>logaddexp</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs.
<i>logaddexp2</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
<i>logical_and</i> (x2[, out])	Compute the truth value of x1 AND x2 element-wise.
<i>logical_not</i> ([out])	Compute the truth value of NOT x element-wise.
<i>logical_or</i> (x2[, out])	Compute the truth value of x1 OR x2 element-wise.
<i>logical_xor</i> (x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
<i>max</i> ()	Maximum value in array.
<i>maximum</i> (x2[, out])	Element-wise maximum of array elements.
<i>min</i> ()	Minimum value in array.
<i>minimum</i> (x2[, out])	Element-wise minimum of array elements.
<i>mod</i> (x2[, out])	Return element-wise remainder of division.
<i>modf</i> ([out1, out2])	Return the fractional and integral parts of an array, element-wise.
<i>multiply</i> (x2[, out])	Multiply arguments element-wise.
<i>negative</i> ([out])	Numerical negative, element-wise.
<i>not_equal</i> (x2[, out])	Return (x1 != x2) element-wise.
<i>power</i> (x2[, out])	First array elements raised to powers from second array, element-wise.
<i>prod</i> ()	Product of array elements.
<i>rad2deg</i> ([out])	Convert angles from radians to degrees.
<i>reciprocal</i> ([out])	Return the reciprocal of the argument, element-wise.
<i>remainder</i> (x2[, out])	Return element-wise remainder of division.
<i>right_shift</i> (x2[, out])	Shift the bits of an integer to the right.
<i>rint</i> ([out])	Round elements of the array to the nearest integer.
<i>sign</i> ([out])	Returns an element-wise indication of the sign of a number.
<i>signbit</i> ([out])	Returns element-wise True where signbit is set (less than zero).

Continued on next page

Table 2.304 – continued from previous page

<code>sin([out])</code>	Trigonometric sine, element-wise.
<code>sinh([out])</code>	Hyperbolic sine, element-wise.
<code>sqrt([out])</code>	Return the positive square-root of an array, element-wise.
<code>square([out])</code>	Return the element-wise square of the input.
<code>subtract(x2[, out])</code>	Subtract arguments, element-wise.
<code>sum()</code>	Sum of array elements.
<code>tan([out])</code>	Compute tangent element-wise.
<code>tanh([out])</code>	Compute hyperbolic tangent element-wise.
<code>true_divide(x2[, out])</code>	Returns a true division of the inputs, element-wise.
<code>trunc([out])</code>	Return the truncated value of the input, element-wise.

NtuplesBaseUFuncs.absolute

`NtuplesBaseUFuncs.absolute (out=None)`

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

NtuplesBaseUFuncs.add

`NtuplesBaseUFuncs.add (x2, out=None)`

Add arguments element-wise.

See also:

`numpy.add`

NtuplesBaseUFuncs.arccos

`NtuplesBaseUFuncs.arccos (out=None)`

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

NtuplesBaseUFuncs.arccosh

`NtuplesBaseUFuncs.arccosh (out=None)`

Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

NtuplesBaseUFuncs.arcsin

`NtuplesBaseUFuncs.arcsin (out=None)`

Inverse sine, element-wise.

See also:

`numpy.arcsin`

NtuplesBaseUFuncs.arcsinh

`NtuplesBaseUFuncs.arcsinh` (*out=None*)

Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

NtuplesBaseUFuncs.arctan

`NtuplesBaseUFuncs.arctan` (*out=None*)

Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

NtuplesBaseUFuncs.arctan2

`NtuplesBaseUFuncs.arctan2` (*x2, out=None*)

Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

See also:

`numpy.arctan2`

NtuplesBaseUFuncs.arctanh

`NtuplesBaseUFuncs.arctanh` (*out=None*)

Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

NtuplesBaseUFuncs.bitwise_and

`NtuplesBaseUFuncs.bitwise_and` (*x2, out=None*)

Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

NtuplesBaseUFuncs.bitwise_or

`NtuplesBaseUFuncs.bitwise_or` (*x2, out=None*)

Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

NtuplesBaseUFuncs.bitwise_xor

`NtuplesBaseUFuncs.bitwise_xor` (*x2, out=None*)

Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

NtuplesBaseUFuncs.ceil

`NtuplesBaseUFuncs.ceil` (*out=None*)

Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

NtuplesBaseUFuncs.conj

`NtuplesBaseUFuncs.conj` (*out=None*)

Return the complex conjugate, element-wise.

See also:

`numpy.conj`

NtuplesBaseUFuncs.copysign

`NtuplesBaseUFuncs.copysign` (*x2, out=None*)

Change the sign of *x1* to that of *x2*, element-wise.

See also:

`numpy.copysign`

NtuplesBaseUFuncs.cos

`NtuplesBaseUFuncs.cos` (*out=None*)

Cosine element-wise.

See also:

`numpy.cos`

NtuplesBaseUFuncs.cosh

`NtuplesBaseUFuncs.cosh` (*out=None*)

Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

NtuplesBaseUFuncs.deg2rad

`NtuplesBaseUFuncs.deg2rad` (*out=None*)

Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

NtuplesBaseUFuncs.divide

`NtuplesBaseUFuncs.divide` (*x2, out=None*)

Returns a true division of the inputs, element-wise.

See also:

`numpy.divide`

NtuplesBaseUFuncs.equal

`NtuplesBaseUFuncs.equal(x2, out=None)`

Return $(x1 == x2)$ element-wise.

See also:

`numpy.equal`

NtuplesBaseUFuncs.exp

`NtuplesBaseUFuncs.exp(out=None)`

Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

NtuplesBaseUFuncs.exp2

`NtuplesBaseUFuncs.exp2(out=None)`

Calculate 2^{**p} for all p in the input array.

See also:

`numpy.exp2`

NtuplesBaseUFuncs.expm1

`NtuplesBaseUFuncs.expm1(out=None)`

Calculate $\exp(x) - 1$ for all elements in the array.

See also:

`numpy.expm1`

NtuplesBaseUFuncs.floor

`NtuplesBaseUFuncs.floor(out=None)`

Return the floor of the input, element-wise.

See also:

`numpy.floor`

NtuplesBaseUFuncs.floor_divide

`NtuplesBaseUFuncs.floor_divide(x2, out=None)`

Return the largest integer smaller or equal to the division of the

See also:

`numpy.floor_divide`

NtuplesBaseUFuncs.fmax

`NtuplesBaseUFuncs.fmax(x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.fmax`

NtuplesBaseUFuncs.fmin

`NtuplesBaseUFuncs.fmin` (*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

`numpy.fmin`

NtuplesBaseUFuncs.fmod

`NtuplesBaseUFuncs.fmod` (*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

`numpy.fmod`

NtuplesBaseUFuncs.greater

`NtuplesBaseUFuncs.greater` (*x2*, *out=None*)
Return the truth value of (*x1* > *x2*) element-wise.

See also:

`numpy.greater`

NtuplesBaseUFuncs.greater_equal

`NtuplesBaseUFuncs.greater_equal` (*x2*, *out=None*)
Return the truth value of (*x1* >= *x2*) element-wise.

See also:

`numpy.greater_equal`

NtuplesBaseUFuncs.hypot

`NtuplesBaseUFuncs.hypot` (*x2*, *out=None*)
Given the “legs” of a right triangle, return its hypotenuse.

See also:

`numpy.hypot`

NtuplesBaseUFuncs.invert

`NtuplesBaseUFuncs.invert` (*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

`numpy.invert`

NtuplesBaseUFuncs.isfinite

`NtuplesBaseUFuncs.isfinite` (*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

`numpy.isfinite`

NtuplesBaseUFuncs.isinf

`NtuplesBaseUFuncs.isinf` (*out=None*)

Test element-wise for positive or negative infinity.

See also:

`numpy.isinf`

NtuplesBaseUFuncs.isnan

`NtuplesBaseUFuncs.isnan` (*out=None*)

Test element-wise for NaN and return result as a boolean array.

See also:

`numpy.isnan`

NtuplesBaseUFuncs.left_shift

`NtuplesBaseUFuncs.left_shift` (*x2, out=None*)

Shift the bits of an integer to the left.

See also:

`numpy.left_shift`

NtuplesBaseUFuncs.less

`NtuplesBaseUFuncs.less` (*x2, out=None*)

Return the truth value of (*x1* < *x2*) element-wise.

See also:

`numpy.less`

NtuplesBaseUFuncs.less_equal

`NtuplesBaseUFuncs.less_equal` (*x2, out=None*)

Return the truth value of (*x1* ≤ *x2*) element-wise.

See also:

`numpy.less_equal`

NtuplesBaseUFuncs.log

`NtuplesBaseUFuncs.log` (*out=None*)

Natural logarithm, element-wise.

See also:

`numpy.log`

NtuplesBaseUFuncs.log10

`NtuplesBaseUFuncs.log10` (*out=None*)

Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

NtuplesBaseUFuncs.log1p

`NtuplesBaseUFuncs.log1p` (*out=None*)

Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

NtuplesBaseUFuncs.log2

`NtuplesBaseUFuncs.log2` (*out=None*)

Base-2 logarithm of x.

See also:

`numpy.log2`

NtuplesBaseUFuncs.logaddexp

`NtuplesBaseUFuncs.logaddexp` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

NtuplesBaseUFuncs.logaddexp2

`NtuplesBaseUFuncs.logaddexp2` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

NtuplesBaseUFuncs.logical_and

`NtuplesBaseUFuncs.logical_and` (*x2, out=None*)

Compute the truth value of x1 AND x2 element-wise.

See also:

`numpy.logical_and`

NtuplesBaseUFuncs.logical_not

`NtuplesBaseUFuncs.logical_not` (*out=None*)

Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

NtuplesBaseUFuncs.logical_or

`NtuplesBaseUFuncs.logical_or` (*x2, out=None*)

Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

NtuplesBaseUFuncs.logical_xor

`NtuplesBaseUFuncs.logical_xor(x2, out=None)`

Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

NtuplesBaseUFuncs.max

`NtuplesBaseUFuncs.max()`

Maximum value in array.

See also:

`numpy.amax`

NtuplesBaseUFuncs.maximum

`NtuplesBaseUFuncs.maximum(x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.maximum`

NtuplesBaseUFuncs.min

`NtuplesBaseUFuncs.min()`

Minimum value in array.

See also:

`numpy.amin`

NtuplesBaseUFuncs.minimum

`NtuplesBaseUFuncs.minimum(x2, out=None)`

Element-wise minimum of array elements.

See also:

`numpy.minimum`

NtuplesBaseUFuncs.mod

`NtuplesBaseUFuncs.mod(x2, out=None)`

Return element-wise remainder of division.

See also:

`numpy.mod`

NtuplesBaseUFuncs.modf

`NtuplesBaseUFuncs.modf(out1=None, out2=None)`

Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

NtuplesBaseUFuncs.multiply

`NtuplesBaseUFuncs.multiply` (*x2*, *out=None*)

Multiply arguments element-wise.

See also:

`numpy.multiply`

NtuplesBaseUFuncs.negative

`NtuplesBaseUFuncs.negative` (*out=None*)

Numerical negative, element-wise.

See also:

`numpy.negative`

NtuplesBaseUFuncs.not_equal

`NtuplesBaseUFuncs.not_equal` (*x2*, *out=None*)

Return (*x1* != *x2*) element-wise.

See also:

`numpy.not_equal`

NtuplesBaseUFuncs.power

`NtuplesBaseUFuncs.power` (*x2*, *out=None*)

First array elements raised to powers from second array, element-wise.

See also:

`numpy.power`

NtuplesBaseUFuncs.prod

`NtuplesBaseUFuncs.prod` ()

Product of array elements.

See also:

`numpy.prod`

NtuplesBaseUFuncs.rad2deg

`NtuplesBaseUFuncs.rad2deg` (*out=None*)

Convert angles from radians to degrees.

See also:

`numpy.rad2deg`

NtuplesBaseUFuncs.reciprocal

`NtuplesBaseUFuncs.reciprocal` (*out=None*)

Return the reciprocal of the argument, element-wise.

See also:

`numpy.reciprocal`

NtuplesBaseUFuncs.remainder

`NtuplesBaseUFuncs.remainder` (*x2*, *out=None*)

Return element-wise remainder of division.

See also:

`numpy.remainder`

NtuplesBaseUFuncs.right_shift

`NtuplesBaseUFuncs.right_shift` (*x2*, *out=None*)

Shift the bits of an integer to the right.

See also:

`numpy.right_shift`

NtuplesBaseUFuncs rint

`NtuplesBaseUFuncs.rint` (*out=None*)

Round elements of the array to the nearest integer.

See also:

`numpy.rint`

NtuplesBaseUFuncs.sign

`NtuplesBaseUFuncs.sign` (*out=None*)

Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

NtuplesBaseUFuncs.signbit

`NtuplesBaseUFuncs.signbit` (*out=None*)

Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

NtuplesBaseUFuncs.sin

`NtuplesBaseUFuncs.sin` (*out=None*)

Trigonometric sine, element-wise.

See also:

`numpy.sin`

NtuplesBaseUFuncs.sinh

`NtuplesBaseUFuncs.sinh` (*out=None*)

Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

NtuplesBaseUFuncs.sqrt

`NtuplesBaseUFuncs.sqrt (out=None)`

Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

NtuplesBaseUFuncs.square

`NtuplesBaseUFuncs.square (out=None)`

Return the element-wise square of the input.

See also:

`numpy.square`

NtuplesBaseUFuncs.subtract

`NtuplesBaseUFuncs.subtract (x2, out=None)`

Subtract arguments, element-wise.

See also:

`numpy.subtract`

NtuplesBaseUFuncs.sum

`NtuplesBaseUFuncs.sum ()`

Sum of array elements.

See also:

`numpy.sum`

NtuplesBaseUFuncs.tan

`NtuplesBaseUFuncs.tan (out=None)`

Compute tangent element-wise.

See also:

`numpy.tan`

NtuplesBaseUFuncs.tanh

`NtuplesBaseUFuncs.tanh (out=None)`

Compute hyperbolic tangent element-wise.

See also:

`numpy.tanh`

NtuplesBaseUFuncs.true_divide

`NtuplesBaseUFuncs.true_divide (x2, out=None)`

Returns a true division of the inputs, element-wise.

See also:

`numpy.true_divide`

NtuplesBaseUFuncs.trunc

`NtuplesBaseUFuncs.trunc` (*out=None*)

Return the truncated value of the input, element-wise.

See also:

`numpy.trunc`

`__init__` (*vector*)

Create ufunc wrapper for vector.

NtuplesUFuncs

class `odl.util.ufuncs.NtuplesUFuncs` (*vector*)

Bases: `odl.util.ufuncs.NtuplesBaseUFuncs`

UFuncs for *NtuplesVector* objects.

Internal object, should not be created except in *NtuplesVector*.

Methods

<code>__eq__</code>	Return self==value.
<code>absolute([out])</code>	Calculate the absolute value element-wise.
<code>add(x2[, out])</code>	Add arguments element-wise.
<code>arccos([out])</code>	Trigonometric inverse cosine, element-wise.
<code>arccosh([out])</code>	Inverse hyperbolic cosine, element-wise.
<code>arcsin([out])</code>	Inverse sine, element-wise.
<code>arcsinh([out])</code>	Inverse hyperbolic sine element-wise.
<code>arctan([out])</code>	Trigonometric inverse tangent, element-wise.
<code>arctan2(x2[, out])</code>	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
<code>arctanh([out])</code>	Inverse hyperbolic tangent element-wise.
<code>bitwise_and(x2[, out])</code>	Compute the bit-wise AND of two arrays element-wise.
<code>bitwise_or(x2[, out])</code>	Compute the bit-wise OR of two arrays element-wise.
<code>bitwise_xor(x2[, out])</code>	Compute the bit-wise XOR of two arrays element-wise.
<code>ceil([out])</code>	Return the ceiling of the input, element-wise.
<code>conj([out])</code>	Return the complex conjugate, element-wise.
<code>copysign(x2[, out])</code>	Change the sign of x1 to that of x2, element-wise.
<code>cos([out])</code>	Cosine element-wise.
<code>cosh([out])</code>	Hyperbolic cosine, element-wise.
<code>deg2rad([out])</code>	Convert angles from degrees to radians.
<code>divide(x2[, out])</code>	Returns a true division of the inputs, element-wise.
<code>equal(x2[, out])</code>	Return (x1 == x2) element-wise.
<code>exp([out])</code>	Calculate the exponential of all elements in the input array.
<code>exp2([out])</code>	Calculate 2**p for all p in the input array.
<code>expm1([out])</code>	Calculate $\exp(x) - 1$ for all elements in the array.
<code>floor([out])</code>	Return the floor of the input, element-wise.
<code>floor_divide(x2[, out])</code>	Return the largest integer smaller or equal to the division of the
<code>fmax(x2[, out])</code>	Element-wise maximum of array elements.
<code>fmin(x2[, out])</code>	Element-wise minimum of array elements.
<code>fmod(x2[, out])</code>	Return the element-wise remainder of division.
<code>greater(x2[, out])</code>	Return the truth value of (x1 > x2) element-wise.
<code>greater_equal(x2[, out])</code>	Return the truth value of (x1 >= x2) element-wise.
<code>hypot(x2[, out])</code>	Given the “legs” of a right triangle, return its hypotenuse.

Continued on next page

Table 2.305 – continued from previous page

<i>invert</i> ([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<i>isfinite</i> ([out])	Test element-wise for finiteness (not infinity or not Not a Number).
<i>isinf</i> ([out])	Test element-wise for positive or negative infinity.
<i>isnan</i> ([out])	Test element-wise for NaN and return result as a boolean array.
<i>left_shift</i> (x2[, out])	Shift the bits of an integer to the left.
<i>less</i> (x2[, out])	Return the truth value of (x1 < x2) element-wise.
<i>less_equal</i> (x2[, out])	Return the truth value of (x1 ≤ x2) element-wise.
<i>log</i> ([out])	Natural logarithm, element-wise.
<i>log10</i> ([out])	Return the base 10 logarithm of the input array, element-wise.
<i>log1p</i> ([out])	Return the natural logarithm of one plus the input array, element-wise.
<i>log2</i> ([out])	Base-2 logarithm of x.
<i>logaddexp</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs.
<i>logaddexp2</i> (x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
<i>logical_and</i> (x2[, out])	Compute the truth value of x1 AND x2 element-wise.
<i>logical_not</i> ([out])	Compute the truth value of NOT x element-wise.
<i>logical_or</i> (x2[, out])	Compute the truth value of x1 OR x2 element-wise.
<i>logical_xor</i> (x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
<i>max</i> ()	Maximum value in array.
<i>maximum</i> (x2[, out])	Element-wise maximum of array elements.
<i>min</i> ()	Minimum value in array.
<i>minimum</i> (x2[, out])	Element-wise minimum of array elements.
<i>mod</i> (x2[, out])	Return element-wise remainder of division.
<i>modf</i> ([out1, out2])	Return the fractional and integral parts of an array, element-wise.
<i>multiply</i> (x2[, out])	Multiply arguments element-wise.
<i>negative</i> ([out])	Numerical negative, element-wise.
<i>not_equal</i> (x2[, out])	Return (x1 != x2) element-wise.
<i>power</i> (x2[, out])	First array elements raised to powers from second array, element-wise.
<i>prod</i> ()	Product of array elements.
<i>rad2deg</i> ([out])	Convert angles from radians to degrees.
<i>reciprocal</i> ([out])	Return the reciprocal of the argument, element-wise.
<i>remainder</i> (x2[, out])	Return element-wise remainder of division.
<i>right_shift</i> (x2[, out])	Shift the bits of an integer to the right.
<i>rint</i> ([out])	Round elements of the array to the nearest integer.
<i>sign</i> ([out])	Returns an element-wise indication of the sign of a number.
<i>signbit</i> ([out])	Returns element-wise True where signbit is set (less than zero).
<i>sin</i> ([out])	Trigonometric sine, element-wise.
<i>sinh</i> ([out])	Hyperbolic sine, element-wise.
<i>sqrt</i> ([out])	Return the positive square-root of an array, element-wise.
<i>square</i> ([out])	Return the element-wise square of the input.
<i>subtract</i> (x2[, out])	Subtract arguments, element-wise.
<i>sum</i> ()	Sum of array elements.
<i>tan</i> ([out])	Compute tangent element-wise.
<i>tanh</i> ([out])	Compute hyperbolic tangent element-wise.
<i>true_divide</i> (x2[, out])	Returns a true division of the inputs, element-wise.
<i>trunc</i> ([out])	Return the truncated value of the input, element-wise.

NtuplesUFuncs.absoluteNtuplesUFuncs.**absolute** (*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

NtuplesUFuncs.add

`NtuplesUFuncs.add(x2, out=None)`

Add arguments element-wise.

See also:

`numpy.add`

NtuplesUFuncs.arccos

`NtuplesUFuncs.arccos(out=None)`

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

NtuplesUFuncs.arccosh

`NtuplesUFuncs.arccosh(out=None)`

Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

NtuplesUFuncs.arcsin

`NtuplesUFuncs.arcsin(out=None)`

Inverse sine, element-wise.

See also:

`numpy.arcsin`

NtuplesUFuncs.arcsinh

`NtuplesUFuncs.arcsinh(out=None)`

Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

NtuplesUFuncs.arctan

`NtuplesUFuncs.arctan(out=None)`

Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

NtuplesUFuncs.arctan2

`NtuplesUFuncs.arctan2(x2, out=None)`

Element-wise arc tangent of x_1/x_2 choosing the quadrant correctly.

See also:

`numpy.arctan2`

NtuplesUFuncs.arctanh

`NtuplesUFuncs.arctanh` (*out=None*)

Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

NtuplesUFuncs.bitwise_and

`NtuplesUFuncs.bitwise_and` (*x2, out=None*)

Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

NtuplesUFuncs.bitwise_or

`NtuplesUFuncs.bitwise_or` (*x2, out=None*)

Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

NtuplesUFuncs.bitwise_xor

`NtuplesUFuncs.bitwise_xor` (*x2, out=None*)

Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

NtuplesUFuncs.ceil

`NtuplesUFuncs.ceil` (*out=None*)

Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

NtuplesUFuncs.conj

`NtuplesUFuncs.conj` (*out=None*)

Return the complex conjugate, element-wise.

See also:

`numpy.conj`

NtuplesUFuncs.copysign

`NtuplesUFuncs.copysign` (*x2, out=None*)

Change the sign of *x1* to that of *x2*, element-wise.

See also:

`numpy.copysign`

NtuplesUFuncs.cos

`NtuplesUFuncs.cos` (*out=None*)

Cosine element-wise.

See also:

`numpy.cos`

NtuplesUFuncs.cosh

`NtuplesUFuncs.cosh` (*out=None*)

Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

NtuplesUFuncs.deg2rad

`NtuplesUFuncs.deg2rad` (*out=None*)

Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

NtuplesUFuncs.divide

`NtuplesUFuncs.divide` (*x2, out=None*)

Returns a true division of the inputs, element-wise.

See also:

`numpy.divide`

NtuplesUFuncs.equal

`NtuplesUFuncs.equal` (*x2, out=None*)

Return $(x1 == x2)$ element-wise.

See also:

`numpy.equal`

NtuplesUFuncs.exp

`NtuplesUFuncs.exp` (*out=None*)

Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

NtuplesUFuncs.exp2

`NtuplesUFuncs.exp2` (*out=None*)

Calculate 2^{**p} for all p in the input array.

See also:

`numpy.exp2`

NtuplesUFuncs.expm1

`NtuplesUFuncs.expm1` (*out=None*)

Calculate $\exp(x) - 1$ for all elements in the array.

See also:

`numpy.expm1`

NtuplesUFuncs.floor

`NtuplesUFuncs.floor` (*out=None*)

Return the floor of the input, element-wise.

See also:

`numpy.floor`

NtuplesUFuncs.floor_divide

`NtuplesUFuncs.floor_divide` (*x2, out=None*)

Return the largest integer smaller or equal to the division of the

See also:

`numpy.floor_divide`

NtuplesUFuncs.fmax

`NtuplesUFuncs.fmax` (*x2, out=None*)

Element-wise maximum of array elements.

See also:

`numpy.fmax`

NtuplesUFuncs.fmin

`NtuplesUFuncs.fmin` (*x2, out=None*)

Element-wise minimum of array elements.

See also:

`numpy.fmin`

NtuplesUFuncs.fmod

`NtuplesUFuncs.fmod` (*x2, out=None*)

Return the element-wise remainder of division.

See also:

`numpy.fmod`

NtuplesUFuncs.greater

`NtuplesUFuncs.greater` (*x2, out=None*)

Return the truth value of ($x1 > x2$) element-wise.

See also:

`numpy.greater`

NtuplesUFuncs.greater_equal

`NtuplesUFuncs.greater_equal` (*x2*, *out=None*)

Return the truth value of (*x1* >= *x2*) element-wise.

See also:

`numpy.greater_equal`

NtuplesUFuncs.hypot

`NtuplesUFuncs.hypot` (*x2*, *out=None*)

Given the “legs” of a right triangle, return its hypotenuse.

See also:

`numpy.hypot`

NtuplesUFuncs.invert

`NtuplesUFuncs.invert` (*out=None*)

Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

`numpy.invert`

NtuplesUFuncs.isfinite

`NtuplesUFuncs.isfinite` (*out=None*)

Test element-wise for finiteness (not infinity or not Not a Number).

See also:

`numpy.isfinite`

NtuplesUFuncs.isinf

`NtuplesUFuncs.isinf` (*out=None*)

Test element-wise for positive or negative infinity.

See also:

`numpy.isinf`

NtuplesUFuncs.isnan

`NtuplesUFuncs.isnan` (*out=None*)

Test element-wise for NaN and return result as a boolean array.

See also:

`numpy.isnan`

NtuplesUFuncs.left_shift

`NtuplesUFuncs.left_shift` (*x2*, *out=None*)

Shift the bits of an integer to the left.

See also:

`numpy.left_shift`

NtuplesUFuncs.less

`NtuplesUFuncs.less(x2, out=None)`

Return the truth value of $(x1 < x2)$ element-wise.

See also:

`numpy.less`

NtuplesUFuncs.less_equal

`NtuplesUFuncs.less_equal(x2, out=None)`

Return the truth value of $(x1 \leq x2)$ element-wise.

See also:

`numpy.less_equal`

NtuplesUFuncs.log

`NtuplesUFuncs.log(out=None)`

Natural logarithm, element-wise.

See also:

`numpy.log`

NtuplesUFuncs.log10

`NtuplesUFuncs.log10(out=None)`

Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

NtuplesUFuncs.log1p

`NtuplesUFuncs.log1p(out=None)`

Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

NtuplesUFuncs.log2

`NtuplesUFuncs.log2(out=None)`

Base-2 logarithm of x .

See also:

`numpy.log2`

NtuplesUFuncs.logaddexp

`NtuplesUFuncs.logaddexp(x2, out=None)`

Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

NtuplesUFuncs.logaddexp2

`NtuplesUFuncs.logaddexp2(x2, out=None)`

Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

NtuplesUFuncs.logical_and

`NtuplesUFuncs.logical_and(x2, out=None)`

Compute the truth value of x1 AND x2 element-wise.

See also:

`numpy.logical_and`

NtuplesUFuncs.logical_not

`NtuplesUFuncs.logical_not(out=None)`

Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

NtuplesUFuncs.logical_or

`NtuplesUFuncs.logical_or(x2, out=None)`

Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

NtuplesUFuncs.logical_xor

`NtuplesUFuncs.logical_xor(x2, out=None)`

Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

NtuplesUFuncs.max

`NtuplesUFuncs.max()`

Maximum value in array.

See also:

`numpy.amax`

NtuplesUFuncs.maximum

`NtuplesUFuncs.maximum(x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.maximum`

NtuplesUFuncs.min

`NtuplesUFuncs.min()`
Minimum value in array.

See also:

`numpy.amin`

NtuplesUFuncs.minimum

`NtuplesUFuncs.minimum(x2, out=None)`
Element-wise minimum of array elements.

See also:

`numpy.minimum`

NtuplesUFuncs.mod

`NtuplesUFuncs.mod(x2, out=None)`
Return element-wise remainder of division.

See also:

`numpy.mod`

NtuplesUFuncs.modf

`NtuplesUFuncs.modf(out1=None, out2=None)`
Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

NtuplesUFuncs.multiply

`NtuplesUFuncs.multiply(x2, out=None)`
Multiply arguments element-wise.

See also:

`numpy.multiply`

NtuplesUFuncs.negative

`NtuplesUFuncs.negative(out=None)`
Numerical negative, element-wise.

See also:

`numpy.negative`

NtuplesUFuncs.not_equal

`NtuplesUFuncs.not_equal(x2, out=None)`
Return $(x1 \neq x2)$ element-wise.

See also:

`numpy.not_equal`

NtuplesUFuncs.power

`NtuplesUFuncs.power(x2, out=None)`

First array elements raised to powers from second array, element-wise.

See also:

`numpy.power`

NtuplesUFuncs.prod

`NtuplesUFuncs.prod()`

Product of array elements.

See also:

`numpy.prod`

NtuplesUFuncs.rad2deg

`NtuplesUFuncs.rad2deg(out=None)`

Convert angles from radians to degrees.

See also:

`numpy.rad2deg`

NtuplesUFuncs.reciprocal

`NtuplesUFuncs.reciprocal(out=None)`

Return the reciprocal of the argument, element-wise.

See also:

`numpy.reciprocal`

NtuplesUFuncs.remainder

`NtuplesUFuncs.remainder(x2, out=None)`

Return element-wise remainder of division.

See also:

`numpy.remainder`

NtuplesUFuncs.right_shift

`NtuplesUFuncs.right_shift(x2, out=None)`

Shift the bits of an integer to the right.

See also:

`numpy.right_shift`

NtuplesUFuncs rint

`NtuplesUFuncs.rint(out=None)`

Round elements of the array to the nearest integer.

See also:

`numpy.rint`

NtuplesUFuncs.sign

`NtuplesUFuncs.sign` (*out=None*)

Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

NtuplesUFuncs.signbit

`NtuplesUFuncs.signbit` (*out=None*)

Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

NtuplesUFuncs.sin

`NtuplesUFuncs.sin` (*out=None*)

Trigonometric sine, element-wise.

See also:

`numpy.sin`

NtuplesUFuncs.sinh

`NtuplesUFuncs.sinh` (*out=None*)

Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

NtuplesUFuncs.sqrt

`NtuplesUFuncs.sqrt` (*out=None*)

Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

NtuplesUFuncs.square

`NtuplesUFuncs.square` (*out=None*)

Return the element-wise square of the input.

See also:

`numpy.square`

NtuplesUFuncs.subtract

`NtuplesUFuncs.subtract` (*x2, out=None*)

Subtract arguments, element-wise.

See also:

`numpy.subtract`

NtuplesUFuncs.sum

`NtuplesUFuncs.sum()`
Sum of array elements.

See also:

`numpy.sum`

NtuplesUFuncs.tan

`NtuplesUFuncs.tan(out=None)`
Compute tangent element-wise.

See also:

`numpy.tan`

NtuplesUFuncs.tanh

`NtuplesUFuncs.tanh(out=None)`
Compute hyperbolic tangent element-wise.

See also:

`numpy.tanh`

NtuplesUFuncs.true_divide

`NtuplesUFuncs.true_divide(x2, out=None)`
Returns a true division of the inputs, element-wise.

See also:

`numpy.true_divide`

NtuplesUFuncs.trunc

`NtuplesUFuncs.trunc(out=None)`
Return the truncated value of the input, element-wise.

See also:

`numpy.trunc`

`__init__(vector)`

Create ufunc wrapper for vector.

ProductSpaceUFuncs

`class odl.util.ufuncs.ProductSpaceUFuncs(vector)`

Bases: object

UFuncs for *ProductSpaceVector* objects.

Internal object, should not be created except in *ProductSpaceVector*.

Methods

<code>__eq__</code>	Return self==value.
<code>absolute([out])</code>	Calculate the absolute value element-wise.
Continued on next page	

Table 2.306 – continued from previous page

<code>add(x2[, out])</code>	Add arguments element-wise.
<code>arccos([out])</code>	Trigonometric inverse cosine, element-wise.
<code>arccosh([out])</code>	Inverse hyperbolic cosine, element-wise.
<code>arcsin([out])</code>	Inverse sine, element-wise.
<code>arcsinh([out])</code>	Inverse hyperbolic sine element-wise.
<code>arctan([out])</code>	Trigonometric inverse tangent, element-wise.
<code>arctan2(x2[, out])</code>	Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.
<code>arctanh([out])</code>	Inverse hyperbolic tangent element-wise.
<code>bitwise_and(x2[, out])</code>	Compute the bit-wise AND of two arrays element-wise.
<code>bitwise_or(x2[, out])</code>	Compute the bit-wise OR of two arrays element-wise.
<code>bitwise_xor(x2[, out])</code>	Compute the bit-wise XOR of two arrays element-wise.
<code>ceil([out])</code>	Return the ceiling of the input, element-wise.
<code>conj([out])</code>	Return the complex conjugate, element-wise.
<code>copysign(x2[, out])</code>	Change the sign of $x1$ to that of $x2$, element-wise.
<code>cos([out])</code>	Cosine element-wise.
<code>cosh([out])</code>	Hyperbolic cosine, element-wise.
<code>deg2rad([out])</code>	Convert angles from degrees to radians.
<code>divide(x2[, out])</code>	Returns a true division of the inputs, element-wise.
<code>equal(x2[, out])</code>	Return $(x1 == x2)$ element-wise.
<code>exp([out])</code>	Calculate the exponential of all elements in the input array.
<code>exp2([out])</code>	Calculate 2^{**p} for all p in the input array.
<code>expm1([out])</code>	Calculate $\exp(x) - 1$ for all elements in the array.
<code>floor([out])</code>	Return the floor of the input, element-wise.
<code>floor_divide(x2[, out])</code>	Return the largest integer smaller or equal to the division of the
<code>fmax(x2[, out])</code>	Element-wise maximum of array elements.
<code>fmin(x2[, out])</code>	Element-wise minimum of array elements.
<code>fmod(x2[, out])</code>	Return the element-wise remainder of division.
<code>greater(x2[, out])</code>	Return the truth value of $(x1 > x2)$ element-wise.
<code>greater_equal(x2[, out])</code>	Return the truth value of $(x1 \geq x2)$ element-wise.
<code>hypot(x2[, out])</code>	Given the “legs” of a right triangle, return its hypotenuse.
<code>invert([out])</code>	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<code>isfinite([out])</code>	Test element-wise for finiteness (not infinity or not Not a Number).
<code>isinf([out])</code>	Test element-wise for positive or negative infinity.
<code>isnan([out])</code>	Test element-wise for NaN and return result as a boolean array.
<code>left_shift(x2[, out])</code>	Shift the bits of an integer to the left.
<code>less(x2[, out])</code>	Return the truth value of $(x1 < x2)$ element-wise.
<code>less_equal(x2[, out])</code>	Return the truth value of $(x1 \leq x2)$ element-wise.
<code>log([out])</code>	Natural logarithm, element-wise.
<code>log10([out])</code>	Return the base 10 logarithm of the input array, element-wise.
<code>log1p([out])</code>	Return the natural logarithm of one plus the input array, element-wise.
<code>log2([out])</code>	Base-2 logarithm of x .
<code>logaddexp(x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs.
<code>logaddexp2(x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs in base-2.
<code>logical_and(x2[, out])</code>	Compute the truth value of $x1$ AND $x2$ element-wise.
<code>logical_not([out])</code>	Compute the truth value of NOT x element-wise.
<code>logical_or(x2[, out])</code>	Compute the truth value of $x1$ OR $x2$ element-wise.
<code>logical_xor(x2[, out])</code>	Compute the truth value of $x1$ XOR $x2$, element-wise.
<code>max()</code>	Maximum value in array.
<code>maximum(x2[, out])</code>	Element-wise maximum of array elements.
<code>min()</code>	Minimum value in array.

Continued on next page

Table 2.306 – continued from previous page

<i>minimum</i> (x2[, out])	Element-wise minimum of array elements.
<i>mod</i> (x2[, out])	Return element-wise remainder of division.
<i>modf</i> ([out1, out2])	Return the fractional and integral parts of an array, element-wise.
<i>multiply</i> (x2[, out])	Multiply arguments element-wise.
<i>negative</i> ([out])	Numerical negative, element-wise.
<i>not_equal</i> (x2[, out])	Return (x1 != x2) element-wise.
<i>power</i> (x2[, out])	First array elements raised to powers from second array, element-wise.
<i>prod</i> ()	Product of array elements.
<i>rad2deg</i> ([out])	Convert angles from radians to degrees.
<i>reciprocal</i> ([out])	Return the reciprocal of the argument, element-wise.
<i>remainder</i> (x2[, out])	Return element-wise remainder of division.
<i>right_shift</i> (x2[, out])	Shift the bits of an integer to the right.
<i>rint</i> ([out])	Round elements of the array to the nearest integer.
<i>sign</i> ([out])	Returns an element-wise indication of the sign of a number.
<i>signbit</i> ([out])	Returns element-wise True where signbit is set (less than zero).
<i>sin</i> ([out])	Trigonometric sine, element-wise.
<i>sinh</i> ([out])	Hyperbolic sine, element-wise.
<i>sqrt</i> ([out])	Return the positive square-root of an array, element-wise.
<i>square</i> ([out])	Return the element-wise square of the input.
<i>subtract</i> (x2[, out])	Subtract arguments, element-wise.
<i>sum</i> ()	Sum of array elements.
<i>tan</i> ([out])	Compute tangent element-wise.
<i>tanh</i> ([out])	Compute hyperbolic tangent element-wise.
<i>true_divide</i> (x2[, out])	Returns a true division of the inputs, element-wise.
<i>trunc</i> ([out])	Return the truncated value of the input, element-wise.

ProductSpaceUFuncs.absolute

`ProductSpaceUFuncs.absolute` (*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

ProductSpaceUFuncs.add

`ProductSpaceUFuncs.add` (*x2, out=None*)

Add arguments element-wise.

See also:

`numpy.add`

ProductSpaceUFuncs.arccos

`ProductSpaceUFuncs.arccos` (*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

ProductSpaceUFuncs.arccosh

`ProductSpaceUFuncs.arccosh (out=None)`
Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

ProductSpaceUFuncs.arcsin

`ProductSpaceUFuncs.arcsin (out=None)`
Inverse sine, element-wise.

See also:

`numpy.arcsin`

ProductSpaceUFuncs.arcsinh

`ProductSpaceUFuncs.arcsinh (out=None)`
Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

ProductSpaceUFuncs.arctan

`ProductSpaceUFuncs.arctan (out=None)`
Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

ProductSpaceUFuncs.arctan2

`ProductSpaceUFuncs.arctan2 (x2, out=None)`
Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

See also:

`numpy.arctan2`

ProductSpaceUFuncs.arctanh

`ProductSpaceUFuncs.arctanh (out=None)`
Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

ProductSpaceUFuncs.bitwise_and

`ProductSpaceUFuncs.bitwise_and (x2, out=None)`
Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

ProductSpaceUFuncs.bitwise_or

`ProductSpaceUFuncs.bitwise_or` (*x2*, *out=None*)

Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

ProductSpaceUFuncs.bitwise_xor

`ProductSpaceUFuncs.bitwise_xor` (*x2*, *out=None*)

Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

ProductSpaceUFuncs.ceil

`ProductSpaceUFuncs.ceil` (*out=None*)

Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

ProductSpaceUFuncs.conj

`ProductSpaceUFuncs.conj` (*out=None*)

Return the complex conjugate, element-wise.

See also:

`numpy.conj`

ProductSpaceUFuncs.copysign

`ProductSpaceUFuncs.copysign` (*x2*, *out=None*)

Change the sign of *x1* to that of *x2*, element-wise.

See also:

`numpy.copysign`

ProductSpaceUFuncs.cos

`ProductSpaceUFuncs.cos` (*out=None*)

Cosine element-wise.

See also:

`numpy.cos`

ProductSpaceUFuncs.cosh

`ProductSpaceUFuncs.cosh` (*out=None*)

Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

ProductSpaceUFuncs.deg2rad

`ProductSpaceUFuncs.deg2rad (out=None)`

Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

ProductSpaceUFuncs.divide

`ProductSpaceUFuncs.divide (x2, out=None)`

Returns a true division of the inputs, element-wise.

See also:

`numpy.divide`

ProductSpaceUFuncs.equal

`ProductSpaceUFuncs.equal (x2, out=None)`

Return $(x1 == x2)$ element-wise.

See also:

`numpy.equal`

ProductSpaceUFuncs.exp

`ProductSpaceUFuncs.exp (out=None)`

Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

ProductSpaceUFuncs.exp2

`ProductSpaceUFuncs.exp2 (out=None)`

Calculate 2^{**p} for all p in the input array.

See also:

`numpy.exp2`

ProductSpaceUFuncs.expm1

`ProductSpaceUFuncs.expm1 (out=None)`

Calculate $\exp(x) - 1$ for all elements in the array.

See also:

`numpy.expm1`

ProductSpaceUFuncs.floor

`ProductSpaceUFuncs.floor (out=None)`

Return the floor of the input, element-wise.

See also:

`numpy.floor`

ProductSpaceUFuncs.floor_divide

`ProductSpaceUFuncs.floor_divide` (*x2*, *out=None*)

Return the largest integer smaller or equal to the division of the

See also:

`numpy.floor_divide`

ProductSpaceUFuncs.fmax

`ProductSpaceUFuncs.fmax` (*x2*, *out=None*)

Element-wise maximum of array elements.

See also:

`numpy.fmax`

ProductSpaceUFuncs.fmin

`ProductSpaceUFuncs.fmin` (*x2*, *out=None*)

Element-wise minimum of array elements.

See also:

`numpy.fmin`

ProductSpaceUFuncs.fmod

`ProductSpaceUFuncs.fmod` (*x2*, *out=None*)

Return the element-wise remainder of division.

See also:

`numpy.fmod`

ProductSpaceUFuncs.greater

`ProductSpaceUFuncs.greater` (*x2*, *out=None*)

Return the truth value of (*x1* > *x2*) element-wise.

See also:

`numpy.greater`

ProductSpaceUFuncs.greater_equal

`ProductSpaceUFuncs.greater_equal` (*x2*, *out=None*)

Return the truth value of (*x1* >= *x2*) element-wise.

See also:

`numpy.greater_equal`

ProductSpaceUFuncs.hypot

`ProductSpaceUFuncs.hypot` (*x2*, *out=None*)

Given the “legs” of a right triangle, return its hypotenuse.

See also:

`numpy.hypot`

ProductSpaceUFuncs.invert

`ProductSpaceUFuncs.invert (out=None)`

Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

`numpy.invert`

ProductSpaceUFuncs.isfinite

`ProductSpaceUFuncs.isfinite (out=None)`

Test element-wise for finiteness (not infinity or not Not a Number).

See also:

`numpy.isfinite`

ProductSpaceUFuncs.isinf

`ProductSpaceUFuncs.isinf (out=None)`

Test element-wise for positive or negative infinity.

See also:

`numpy.isinf`

ProductSpaceUFuncs.isnan

`ProductSpaceUFuncs.isnan (out=None)`

Test element-wise for NaN and return result as a boolean array.

See also:

`numpy.isnan`

ProductSpaceUFuncs.left_shift

`ProductSpaceUFuncs.left_shift (x2, out=None)`

Shift the bits of an integer to the left.

See also:

`numpy.left_shift`

ProductSpaceUFuncs.less

`ProductSpaceUFuncs.less (x2, out=None)`

Return the truth value of ($x_1 < x_2$) element-wise.

See also:

`numpy.less`

ProductSpaceUFuncs.less_equal

`ProductSpaceUFuncs.less_equal (x2, out=None)`

Return the truth value of ($x_1 \leq x_2$) element-wise.

See also:

`numpy.less_equal`

ProductSpaceUFuncs.log

`ProductSpaceUFuncs.log` (*out=None*)

Natural logarithm, element-wise.

See also:

`numpy.log`

ProductSpaceUFuncs.log10

`ProductSpaceUFuncs.log10` (*out=None*)

Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

ProductSpaceUFuncs.log1p

`ProductSpaceUFuncs.log1p` (*out=None*)

Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

ProductSpaceUFuncs.log2

`ProductSpaceUFuncs.log2` (*out=None*)

Base-2 logarithm of x.

See also:

`numpy.log2`

ProductSpaceUFuncs.logaddexp

`ProductSpaceUFuncs.logaddexp` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

ProductSpaceUFuncs.logaddexp2

`ProductSpaceUFuncs.logaddexp2` (*x2, out=None*)

Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

ProductSpaceUFuncs.logical_and

`ProductSpaceUFuncs.logical_and` (*x2, out=None*)

Compute the truth value of x1 AND x2 element-wise.

See also:

`numpy.logical_and`

ProductSpaceUFuncs.logical_not

`ProductSpaceUFuncs.logical_not (out=None)`

Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

ProductSpaceUFuncs.logical_or

`ProductSpaceUFuncs.logical_or (x2, out=None)`

Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

ProductSpaceUFuncs.logical_xor

`ProductSpaceUFuncs.logical_xor (x2, out=None)`

Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

ProductSpaceUFuncs.max

`ProductSpaceUFuncs.max ()`

Maximum value in array.

See also:

`numpy.amax`

ProductSpaceUFuncs.maximum

`ProductSpaceUFuncs.maximum (x2, out=None)`

Element-wise maximum of array elements.

See also:

`numpy.maximum`

ProductSpaceUFuncs.min

`ProductSpaceUFuncs.min ()`

Minimum value in array.

See also:

`numpy.amin`

ProductSpaceUFuncs.minimum

`ProductSpaceUFuncs.minimum (x2, out=None)`

Element-wise minimum of array elements.

See also:

`numpy.minimum`

ProductSpaceUFuncs.mod

`ProductSpaceUFuncs.mod(x2, out=None)`

Return element-wise remainder of division.

See also:

`numpy.mod`

ProductSpaceUFuncs.modf

`ProductSpaceUFuncs.modf(out1=None, out2=None)`

Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

ProductSpaceUFuncs.multiply

`ProductSpaceUFuncs.multiply(x2, out=None)`

Multiply arguments element-wise.

See also:

`numpy.multiply`

ProductSpaceUFuncs.negative

`ProductSpaceUFuncs.negative(out=None)`

Numerical negative, element-wise.

See also:

`numpy.negative`

ProductSpaceUFuncs.not_equal

`ProductSpaceUFuncs.not_equal(x2, out=None)`

Return $(x1 \neq x2)$ element-wise.

See also:

`numpy.not_equal`

ProductSpaceUFuncs.power

`ProductSpaceUFuncs.power(x2, out=None)`

First array elements raised to powers from second array, element-wise.

See also:

`numpy.power`

ProductSpaceUFuncs.prod

`ProductSpaceUFuncs.prod()`

Product of array elements.

See also:

`numpy.prod`

ProductSpaceUFuncs.rad2deg

`ProductSpaceUFuncs.rad2deg (out=None)`

Convert angles from radians to degrees.

See also:

`numpy.rad2deg`

ProductSpaceUFuncs.reciprocal

`ProductSpaceUFuncs.reciprocal (out=None)`

Return the reciprocal of the argument, element-wise.

See also:

`numpy.reciprocal`

ProductSpaceUFuncs.remainder

`ProductSpaceUFuncs.remainder (x2, out=None)`

Return element-wise remainder of division.

See also:

`numpy.remainder`

ProductSpaceUFuncs.right_shift

`ProductSpaceUFuncs.right_shift (x2, out=None)`

Shift the bits of an integer to the right.

See also:

`numpy.right_shift`

ProductSpaceUFuncs rint

`ProductSpaceUFuncs.rint (out=None)`

Round elements of the array to the nearest integer.

See also:

`numpy.rint`

ProductSpaceUFuncs.sign

`ProductSpaceUFuncs.sign (out=None)`

Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

ProductSpaceUFuncs.signbit

`ProductSpaceUFuncs.signbit (out=None)`

Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

ProductSpaceUFuncs.sin

`ProductSpaceUFuncs.sin(out=None)`
Trigonometric sine, element-wise.

See also:

`numpy.sin`

ProductSpaceUFuncs.sinh

`ProductSpaceUFuncs.sinh(out=None)`
Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

ProductSpaceUFuncs.sqrt

`ProductSpaceUFuncs.sqrt(out=None)`
Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

ProductSpaceUFuncs.square

`ProductSpaceUFuncs.square(out=None)`
Return the element-wise square of the input.

See also:

`numpy.square`

ProductSpaceUFuncs.subtract

`ProductSpaceUFuncs.subtract(x2, out=None)`
Subtract arguments, element-wise.

See also:

`numpy.subtract`

ProductSpaceUFuncs.sum

`ProductSpaceUFuncs.sum()`
Sum of array elements.

See also:

`numpy.sum`

ProductSpaceUFuncs.tan

`ProductSpaceUFuncs.tan(out=None)`
Compute tangent element-wise.

See also:

`numpy.tan`

ProductSpaceUFuncs.tanh

`ProductSpaceUFuncs.tanh` (*out=None*)
Compute hyperbolic tangent element-wise.

See also:

`numpy.tanh`

ProductSpaceUFuncs.true_divide

`ProductSpaceUFuncs.true_divide` (*x2, out=None*)
Returns a true division of the inputs, element-wise.

See also:

`numpy.true_divide`

ProductSpaceUFuncs.trunc

`ProductSpaceUFuncs.trunc` (*out=None*)
Return the truncated value of the input, element-wise.

See also:

`numpy.trunc`

`__init__` (*vector*)
Create ufunc wrapper for vector.

Functions

<code>method(self)</code>	Maximum value in array.
<code>wrap_reduction_base(name, doc)</code>	Add ufunc methods to <i>NtuplesBaseUFuncs</i> .
<code>wrap_reduction_discretelp(name, doc)</code>	
<code>wrap_reduction_productspace(name, doc)</code>	Add reduction methods to <i>ProductSpaceVector</i> .
<code>wrap_ufunc_base(name, n_in, n_out, doc)</code>	Add ufunc methods to <i>NtuplesBaseUFuncs</i> .
<code>wrap_ufunc_discretelp(name, n_in, n_out, doc)</code>	Add ufunc methods to <i>DiscreteLpUFuncs</i> .
<code>wrap_ufunc_ntuples(name, n_in, n_out, doc)</code>	Add ufunc methods to <i>NtuplesUFuncs</i> .
<code>wrap_ufunc_productspace(name, n_in, n_out, doc)</code>	Add ufunc methods to <i>ProductSpaceVector</i> .

method

`odl.util.ufuncs.method` (*self*)
Maximum value in array.

See also:

`numpy.amax`

wrap_reduction_base

`odl.util.ufuncs.wrap_reduction_base` (*name, doc*)
Add ufunc methods to *NtuplesBaseUFuncs*.

wrap_reduction_discretelp

`odl.util.ufuncs.wrap_reduction_discretelp` (*name, doc*)

wrap_reduction_productspace

`odl.util.ufuncs.wrap_reduction_productspace(name, doc)`

Add reduction methods to *ProductSpaceVector*.

wrap_ufunc_base

`odl.util.ufuncs.wrap_ufunc_base(name, n_in, n_out, doc)`

Add ufunc methods to *NtuplesBaseUFuncs*.

wrap_ufunc_discretelp

`odl.util.ufuncs.wrap_ufunc_discretelp(name, n_in, n_out, doc)`

Add ufunc methods to *DiscreteLpUFuncs*.

wrap_ufunc_ntuples

`odl.util.ufuncs.wrap_ufunc_ntuples(name, n_in, n_out, doc)`

Add ufunc methods to *NtuplesUFuncs*.

wrap_ufunc_productspace

`odl.util.ufuncs.wrap_ufunc_productspace(name, n_in, n_out, doc)`

Add ufunc methods to *ProductSpaceVector*.

utility

Utilities for internal use.

Functions

<code>array1d_repr(array[, nprint])</code>	Stringification of a 1D array, keeping byte / unicode.
<code>array1d_str(array[, nprint])</code>	Stringification of a 1D array, regardless of byte or unicode.
<code>arraynd_repr(array[, nprint])</code>	Stringification of an nD array, keeping byte / unicode.
<code>arraynd_str(array[, nprint])</code>	Stringification of an nD array, regardless of byte or unicode.
<code>conj_exponent(exp)</code>	The conjugate exponent $p / (p-1)$.
<code>dtype_repr(dtype)</code>	Stringification of data type with default for int and float.
<code>is_complex_floating_dtype(dtype)</code>	True if dtype is complex floating-point, else False.
<code>is_floating_dtype(dtype)</code>	True if dtype is floating-point, else False.
<code>is_int_dtype(dtype)</code>	True if dtype is integer, else False.
<code>is_real_dtype(dtype)</code>	True if dtype is real (including integer), else False.
<code>is_real_floating_dtype(dtype)</code>	True if dtype is real floating-point, else False.
<code>is_scalar_dtype(dtype)</code>	True if dtype is scalar, else False.
<code>preload_first_arg(instance, mode)</code>	Decorator to preload the first argument of a call method.
<code>with_metaclass(meta, *bases)</code>	Function from <code>jinja2/_compat.py</code> .

array1d_repr

`odl.util.utility.array1d_repr(array, nprint=6)`

Stringification of a 1D array, keeping byte / unicode.

Parameters`array` : array-like

The array to print

nprint : int

Maximum number of elements to print

array1d_str

`odl.util.utility.array1d_str(array, nprint=6)`
Stringification of a 1D array, regardless of byte or unicode.

Parameters`array` : array-like

The array to print

nprint : int

Maximum number of elements to print

arraynd_repr

`odl.util.utility.arraynd_repr(array, nprint=None)`
Stringification of an nD array, keeping byte / unicode.

Parameters`array` : array-like

The array to print

nprint : int

Maximum number of elements to print. Default: 6 if `array.ndim <= 2`, else 2

Examples

```
>>> print(arraynd_repr([[1, 2, 3], [4, 5, 6]]))
[[1, 2, 3],
 [4, 5, 6]]
>>> print(arraynd_repr([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

arraynd_str

`odl.util.utility.arraynd_str(array, nprint=None)`
Stringification of an nD array, regardless of byte or unicode.

Parameters`array` : *array-like*

The array to print

nprint : int

Maximum number of elements to print. Default: 6 if `array.ndim <= 2`, else 2

Examples

```
>>> print(arraynd_str([[1, 2, 3], [4, 5, 6]]))
[[1, 2, 3],
 [4, 5, 6]]
>>> print(arraynd_str([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

conj_exponent

`odl.util.utility.conj_exponent(exp)`

The conjugate exponent $p / (p-1)$.

Parameter`sexp` : positive float or inf

Exponent for which to calculate the conjugate. Must be at least 1.0.

Returns`conj` : positive float or inf

Conjugate exponent. For `exp=1`, return `float('inf')`, for `exp=float('inf')` return 1. In all other cases, return `exp / (exp - 1)`.

dtype_repr

`odl.util.utility.dtype_repr(dtype)`

Stringification of data type with default for int and float.

is_complex_floating_dtype

`odl.util.utility.is_complex_floating_dtype(dtype)`

True if dtype is complex floating-point, else False.

is_floating_dtype

`odl.util.utility.is_floating_dtype(dtype)`

True if dtype is floating-point, else False.

is_int_dtype

`odl.util.utility.is_int_dtype(dtype)`

True if dtype is integer, else False.

is_real_dtype

`odl.util.utility.is_real_dtype(dtype)`

True if dtype is real (including integer), else False.

is_real_floating_dtype

`odl.util.utility.is_real_floating_dtype(dtype)`
True if dtype is real floating-point, else False.

is_scalar_dtype

`odl.util.utility.is_scalar_dtype(dtype)`
True if dtype is scalar, else False.

preload_first_arg

`odl.util.utility.preload_first_arg(instance, mode)`
Decorator to preload the first argument of a call method.

Parametersinstance :

Class instance to preload the call with

mode : {'out-of-place', 'in-place'}

'out-of-place': call is out-of-place – `f(x, **kwargs)`

'in-place': call is in-place – `f(x, out, **kwargs)`

Notes

The decorated function has the signature according to mode.

Examples

Define two functions which need some instance to act on and decorate them manually:

```
>>> class A(object):
...     '''My name is A.'''
>>> a = A()
...
>>> def f_oop(inst, x):
...     print(inst.__doc__)
...
>>> def f_ip(inst, out, x):
...     print(inst.__doc__)
...
>>> f_oop_new = preload_first_arg(a, 'out-of-place')(f_oop)
>>> f_ip_new = preload_first_arg(a, 'in-place')(f_ip)
...
>>> f_oop_new(0)
My name is A.
>>> f_ip_new(0, out=1)
My name is A.
```

Decorate upon definition:

```
>>> @preload_first_arg(a, 'out-of-place')
... def set_x(obj, x):
...     '''Function to set x in ``obj`` to a given value.'''
...     obj.x = x
>>> set_x(0)
>>> a.x
0
```

The function's name and docstring are preserved:

```
>>> set_x.__name__
'set_x'
>>> set_x.__doc__
'Function to set x in ``obj`` to a given value.'
```

with_metaclass

`odl.util.utility.with_metaclass(meta, *bases)`

Function from `jinja2/_compat.py`. License: BSD.

Use it like this:

```
class BaseForm(object):
    pass

class FormType(type):
    pass

class Form(with_metaclass(FormType, BaseForm)):
    pass
```

This requires a bit of explanation: the basic idea is to make a dummy metaclass for one level of class instantiation that replaces itself with the actual metaclass. Because of internal type checks we also need to make sure that we downgrade the custom metaclass for one level to something closer to type (that's why `__call__` and `__init__` comes back from type etc.).

This has the advantage over `six.with_metaclass` of not introducing dummy classes into the final MRO.

vectorization

Utilities for internal functionality connected to vectorization.

Classes

<i>OptionalArgDecorator</i>	Abstract class to create decorators with optional arguments.
<i>vectorize</i>	Decorator class for function vectorization.

OptionalArgDecorator

`class odl.util.vectorization.OptionalArgDecorator`

Bases: `object`

Abstract class to create decorators with optional arguments.

This class implements the functionality of a decorator that can be used with and without arguments, i.e. the following patterns both work:

```
@decorator
def myfunc(x, *args, **kwargs):
    pass

@decorator(param, **dec_kwargs)
def myfunc(x, *args, **kwargs):
    pass
```

The arguments to the decorator are passed on to the underlying wrapper.

To use this class, subclass it and implement the static `_wrapper` method.

Methods

<code>__call__(func)</code>	Return <code>self(func)</code> .
<code>__eq__</code>	Return <code>self==value</code> .

OptionalArgDecorator.`__call__`

OptionalArgDecorator.`__call__`(*func*)
Return `self(func)`.

This method is invoked when the decorator was created with arguments.

Parameters`func`: callable

Original function to be wrapped

Returns`wrapped`: callable

The wrapped function

vectorize

class odl.util.vectorization.**vectorize**

Bases: *odl.util.vectorization.OptionalArgDecorator*

Decorator class for function vectorization.

This vectorizer expects a function with exactly one positional argument (input) and optional keyword arguments. The decorated function has an optional `out` parameter for in-place evaluation.

Examples

Use the decorator without arguments:

```
>>> @vectorize
... def f(x):
...     return x[0] + x[1] if x[0] < x[1] else x[0] - x[1]
>>>
>>> f([0, 1]) # np.vectorize'd functions always return an array
array(1)
>>> f([[0, -2], [1, 4]]) # corresponds to points [0, 1], [-2, 4]
array([1, 2])
```

The function may have kwargs:

```
>>> @vectorize
... def f(x, param=1.0):
...     return x[0] + x[1] if x[0] < param else x[0] - x[1]
>>>
>>> f([[0, -2], [1, 4]])
array([1, 2])
>>> f([[0, -2], [1, 4]], param=-1.0)
array([-1, 2])
```

You can pass arguments to the vectorizer, too:

```
>>> @vectorize(otypes=['float32'])
... def f(x):
...     return x[0] + x[1] if x[0] < x[1] else x[0] - x[1]
>>> f([[0, -2], [1, 4]])
array([ 1.,  2.], dtype=float32)
```

Methods

<code>__call__(func)</code>	Return <code>self(func)</code> .
<code>__eq__</code>	Return <code>self==value</code> .

`vectorize.__call__`

`vectorize.__call__(func)`
Return `self(func)`.

This method is invoked when the decorator was created with arguments.

Parameters`func`: callable

Original function to be wrapped

Returns`wrapped`: callable

The wrapped function

Functions

<code>is_valid_input_array(x[, ndim])</code>	Test if <code>x</code> is a correctly shaped point array in R^d .
<code>is_valid_input_meshgrid(x, ndim)</code>	Test if <code>x</code> is a <i>meshgrid</i> sequence for points in R^d .
<code>out_shape_from_array(arr)</code>	Get the output shape from an array.
<code>out_shape_from_meshgrid(mesh)</code>	Get the broadcast output shape from a <i>meshgrid</i> .

`is_valid_input_array`

`odl.util.vectorization.is_valid_input_array(x, ndim=None)`
Test if `x` is a correctly shaped point array in R^d .

is_valid_input_meshgrid

`odl.util.vectorization.is_valid_input_meshgrid(x, ndim)`
Test if `x` is a *meshgrid* sequence for points in \mathbb{R}^d .

out_shape_from_array

`odl.util.vectorization.out_shape_from_array(arr)`
Get the output shape from an array.

out_shape_from_meshgrid

`odl.util.vectorization.out_shape_from_meshgrid(mesh)`
Get the broadcast output shape from a *meshgrid*.

Indices and tables

- `genindex`
- `modindex`
- `search`

- [BB1988] Barzilai, J, and Borwein, J M. *Two-point step size gradient methods*. IMA Journal of Numerical Analysis, 8 (1988), pp 141–148.
- [Bro1965] Broyden, C G. *A class of methods for solving nonlinear simultaneous equations*. Mathematics of computation, 33 (1965), pp 577–593.
- [BV2004] Boyd, S, and Vandenberghe, L. *Convex optimization*. Cambridge university press, 2004.
- [CP2011a] Chambolle, Antonin and Pock, Thomas. *A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging*. Journal of Mathematical Imaging and Vision, 40 (2011), pp 120-145.
- [CP2011b] Chambolle, Antonin and Pock, Thomas. *Diagonal preconditioning for first order primal-dual algorithms in convex optimization*. 2011 IEEE International Conference on Computer Vision (ICCV), 2011, pp 1762-1769.
- [GNS2009] Griva, I, Nash, S G, and Sofer, A. *Linear and nonlinear optimization*. Siam, 2009.
- [Kva1991] Kvaalen, E. *A faster Broyden method*. BIT Numerical Mathematics 31 (1991), pp 369–372.
- [Okt2015] Oktem, O. *Mathematics of electron tomography*. In: Scherzer, O. Handbook of Mathematical Methods in Imaging. Springer, 2015, pp 937–1031.
- [PB2014] Parikh, Neal and Boyd, Stephen. *Proximal Algorithms*. Foundations and Trends in Optimization, 1 (2014), pp 127-239.
- [Pre+2007] Press, W H, Teukolsky, S A, Vetterling, W T, and Flannery, B P. *Numerical Recipes in C - The Art of Scientific Computing* (Volume 3). Cambridge University Press, 2007.
- [Ray1997] Raydan, M. *The Barzilai and Borwein method for the large scale unconstrained minimization problem*. SIAM J. Optim., 7 (1997), pp 26–33.
- [Roc1970] Rockafellar, R. Tyrrell. *Convex analysis*. Princeton University Press, 1970.
- [Sid+2012] Sidky, Emil Y, Jorgensen, Jakob H, and Pan, Xiaochuan. *Convex optimization problem prototyping for image reconstruction in computed tomography with the Chambolle-Pock algorithm*. Physics in Medicine and Biology, 57 (2012), pp 3065-3091.
- [SW1971] Stein, E and Weiss, G. *Introduction to Fourier Analysis on Euclidean Spaces*. Princeton University Press, 1971.

Symbols

- `__call__()` (odl.dscr.dscr_mappings.FunctionSetMapping method), 49
- `__call__()` (odl.dscr.dscr_mappings.LinearInterpolation method), 53
- `__call__()` (odl.dscr.dscr_mappings.NearestInterpolation method), 56
- `__call__()` (odl.dscr.dscr_mappings.PerAxisInterpolation method), 61
- `__call__()` (odl.dscr.dscr_mappings.PointCollocation method), 64
- `__call__()` (odl.dscr.dscr_ops.Divergence method), 68
- `__call__()` (odl.dscr.dscr_ops.Gradient method), 71
- `__call__()` (odl.dscr.dscr_ops.Laplacian method), 74
- `__call__()` (odl.dscr.dscr_ops.PartialDerivative method), 78
- `__call__()` (odl.operator.default_ops.ConstantOperator method), 153
- `__call__()` (odl.operator.default_ops.IdentityOperator method), 156
- `__call__()` (odl.operator.default_ops.InnerProductOperator method), 159
- `__call__()` (odl.operator.default_ops.LinCombOperator method), 162
- `__call__()` (odl.operator.default_ops.MultiplyOperator method), 165
- `__call__()` (odl.operator.default_ops.ResidualOperator method), 168
- `__call__()` (odl.operator.default_ops.ScalingOperator method), 171
- `__call__()` (odl.operator.default_ops.ZeroOperator method), 174
- `__call__()` (odl.operator.operator.FunctionalLeftVectorMult method), 177
- `__call__()` (odl.operator.operator.Operator method), 182
- `__call__()` (odl.operator.operator.OperatorComp method), 185
- `__call__()` (odl.operator.operator.OperatorLeftScalarMult method), 188
- `__call__()` (odl.operator.operator.OperatorLeftVectorMult method), 190
- `__call__()` (odl.operator.operator.OperatorPointwiseProduct method), 193
- `__call__()` (odl.operator.operator.OperatorRightScalarMult method), 195
- `__call__()` (odl.operator.operator.OperatorRightVectorMult method), 198
- `__call__()` (odl.operator.operator.OperatorSum method), 200
- `__call__()` (odl.operator.pspace_ops.BroadcastOperator method), 206
- `__call__()` (odl.operator.pspace_ops.ComponentProjection method), 209
- `__call__()` (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 213
- `__call__()` (odl.operator.pspace_ops.ProductSpaceOperator method), 217
- `__call__()` (odl.operator.pspace_ops.ReductionOperator method), 222
- `__call__()` (odl.solvers.scalar.steplen.BacktrackingLineSearch method), 283
- `__call__()` (odl.solvers.scalar.steplen.BarzilaiBorweinStep method), 284
- `__call__()` (odl.solvers.scalar.steplen.ConstantLineSearch method), 285
- `__call__()` (odl.solvers.scalar.steplen.LineSearch method), 285
- `__call__()` (odl.solvers.scalar.steplen.StepLength method), 286
- `__call__()` (odl.solvers.util.partial.AndPartial method), 287
- `__call__()` (odl.solvers.util.partial.ForEachPartial method), 287
- `__call__()` (odl.solvers.util.partial.Partial method), 288
- `__call__()` (odl.solvers.util.partial.PrintIterationPartial method), 288
- `__call__()` (odl.solvers.util.partial.PrintNormPartial method), 289
- `__call__()` (odl.solvers.util.partial.PrintTimingPartial method), 289

`__call__()` (odl.solvers.util.partial.ShowPartial method), 289
`__call__()` (odl.solvers.util.partial.StorePartial method), 290
`__call__()` (odl.space.fspace.FunctionSetVector method), 354
`__call__()` (odl.space.fspace.FunctionSpaceVector method), 364
`__call__()` (odl.space.ntuples.MatVecOperator method), 402
`__call__()` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper method), 424
`__call__()` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper method), 427
`__call__()` (odl.tomo.operators.ray_trafo.RayBackProjection method), 477
`__call__()` (odl.tomo.operators.ray_trafo.RayTransform method), 479
`__call__()` (odl.trafos.fourier.DiscreteFourierTransform method), 485
`__call__()` (odl.trafos.fourier.DiscreteFourierTransformInverse method), 490
`__call__()` (odl.trafos.fourier.FourierTransform method), 494
`__call__()` (odl.trafos.fourier.FourierTransformInverse method), 499
`__call__()` (odl.trafos.wavelet.WaveletTransform method), 511
`__call__()` (odl.trafos.wavelet.WaveletTransformInverse method), 514
`__call__()` (odl.util.vectorization.OptionalArgDecorator method), 602
`__call__()` (odl.util.vectorization.vectorize method), 603
`__contains__()` (odl.discr.discretization.Discretization method), 84
`__contains__()` (odl.discr.discretization.RawDiscretization method), 96
`__contains__()` (odl.discr.grid.RegularGrid method), 105
`__contains__()` (odl.discr.grid.TensorGrid method), 114
`__contains__()` (odl.discr.lp_discr.DiscreteLp method), 126
`__contains__()` (odl.set.domain.IntervalProd method), 226
`__contains__()` (odl.set.pspace.ProductSpace method), 236
`__contains__()` (odl.set.sets.CartesianProduct method), 247
`__contains__()` (odl.set.sets.ComplexNumbers method), 249
`__contains__()` (odl.set.sets.EmptySet method), 250
`__contains__()` (odl.set.sets.Field method), 251
`__contains__()` (odl.set.sets.Integers method), 252
`__contains__()` (odl.set.sets.RealNumbers method), 253
`__contains__()` (odl.set.sets.Set method), 255
`__contains__()` (odl.set.sets.Strings method), 256
`__contains__()` (odl.set.sets.UniversalSet method), 257
`__contains__()` (odl.set.space.LinearSpace method), 259
`__contains__()` (odl.set.space.UniversalSpace method), 267
`__contains__()` (odl.space.base_ntuples.FnBase method), 293
`__contains__()` (odl.space.base_ntuples.NtuplesBase method), 305
`__contains__()` (odl.space.cu_ntuples.CudaFn method), 312
`__contains__()` (odl.space.cu_ntuples.CudaNtuples method), 339
`__contains__()` (odl.space.fspace.FunctionSet method), 351
`__contains__()` (odl.space.fspace.FunctionSpace method), 357
`__contains__()` (odl.space.ntuples.Fn method), 368
`__contains__()` (odl.space.ntuples.Ntuples method), 405
`__eq__()` (odl.discr.discr_mappings.FunctionSetMapping method), 50
`__eq__()` (odl.discr.discr_mappings.LinearInterpolation method), 54
`__eq__()` (odl.discr.discr_mappings.NearestInterpolation method), 57
`__eq__()` (odl.discr.discr_mappings.PerAxisInterpolation method), 62
`__eq__()` (odl.discr.discr_mappings.PointCollocation method), 65
`__eq__()` (odl.discr.discretization.Discretization method), 85
`__eq__()` (odl.discr.discretization.DiscretizationVector method), 91
`__eq__()` (odl.discr.discretization.RawDiscretization method), 96
`__eq__()` (odl.discr.discretization.RawDiscretizationVector method), 99
`__eq__()` (odl.discr.grid.RegularGrid method), 105
`__eq__()` (odl.discr.grid.TensorGrid method), 114
`__eq__()` (odl.discr.lp_discr.DiscreteLp method), 127
`__eq__()` (odl.discr.lp_discr.DiscreteLpVector method), 135
`__eq__()` (odl.discr.partition.RectPartition method), 147
`__eq__()` (odl.set.domain.IntervalProd method), 227
`__eq__()` (odl.set.pspace.ProductSpace method), 236
`__eq__()` (odl.set.pspace.ProductSpaceVector method), 245
`__eq__()` (odl.set.sets.CartesianProduct method), 248
`__eq__()` (odl.set.sets.ComplexNumbers method), 249
`__eq__()` (odl.set.sets.EmptySet method), 250
`__eq__()` (odl.set.sets.Field method), 251
`__eq__()` (odl.set.sets.Integers method), 252
`__eq__()` (odl.set.sets.RealNumbers method), 253
`__eq__()` (odl.set.sets.Set method), 255

- `__eq__()` (odl.set.sets.Strings method), 256
- `__eq__()` (odl.set.sets.UniversalSet method), 257
- `__eq__()` (odl.set.space.LinearSpace method), 259
- `__eq__()` (odl.set.space.LinearSpaceVector method), 264
- `__eq__()` (odl.set.space.UniversalSpace method), 267
- `__eq__()` (odl.space.base_ntuples.FnBase method), 294
- `__eq__()` (odl.space.base_ntuples.FnBaseVector method), 300
- `__eq__()` (odl.space.base_ntuples.FnWeightingBase method), 303
- `__eq__()` (odl.space.base_ntuples.NtuplesBase method), 306
- `__eq__()` (odl.space.base_ntuples.NtuplesBaseVector method), 308
- `__eq__()` (odl.space.cu_ntuples.CudaFn method), 313
- `__eq__()` (odl.space.cu_ntuples.CudaFnConstWeighting method), 322
- `__eq__()` (odl.space.cu_ntuples.CudaFnCustomDist method), 323
- `__eq__()` (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 325
- `__eq__()` (odl.space.cu_ntuples.CudaFnCustomNorm method), 327
- `__eq__()` (odl.space.cu_ntuples.CudaFnNoWeighting method), 328
- `__eq__()` (odl.space.cu_ntuples.CudaFnVector method), 332
- `__eq__()` (odl.space.cu_ntuples.CudaFnVectorWeighting method), 337
- `__eq__()` (odl.space.cu_ntuples.CudaNtuples method), 340
- `__eq__()` (odl.space.cu_ntuples.CudaNtuplesVector method), 344
- `__eq__()` (odl.space.fspace.FunctionSet method), 352
- `__eq__()` (odl.space.fspace.FunctionSetVector method), 355
- `__eq__()` (odl.space.fspace.FunctionSpace method), 357
- `__eq__()` (odl.space.fspace.FunctionSpaceVector method), 364
- `__eq__()` (odl.space.ntuples.Fn method), 369
- `__eq__()` (odl.space.ntuples.FnConstWeighting method), 379
- `__eq__()` (odl.space.ntuples.FnCustomDist method), 381
- `__eq__()` (odl.space.ntuples.FnCustomInnerProduct method), 382
- `__eq__()` (odl.space.ntuples.FnCustomNorm method), 384
- `__eq__()` (odl.space.ntuples.FnMatrixWeighting method), 386
- `__eq__()` (odl.space.ntuples.FnNoWeighting method), 389
- `__eq__()` (odl.space.ntuples.FnVector method), 394
- `__eq__()` (odl.space.ntuples.FnVectorWeighting method), 400
- `__eq__()` (odl.space.ntuples.Ntuples method), 405
- `__eq__()` (odl.space.ntuples.NtuplesVector method), 410
- `__getitem__()` (odl.dscr.discretization.DiscretizationVector method), 91
- `__getitem__()` (odl.dscr.discretization.RawDiscretizationVector method), 99
- `__getitem__()` (odl.dscr.grid.RegularGrid method), 105
- `__getitem__()` (odl.dscr.grid.TensorGrid method), 115
- `__getitem__()` (odl.dscr.lp_dscr.DiscreteLpVector method), 135
- `__getitem__()` (odl.set.domain.IntervalProd method), 227
- `__getitem__()` (odl.set.pspace.ProductSpace method), 237
- `__getitem__()` (odl.set.pspace.ProductSpaceVector method), 245
- `__getitem__()` (odl.set.sets.CartesianProduct method), 248
- `__getitem__()` (odl.solvers.util.partial.StorePartial method), 290
- `__getitem__()` (odl.space.base_ntuples.FnBaseVector method), 300
- `__getitem__()` (odl.space.base_ntuples.NtuplesBaseVector method), 308
- `__getitem__()` (odl.space.cu_ntuples.CudaFnVector method), 332
- `__getitem__()` (odl.space.cu_ntuples.CudaNtuplesVector method), 345
- `__getitem__()` (odl.space.ntuples.FnVector method), 394
- `__getitem__()` (odl.space.ntuples.NtuplesVector method), 410
- `__init__()` (odl.diagnostics.operator.OperatorTest method), 44
- `__init__()` (odl.diagnostics.space.SpaceTest method), 47
- `__init__()` (odl.dscr.dscr_mappings.FunctionSetMapping method), 51
- `__init__()` (odl.dscr.dscr_mappings.LinearInterpolation method), 54
- `__init__()` (odl.dscr.dscr_mappings.NearestInterpolation method), 58
- `__init__()` (odl.dscr.dscr_mappings.PerAxisInterpolation method), 62
- `__init__()` (odl.dscr.dscr_mappings.PointCollocation method), 66
- `__init__()` (odl.dscr.dscr_ops.Divergence method), 70
- `__init__()` (odl.dscr.dscr_ops.Gradient method), 73
- `__init__()` (odl.dscr.dscr_ops.Laplacian method), 75
- `__init__()` (odl.dscr.dscr_ops.PartialDerivative method), 79
- `__init__()` (odl.dscr.discretization.Discretization method), 88
- `__init__()` (odl.dscr.discretization.DiscretizationVector method), 94
- `__init__()` (odl.dscr.discretization.RawDiscretization method), 97

[__init__\(\) \(odl.discr.discretization.RawDiscretizationVector method\), 101](#)
[__init__\(\) \(odl.discr.grid.RegularGrid method\), 111](#)
[__init__\(\) \(odl.discr.grid.TensorGrid method\), 120](#)
[__init__\(\) \(odl.discr.lp_discr.DiscreteLp method\), 130](#)
[__init__\(\) \(odl.discr.lp_discr.DiscreteLpVector method\), 138](#)
[__init__\(\) \(odl.discr.partition.RectPartition method\), 148](#)
[__init__\(\) \(odl.operator.default_ops.ConstantOperator method\), 155](#)
[__init__\(\) \(odl.operator.default_ops.IdentityOperator method\), 158](#)
[__init__\(\) \(odl.operator.default_ops.InnerProductOperator method\), 161](#)
[__init__\(\) \(odl.operator.default_ops.LinCombOperator method\), 163](#)
[__init__\(\) \(odl.operator.default_ops.MultiplyOperator method\), 167](#)
[__init__\(\) \(odl.operator.default_ops.ResidualOperator method\), 170](#)
[__init__\(\) \(odl.operator.default_ops.ScalingOperator method\), 173](#)
[__init__\(\) \(odl.operator.default_ops.ZeroOperator method\), 176](#)
[__init__\(\) \(odl.operator.operator.FunctionalLeftVectorMult method\), 178](#)
[__init__\(\) \(odl.operator.operator.Operator method\), 183](#)
[__init__\(\) \(odl.operator.operator.OperatorComp method\), 186](#)
[__init__\(\) \(odl.operator.operator.OperatorLeftScalarMult method\), 189](#)
[__init__\(\) \(odl.operator.operator.OperatorLeftVectorMult method\), 191](#)
[__init__\(\) \(odl.operator.operator.OperatorPointwiseProduct method\), 194](#)
[__init__\(\) \(odl.operator.operator.OperatorRightScalarMult method\), 196](#)
[__init__\(\) \(odl.operator.operator.OperatorRightVectorMult method\), 199](#)
[__init__\(\) \(odl.operator.operator.OperatorSum method\), 202](#)
[__init__\(\) \(odl.operator.pspace_ops.BroadcastOperator method\), 208](#)
[__init__\(\) \(odl.operator.pspace_ops.ComponentProjection method\), 211](#)
[__init__\(\) \(odl.operator.pspace_ops.ComponentProjectionAdjoint method\), 215](#)
[__init__\(\) \(odl.operator.pspace_ops.ProductSpaceOperator method\), 220](#)
[__init__\(\) \(odl.operator.pspace_ops.ReductionOperator method\), 224](#)
[__init__\(\) \(odl.set.domain.IntervalProd method\), 233](#)
[__init__\(\) \(odl.set.pspace.ProductSpace method\), 241](#)
[__init__\(\) \(odl.set.pspace.ProductSpaceVector method\), 247](#)
[__init__\(\) \(odl.set.sets.CartesianProduct method\), 248](#)
[__init__\(\) \(odl.set.sets.Strings method\), 256](#)
[__init__\(\) \(odl.set.space.LinearSpace method\), 262](#)
[__init__\(\) \(odl.set.space.LinearSpaceVector method\), 266](#)
[__init__\(\) \(odl.set.space.UniversalSpace method\), 270](#)
[__init__\(\) \(odl.solvers.scalar.steplen.BacktrackingLineSearch method\), 284](#)
[__init__\(\) \(odl.solvers.scalar.steplen.BarzilaiBorweinStep method\), 284](#)
[__init__\(\) \(odl.solvers.scalar.steplen.ConstantLineSearch method\), 285](#)
[__init__\(\) \(odl.solvers.util.partial.AndPartial method\), 287](#)
[__init__\(\) \(odl.solvers.util.partial.ForEachPartial method\), 287](#)
[__init__\(\) \(odl.solvers.util.partial.PrintIterationPartial method\), 288](#)
[__init__\(\) \(odl.solvers.util.partial.PrintNormPartial method\), 289](#)
[__init__\(\) \(odl.solvers.util.partial.PrintTimingPartial method\), 289](#)
[__init__\(\) \(odl.solvers.util.partial.ShowPartial method\), 289](#)
[__init__\(\) \(odl.solvers.util.partial.StorePartial method\), 290](#)
[__init__\(\) \(odl.space.base_ntuples.FnBase method\), 298](#)
[__init__\(\) \(odl.space.base_ntuples.FnBaseVector method\), 302](#)
[__init__\(\) \(odl.space.base_ntuples.FnWeightingBase method\), 304](#)
[__init__\(\) \(odl.space.base_ntuples.NtuplesBase method\), 307](#)
[__init__\(\) \(odl.space.base_ntuples.NtuplesBaseVector method\), 310](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFn method\), 319](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnConstWeighting method\), 323](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnCustomDist method\), 324](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnCustomInnerProduct method\), 326](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnCustomNorm method\), 327](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnNoWeighting method\), 329](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnVector method\), 336](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaFnVectorWeighting method\), 338](#)
[__init__\(\) \(odl.space.cu_ntuples.CudaNtuples method\), 341](#)

`__init__()` (odl.space.cu_ntuples.CudaNtuplesVector method), 349
`__init__()` (odl.space.fspace.FunctionSet method), 352
`__init__()` (odl.space.fspace.FunctionSetVector method), 355
`__init__()` (odl.space.fspace.FunctionSpace method), 361
`__init__()` (odl.space.fspace.FunctionSpaceVector method), 366
`__init__()` (odl.space.ntuples.Fn method), 376
`__init__()` (odl.space.ntuples.FnConstWeighting method), 380
`__init__()` (odl.space.ntuples.FnCustomDist method), 382
`__init__()` (odl.space.ntuples.FnCustomInnerProduct method), 383
`__init__()` (odl.space.ntuples.FnCustomNorm method), 385
`__init__()` (odl.space.ntuples.FnMatrixWeighting method), 388
`__init__()` (odl.space.ntuples.FnNoWeighting method), 390
`__init__()` (odl.space.ntuples.FnVector method), 398
`__init__()` (odl.space.ntuples.FnVectorWeighting method), 401
`__init__()` (odl.space.ntuples.MatVecOperator method), 403
`__init__()` (odl.space.ntuples.Ntuples method), 407
`__init__()` (odl.space.ntuples.NtuplesVector method), 413
`__init__()` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapping method), 425
`__init__()` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapping method), 428
`__init__()` (odl.tomo.backends.stir_bindings.StirVerbosity method), 429
`__init__()` (odl.tomo.geometry.conebeam.CircularConeFlatGeometry method), 434
`__init__()` (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 438
`__init__()` (odl.tomo.geometry.detector.CircleSectionDetector method), 441
`__init__()` (odl.tomo.geometry.detector.Detector method), 443
`__init__()` (odl.tomo.geometry.detector.Flat1dDetector method), 445
`__init__()` (odl.tomo.geometry.detector.Flat2dDetector method), 447
`__init__()` (odl.tomo.geometry.detector.FlatDetector method), 449
`__init__()` (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 453
`__init__()` (odl.tomo.geometry.geometry.AxisOrientedGeometry method), 454
`__init__()` (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 457
`__init__()` (odl.tomo.geometry.geometry.Geometry method), 461
`__init__()` (odl.tomo.geometry.parallel.Parallel2dGeometry method), 464
`__init__()` (odl.tomo.geometry.parallel.Parallel3dAxisGeometry method), 468
`__init__()` (odl.tomo.geometry.parallel.Parallel3dGeometry method), 471
`__init__()` (odl.tomo.geometry.parallel.ParallelGeometry method), 475
`__init__()` (odl.tomo.operators.ray_trafo.RayBackProjection method), 478
`__init__()` (odl.tomo.operators.ray_trafo.RayTransform method), 481
`__init__()` (odl.trafos.fourier.DiscreteFourierTransform method), 487
`__init__()` (odl.trafos.fourier.DiscreteFourierTransformInverse method), 492
`__init__()` (odl.trafos.fourier.FourierTransform method), 497
`__init__()` (odl.trafos.fourier.FourierTransformInverse method), 502
`__init__()` (odl.trafos.wavelet.WaveletTransform method), 512
`__init__()` (odl.trafos.wavelet.WaveletTransformInverse method), 515
`__init__()` (odl.util.testutils.FailCounter method), 529
`__init__()` (odl.util.testutils.ProgressBar method), 530
`__init__()` (odl.util.testutils.ProgressRange method), 530
`__init__()` (odl.util.testutils.Timer method), 530
`__init__()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
`__init__()` (odl.util.ufuncs.DiscreteLpUFuncs method), 558
`__init__()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 571
`__init__()` (odl.util.ufuncs.NtuplesUFuncs method), 583
`__init__()` (odl.util.ufuncs.ProductSpaceUFuncs method), 596
`__setitem__()` (odl.discr.discretization.DiscretizationVector method), 92
`__setitem__()` (odl.discr.discretization.RawDiscretizationVector method), 100
`__setitem__()` (odl.discr.lp_discr.DiscreteLpVector method), 135
`__setitem__()` (odl.set.pspace.ProductSpaceVector method), 245
`__setitem__()` (odl.space.base_ntuples.FnBaseVector method), 300
`__setitem__()` (odl.space.base_ntuples.NtuplesBaseVector method), 308
`__setitem__()` (odl.space.cu_ntuples.CudaFnVector method), 333

- `__setitem__()` (odl.space.cu_ntuples.CudaNtuplesVector method), 345
- `__setitem__()` (odl.space.ntuples.FnVector method), 395
- `__setitem__()` (odl.space.ntuples.NtuplesVector method), 411
- `_call()` (odl.dscr.dscr_mappings.FunctionSetMapping method), 50
- `_call()` (odl.dscr.dscr_mappings.LinearInterpolation method), 54
- `_call()` (odl.dscr.dscr_mappings.NearestInterpolation method), 57
- `_call()` (odl.dscr.dscr_mappings.PerAxisInterpolation method), 62
- `_call()` (odl.dscr.dscr_mappings.PointCollocation method), 65
- `_call()` (odl.dscr.dscr_ops.Divergence method), 69
- `_call()` (odl.dscr.dscr_ops.Gradient method), 72
- `_call()` (odl.dscr.dscr_ops.Laplacian method), 75
- `_call()` (odl.dscr.dscr_ops.PartialDerivative method), 79
- `_call()` (odl.operator.default_ops.ConstantOperator method), 154
- `_call()` (odl.operator.default_ops.IdentityOperator method), 157
- `_call()` (odl.operator.default_ops.InnerProductOperator method), 160
- `_call()` (odl.operator.default_ops.LinCombOperator method), 163
- `_call()` (odl.operator.default_ops.MultiplyOperator method), 166
- `_call()` (odl.operator.default_ops.ResidualOperator method), 169
- `_call()` (odl.operator.default_ops.ScalingOperator method), 172
- `_call()` (odl.operator.default_ops.ZeroOperator method), 175
- `_call()` (odl.operator.operator.FunctionalLeftVectorMult method), 178
- `_call()` (odl.operator.operator.Operator method), 183
- `_call()` (odl.operator.operator.OperatorComp method), 186
- `_call()` (odl.operator.operator.OperatorLeftScalarMult method), 189
- `_call()` (odl.operator.operator.OperatorLeftVectorMult method), 191
- `_call()` (odl.operator.operator.OperatorPointwiseProduct method), 194
- `_call()` (odl.operator.operator.OperatorRightScalarMult method), 196
- `_call()` (odl.operator.operator.OperatorRightVectorMult method), 199
- `_call()` (odl.operator.operator.OperatorSum method), 201
- `_call()` (odl.operator.pspace_ops.BroadcastOperator method), 207
- `_call()` (odl.operator.pspace_ops.ComponentProjection method), 210
- `_call()` (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 214
- `_call()` (odl.operator.pspace_ops.ProductSpaceOperator method), 218
- `_call()` (odl.operator.pspace_ops.ReductionOperator method), 223
- `_call()` (odl.space.fspace.FunctionSetVector method), 355
- `_call()` (odl.space.fspace.FunctionSpaceVector method), 365
- `_call()` (odl.space.ntuples.MatVecOperator method), 403
- `_call()` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper method), 425
- `_call()` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper method), 428
- `_call()` (odl.tomo.operators.ray_trafo.RayBackProjection method), 478
- `_call()` (odl.tomo.operators.ray_trafo.RayTransform method), 480
- `_call()` (odl.trafos.fourier.DiscreteFourierTransform method), 486
- `_call()` (odl.trafos.fourier.DiscreteFourierTransformInverse method), 491
- `_call()` (odl.trafos.fourier.FourierTransform method), 495
- `_call()` (odl.trafos.fourier.FourierTransformInverse method), 500
- `_call()` (odl.trafos.wavelet.WaveletTransform method), 512
- `_call()` (odl.trafos.wavelet.WaveletTransformInverse method), 515
- `_dist()` (odl.dscr.discretization.Discretization method), 85
- `_dist()` (odl.dscr.lp_dscr.DiscreteLp method), 127
- `_dist()` (odl.set.pspace.ProductSpace method), 237
- `_dist()` (odl.set.space.LinearSpace method), 259
- `_dist()` (odl.set.space.UniversalSpace method), 267
- `_dist()` (odl.space.base_ntuples.FnBase method), 294
- `_dist()` (odl.space.cu_ntuples.CudaFn method), 313
- `_dist()` (odl.space.fspace.FunctionSpace method), 357
- `_dist()` (odl.space.ntuples.Fn method), 370
- `_divide()` (odl.dscr.discretization.Discretization method), 85
- `_divide()` (odl.dscr.lp_dscr.DiscreteLp method), 127
- `_divide()` (odl.set.pspace.ProductSpace method), 237
- `_divide()` (odl.set.space.UniversalSpace method), 267
- `_divide()` (odl.space.base_ntuples.FnBase method), 294
- `_divide()` (odl.space.cu_ntuples.CudaFn method), 314
- `_divide()` (odl.space.fspace.FunctionSpace method), 357
- `_divide()` (odl.space.ntuples.Fn method), 370
- `_inner()` (odl.dscr.discretization.Discretization method), 85
- `_inner()` (odl.dscr.lp_dscr.DiscreteLp method), 127
- `_inner()` (odl.set.pspace.ProductSpace method), 237

- `_inner()` (odl.set.space.LinearSpace method), 259
 - `_inner()` (odl.set.space.UniversalSpace method), 267
 - `_inner()` (odl.space.base_ntuples.FnBase method), 295
 - `_inner()` (odl.space.cu_ntuples.CudaFn method), 314
 - `_inner()` (odl.space.fspace.FunctionSpace method), 357
 - `_inner()` (odl.space.ntuples.Fn method), 371
 - `_lincomb()` (odl.diagnostics.space.SpaceTest method), 45
 - `_lincomb()` (odl.dscr.discretization.Discretization method), 85
 - `_lincomb()` (odl.dscr.lp_dscr.DiscreteLp method), 127
 - `_lincomb()` (odl.set.pspace.ProductSpace method), 237
 - `_lincomb()` (odl.set.space.LinearSpace method), 259
 - `_lincomb()` (odl.set.space.UniversalSpace method), 267
 - `_lincomb()` (odl.space.base_ntuples.FnBase method), 295
 - `_lincomb()` (odl.space.cu_ntuples.CudaFn method), 314
 - `_lincomb()` (odl.space.fspace.FunctionSpace method), 358
 - `_lincomb()` (odl.space.ntuples.Fn method), 371
 - `_multiply()` (odl.dscr.discretization.Discretization method), 85
 - `_multiply()` (odl.dscr.lp_dscr.DiscreteLp method), 127
 - `_multiply()` (odl.set.pspace.ProductSpace method), 237
 - `_multiply()` (odl.set.space.LinearSpace method), 260
 - `_multiply()` (odl.set.space.UniversalSpace method), 268
 - `_multiply()` (odl.space.base_ntuples.FnBase method), 295
 - `_multiply()` (odl.space.cu_ntuples.CudaFn method), 315
 - `_multiply()` (odl.space.fspace.FunctionSpace method), 358
 - `_multiply()` (odl.space.ntuples.Fn method), 372
 - `_norm()` (odl.dscr.discretization.Discretization method), 85
 - `_norm()` (odl.dscr.lp_dscr.DiscreteLp method), 127
 - `_norm()` (odl.set.pspace.ProductSpace method), 237
 - `_norm()` (odl.set.space.LinearSpace method), 260
 - `_norm()` (odl.set.space.UniversalSpace method), 268
 - `_norm()` (odl.space.base_ntuples.FnBase method), 295
 - `_norm()` (odl.space.cu_ntuples.CudaFn method), 315
 - `_norm()` (odl.space.fspace.FunctionSpace method), 358
 - `_norm()` (odl.space.ntuples.Fn method), 372
- ## A
- `absolute()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 534
 - `absolute()` (odl.util.ufuncs.DiscreteLpUFuncs method), 547
 - `absolute()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 560
 - `absolute()` (odl.util.ufuncs.NtuplesUFuncs method), 572
 - `absolute()` (odl.util.ufuncs.ProductSpaceUFuncs method), 585
 - `add()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 534
 - `add()` (odl.util.ufuncs.DiscreteLpUFuncs method), 547
 - `add()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 560
 - `add()` (odl.util.ufuncs.NtuplesUFuncs method), 573
 - `add()` (odl.util.ufuncs.ProductSpaceUFuncs method), 585
 - `adjoint` (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 48
 - `adjoint` (odl.dscr.dscr_mappings.LinearInterpolation attribute), 52
 - `adjoint` (odl.dscr.dscr_mappings.NearestInterpolation attribute), 55
 - `adjoint` (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 59
 - `adjoint` (odl.dscr.dscr_mappings.PointCollocation attribute), 63
 - `adjoint` (odl.dscr.dscr_ops.Divergence attribute), 67
 - `adjoint` (odl.dscr.dscr_ops.Gradient attribute), 70
 - `adjoint` (odl.dscr.dscr_ops.Laplacian attribute), 73
 - `adjoint` (odl.dscr.dscr_ops.PartialDerivative attribute), 76
 - `adjoint` (odl.operator.default_ops.ConstantOperator attribute), 152
 - `adjoint` (odl.operator.default_ops.IdentityOperator attribute), 155
 - `adjoint` (odl.operator.default_ops.InnerProductOperator attribute), 158
 - `adjoint` (odl.operator.default_ops.LinCombOperator attribute), 161
 - `adjoint` (odl.operator.default_ops.MultiplyOperator attribute), 164
 - `adjoint` (odl.operator.default_ops.ResidualOperator attribute), 167
 - `adjoint` (odl.operator.default_ops.ScalingOperator attribute), 170
 - `adjoint` (odl.operator.default_ops.ZeroOperator attribute), 173
 - `adjoint` (odl.operator.operator.FunctionalLeftVectorMult attribute), 176
 - `adjoint` (odl.operator.operator.Operator attribute), 181
 - `adjoint` (odl.operator.operator.OperatorComp attribute), 184
 - `adjoint` (odl.operator.operator.OperatorLeftScalarMult attribute), 187
 - `adjoint` (odl.operator.operator.OperatorLeftVectorMult attribute), 189
 - `adjoint` (odl.operator.operator.OperatorPointwiseProduct attribute), 192
 - `adjoint` (odl.operator.operator.OperatorRightScalarMult attribute), 194
 - `adjoint` (odl.operator.operator.OperatorRightVectorMult attribute), 197
 - `adjoint` (odl.operator.operator.OperatorSum attribute), 200
 - `adjoint` (odl.operator.pspace_ops.BroadcastOperator attribute), 205
 - `adjoint` (odl.operator.pspace_ops.ComponentProjection attribute), 208

adjoint (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 212

adjoint (odl.operator.pspace_ops.ProductSpaceOperator attribute), 216

adjoint (odl.operator.pspace_ops.ReductionOperator attribute), 221

adjoint (odl.space.fspace.FunctionSetVector attribute), 353

adjoint (odl.space.fspace.FunctionSpaceVector attribute), 362

adjoint (odl.space.ntuples.MatVecOperator attribute), 401

adjoint (odl.tomo.backends.stir_bindings.BackProjectorByBinWrappers attribute), 423

adjoint (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrappers attribute), 426

adjoint (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476

adjoint (odl.tomo.operators.ray_trafo.RayTransform attribute), 479

adjoint (odl.trafos.fourier.DiscreteFourierTransform attribute), 484

adjoint (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489

adjoint (odl.trafos.fourier.FourierTransform attribute), 493

adjoint (odl.trafos.fourier.FourierTransformInverse attribute), 498

adjoint (odl.trafos.wavelet.WaveletTransform attribute), 510

adjoint (odl.trafos.wavelet.WaveletTransformInverse attribute), 513

adjoint() (odl.diagnostics.operator.OperatorTest method), 43

all_almost_equal() (in module odl.util.testutils), 531

all_almost_equal_array() (in module odl.util.testutils), 531

all_equal() (in module odl.util.testutils), 531

almost_equal() (in module odl.util.testutils), 531

AndPartial (class in odl.solvers.util.partial), 287

angles_from_matrix() (in module odl.tomo.util.utility), 481

append() (odl.discr.grid.RegularGrid method), 106

append() (odl.discr.grid.TensorGrid method), 115

append() (odl.set.domain.IntervalProd method), 227

apply_on_boundary() (in module odl.util.numerics), 523

approx_contains() (odl.discr.grid.RegularGrid method), 106

approx_contains() (odl.discr.grid.TensorGrid method), 115

approx_contains() (odl.set.domain.IntervalProd method), 228

approx_equals() (odl.discr.grid.RegularGrid method), 106

approx_equals() (odl.discr.grid.TensorGrid method), 116

approx_equals() (odl.discr.partition.RectPartition method), 147

approx_equals() (odl.set.domain.IntervalProd method), 228

arccos() (odl.util.ufuncs.CudaNtuplesUFuncs method), 535

arccos() (odl.util.ufuncs.DiscreteLpUFuncs method), 547

arccos() (odl.util.ufuncs.NtuplesBaseUFuncs method), 560

arccos() (odl.util.ufuncs.NtuplesUFuncs method), 573

arccos() (odl.util.ufuncs.ProductSpaceUFuncs method), 585

arccosh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 547

arccosh() (odl.util.ufuncs.DiscreteLpUFuncs method), 560

arccosh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 573

arccosh() (odl.util.ufuncs.NtuplesUFuncs method), 585

arccosh() (odl.util.ufuncs.ProductSpaceUFuncs method), 585

arcsin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 535

arcsin() (odl.util.ufuncs.DiscreteLpUFuncs method), 548

arcsin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 560

arcsin() (odl.util.ufuncs.NtuplesUFuncs method), 573

arcsin() (odl.util.ufuncs.ProductSpaceUFuncs method), 586

arcsinh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 535

arcsinh() (odl.util.ufuncs.DiscreteLpUFuncs method), 548

arcsinh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 561

arcsinh() (odl.util.ufuncs.NtuplesUFuncs method), 573

arcsinh() (odl.util.ufuncs.ProductSpaceUFuncs method), 586

arctan() (odl.util.ufuncs.CudaNtuplesUFuncs method), 535

arctan() (odl.util.ufuncs.DiscreteLpUFuncs method), 548

arctan() (odl.util.ufuncs.NtuplesBaseUFuncs method), 561

arctan() (odl.util.ufuncs.NtuplesUFuncs method), 573

arctan() (odl.util.ufuncs.ProductSpaceUFuncs method), 586

arctan2() (odl.util.ufuncs.CudaNtuplesUFuncs method), 535

arctan2() (odl.util.ufuncs.DiscreteLpUFuncs method), 548

arctan2() (odl.util.ufuncs.NtuplesBaseUFuncs method), 561

arctan2() (odl.util.ufuncs.NtuplesUFuncs method), 573

- arctan2() (odl.util.ufuncs.ProductSpaceUFuncs method), 586
- arctanh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 536
- arctanh() (odl.util.ufuncs.DiscreteLpUFuncs method), 548
- arctanh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 561
- arctanh() (odl.util.ufuncs.NtuplesUFuncs method), 574
- arctanh() (odl.util.ufuncs.ProductSpaceUFuncs method), 586
- area (odl.set.domain.IntervalProd attribute), 225
- array-like, 34
- array1d_repr() (in module odl.util.utility), 597
- array1d_str() (in module odl.util.utility), 598
- array_to_pywt_coeff() (in module odl.trafos.wavelet), 516
- arraynd_repr() (in module odl.util.utility), 598
- arraynd_str() (in module odl.util.utility), 598
- asarray() (odl.discr.discretization.DiscretizationVector method), 92
- asarray() (odl.discr.discretization.RawDiscretizationVector method), 100
- asarray() (odl.discr.lp_discr.DiscreteLpVector method), 135
- asarray() (odl.space.base_ntuples.FnBaseVector method), 300
- asarray() (odl.space.base_ntuples.NtuplesBaseVector method), 309
- asarray() (odl.space.cu_ntuples.CudaFnVector method), 334
- asarray() (odl.space.cu_ntuples.CudaNtuplesVector method), 346
- asarray() (odl.space.ntuples.FnVector method), 396
- asarray() (odl.space.ntuples.NtuplesVector method), 412
- assign() (odl.discr.discretization.DiscretizationVector method), 92
- assign() (odl.discr.lp_discr.DiscreteLpVector method), 135
- assign() (odl.set.pspace.ProductSpaceVector method), 245
- assign() (odl.set.space.LinearSpaceVector method), 265
- assign() (odl.space.base_ntuples.FnBaseVector method), 301
- assign() (odl.space.cu_ntuples.CudaFnVector method), 335
- assign() (odl.space.cu_ntuples.CudaNtuplesVector method), 347
- assign() (odl.space.fspace.FunctionSetVector method), 355
- assign() (odl.space.fspace.FunctionSpaceVector method), 365
- assign() (odl.space.ntuples.FnVector method), 396
- astra_algorithm() (in module odl.tomo.backends.astra_setup), 420
- astra_conebeam_2d_geom_to_vec() (in module odl.tomo.backends.astra_setup), 420
- astra_conebeam_3d_geom_to_vec() (in module odl.tomo.backends.astra_setup), 421
- astra_cpu_back_projector() (in module odl.tomo.backends.astra_cpu), 417
- astra_cpu_forward_projector() (in module odl.tomo.backends.astra_cpu), 418
- astra_cuda_back_projector() (in module odl.tomo.backends.astra_cuda), 418
- astra_cuda_forward_projector() (in module odl.tomo.backends.astra_cuda), 419
- astra_data() (in module odl.tomo.backends.astra_setup), 421
- astra_parallel_3d_geom_to_vec() (in module odl.tomo.backends.astra_setup), 421
- astra_projection_geometry() (in module odl.tomo.backends.astra_setup), 422
- astra_projector() (in module odl.tomo.backends.astra_setup), 422
- astra_volume_geometry() (in module odl.tomo.backends.astra_setup), 422
- astype() (odl.discr.discretization.Discretization method), 85
- astype() (odl.discr.lp_discr.DiscreteLp method), 128
- astype() (odl.space.base_ntuples.FnBase method), 295
- astype() (odl.space.cu_ntuples.CudaFn method), 316
- astype() (odl.space.fspace.FunctionSpace method), 358
- astype() (odl.space.ntuples.Fn method), 373
- axes (odl.tomo.geometry.detector.Flat2dDetector attribute), 445
- axes (odl.trafos.fourier.DiscreteFourierTransform attribute), 484
- axes (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
- axes (odl.trafos.fourier.FourierTransform attribute), 493
- axes (odl.trafos.fourier.FourierTransformInverse attribute), 498
- axis (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 430
- axis (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435
- axis (odl.tomo.geometry.detector.Flat1dDetector attribute), 443
- axis (odl.tomo.geometry.geometry.AxisOrientedGeometry attribute), 454
- axis (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 465
- axis_rotation() (in module odl.tomo.util.utility), 481
- axis_rotation_matrix() (in module odl.tomo.util.utility), 482

AxisOrientedGeometry (class
odl.tomo.geometry.geometry), 453

B

BackProjectorByBinWrapper (class
odl.tomo.backends.stir_bindings), 423

BacktrackingLineSearch (class
odl.solvers.scalar.steplen), 283

BarzilaiBorweinStep (class in odl.solvers.scalar.steplen),
284

begin (odl.discr.partition.RectPartition attribute), 144

begin (odl.set.domain.IntervalProd attribute), 225

bfgs_method() (in module odl.solvers.findroot.newton),
276

bitwise_and() (odl.util.ufuncs.CudaNtuplesUFuncs
method), 536

bitwise_and() (odl.util.ufuncs.DiscreteLpUFuncs
method), 548

bitwise_and() (odl.util.ufuncs.NtuplesBaseUFuncs
method), 561

bitwise_and() (odl.util.ufuncs.NtuplesUFuncs method),
574

bitwise_and() (odl.util.ufuncs.ProductSpaceUFuncs
method), 586

bitwise_or() (odl.util.ufuncs.CudaNtuplesUFuncs
method), 536

bitwise_or() (odl.util.ufuncs.DiscreteLpUFuncs method),
549

bitwise_or() (odl.util.ufuncs.NtuplesBaseUFuncs
method), 561

bitwise_or() (odl.util.ufuncs.NtuplesUFuncs method),
574

bitwise_or() (odl.util.ufuncs.ProductSpaceUFuncs
method), 587

bitwise_xor() (odl.util.ufuncs.CudaNtuplesUFuncs
method), 536

bitwise_xor() (odl.util.ufuncs.DiscreteLpUFuncs
method), 549

bitwise_xor() (odl.util.ufuncs.NtuplesBaseUFuncs
method), 561

bitwise_xor() (odl.util.ufuncs.NtuplesUFuncs method),
574

bitwise_xor() (odl.util.ufuncs.ProductSpaceUFuncs
method), 587

boundary_cell_fractions (odl.discr.partition.RectPartition
attribute), 144

BroadcastOperator (class in odl.operator.pspace_ops),
204

broydens_first_method() (in module
odl.solvers.findroot.newton), 277

broydens_second_method() (in module
odl.solvers.findroot.newton), 277

C

CartesianProduct (class in odl.set.sets), 247

ceil() (odl.util.ufuncs.CudaNtuplesUFuncs method), 536

ceil() (odl.util.ufuncs.DiscreteLpUFuncs method), 549

ceil() (odl.util.ufuncs.NtuplesBaseUFuncs method), 562

ceil() (odl.util.ufuncs.NtuplesUFuncs method), 574

ceil() (odl.util.ufuncs.ProductSpaceUFuncs method), 587

cell_boundary_vecs (odl.discr.partition.RectPartition at-
tribute), 145

cell_sides (odl.discr.lp_discr.DiscreteLp attribute), 124

cell_sides (odl.discr.lp_discr.DiscreteLpVector attribute),
132

cell_sides (odl.discr.partition.RectPartition attribute), 145

cell_sizes_vecs (odl.discr.partition.RectPartition at-
tribute), 145

cell_volume (odl.discr.lp_discr.DiscreteLp attribute), 124

cell_volume (odl.discr.lp_discr.DiscreteLpVector at-
tribute), 132

cell_volume (odl.discr.partition.RectPartition attribute),
146

center (odl.discr.grid.RegularGrid attribute), 102

chambolle_pock_solver() (in module
odl.solvers.advanced.chambolle_pock), 271

circ_rad (odl.tomo.geometry.detector.CircleSectionDetector
attribute), 440

CircleSectionDetector (class in
odl.tomo.geometry.detector), 439

CircularConeFlatGeometry (class in
odl.tomo.geometry.conebeam), 429

clear_fftw_plan() (odl.trafos.fourier.DiscreteFourierTransform
method), 487

clear_fftw_plan() (odl.trafos.fourier.DiscreteFourierTransformInverse
method), 491

clear_fftw_plan() (odl.trafos.fourier.FourierTransform
method), 495

clear_fftw_plan() (odl.trafos.fourier.FourierTransformInverse
method), 501

clear_temporaries() (odl.trafos.fourier.FourierTransform
method), 496

clear_temporaries() (odl.trafos.fourier.FourierTransformInverse
method), 501

Cn() (in module odl.space.ntuples), 414

coeff_size_list() (in module odl.trafos.wavelet), 517

collapse() (odl.set.domain.IntervalProd method), 229

combine_proximals() (in module
odl.solvers.advanced.proximal_operators),
274

ComplexNumbers (class in odl.set.sets), 249

ComponentProjection (class in odl.operator.pspace_ops),
208

ComponentProjectionAdjoint (class in
odl.operator.pspace_ops), 211

conj() (odl.discr.lp_discr.DiscreteLpVector method), 136

- conj() (odl.space.fspace.FunctionSpaceVector method), 365
- conj() (odl.space.ntuples.FnVector method), 396
- conj() (odl.util.ufuncs.CudaNtuplesUFuncs method), 536
- conj() (odl.util.ufuncs.DiscreteLpUFuncs method), 549
- conj() (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
- conj() (odl.util.ufuncs.NtuplesUFuncs method), 574
- conj() (odl.util.ufuncs.ProductSpaceUFuncs method), 587
- conj_exponent() (in module odl.util.utility), 599
- conjugate_gradient() (in module odl.solvers.iterative.iterative), 278
- conjugate_gradient_normal() (in module odl.solvers.iterative.iterative), 279
- const (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 321
- const (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 328
- const (odl.space.ntuples.FnConstWeighting attribute), 379
- const (odl.space.ntuples.FnNoWeighting attribute), 388
- ConstantLineSearch (class in odl.solvers.scalar.steplen), 285
- ConstantOperator (class in odl.operator.default_ops), 152
- contains() (odl.diagnostics.space.SpaceTest method), 45
- contains_all() (odl.discr.discretization.Discretization method), 86
- contains_all() (odl.discr.discretization.RawDiscretization method), 96
- contains_all() (odl.discr.grid.RegularGrid method), 107
- contains_all() (odl.discr.grid.TensorGrid method), 116
- contains_all() (odl.discr.lp_discr.DiscreteLp method), 128
- contains_all() (odl.set.domain.IntervalProd method), 229
- contains_all() (odl.set.pspace.ProductSpace method), 238
- contains_all() (odl.set.sets.CartesianProduct method), 248
- contains_all() (odl.set.sets.ComplexNumbers method), 249
- contains_all() (odl.set.sets.EmptySet method), 250
- contains_all() (odl.set.sets.Field method), 251
- contains_all() (odl.set.sets.Integers method), 252
- contains_all() (odl.set.sets.RealNumbers method), 253
- contains_all() (odl.set.sets.Set method), 255
- contains_all() (odl.set.sets.Strings method), 256
- contains_all() (odl.set.sets.UniversalSet method), 257
- contains_all() (odl.set.space.LinearSpace method), 260
- contains_all() (odl.set.space.UniversalSpace method), 268
- contains_all() (odl.space.base_ntuples.FnBase method), 295
- contains_all() (odl.space.base_ntuples.NtuplesBase method), 306
- contains_all() (odl.space.cu_ntuples.CudaFn method), 316
- contains_all() (odl.space.cu_ntuples.CudaNtuples method), 340
- contains_all() (odl.space.fspace.FunctionSet method), 352
- contains_all() (odl.space.fspace.FunctionSpace method), 358
- contains_all() (odl.space.ntuples.Fn method), 373
- contains_all() (odl.space.ntuples.Ntuples method), 406
- convex_hull() (odl.discr.grid.RegularGrid method), 107
- convex_hull() (odl.discr.grid.TensorGrid method), 116
- coord_vectors (odl.discr.grid.RegularGrid attribute), 103
- coord_vectors (odl.discr.grid.TensorGrid attribute), 112
- copy() (odl.discr.discretization.DiscretizationVector method), 92
- contains_all() (odl.space.cu_ntuples.CudaFn method), 316
- contains_all() (odl.space.cu_ntuples.CudaNtuples method), 340
- contains_all() (odl.space.fspace.FunctionSet method), 352
- contains_all() (odl.space.fspace.FunctionSpace method), 358
- contains_all() (odl.space.ntuples.Fn method), 373
- contains_all() (odl.space.ntuples.Ntuples method), 406
- contains_set() (odl.discr.discretization.Discretization method), 86
- contains_set() (odl.discr.discretization.RawDiscretization method), 96
- contains_set() (odl.discr.grid.RegularGrid method), 107
- contains_set() (odl.discr.grid.TensorGrid method), 116
- contains_set() (odl.discr.lp_discr.DiscreteLp method), 128
- contains_set() (odl.set.domain.IntervalProd method), 230
- contains_set() (odl.set.pspace.ProductSpace method), 238
- contains_set() (odl.set.sets.CartesianProduct method), 248
- contains_set() (odl.set.sets.ComplexNumbers method), 249
- contains_set() (odl.set.sets.EmptySet method), 250
- contains_set() (odl.set.sets.Field method), 252
- contains_set() (odl.set.sets.Integers method), 252
- contains_set() (odl.set.sets.RealNumbers method), 254
- contains_set() (odl.set.sets.Set method), 255
- contains_set() (odl.set.sets.Strings method), 256
- contains_set() (odl.set.sets.UniversalSet method), 257
- contains_set() (odl.set.space.LinearSpace method), 260
- contains_set() (odl.set.space.UniversalSpace method), 268
- contains_set() (odl.space.base_ntuples.FnBase method), 295
- contains_set() (odl.space.base_ntuples.NtuplesBase method), 306
- contains_set() (odl.space.cu_ntuples.CudaFn method), 316
- contains_set() (odl.space.cu_ntuples.CudaNtuples method), 340
- contains_set() (odl.space.fspace.FunctionSet method), 352
- contains_set() (odl.space.fspace.FunctionSpace method), 358
- contains_set() (odl.space.ntuples.Fn method), 373
- contains_set() (odl.space.ntuples.Ntuples method), 406
- convex_hull() (odl.discr.grid.RegularGrid method), 107
- convex_hull() (odl.discr.grid.TensorGrid method), 116
- coord_vectors (odl.discr.grid.RegularGrid attribute), 103
- coord_vectors (odl.discr.grid.TensorGrid attribute), 112
- copy() (odl.discr.discretization.DiscretizationVector method), 92

`copy()` (odl.dscr.discretization.RawDiscretizationVector method), 100
`copy()` (odl.dscr.lp_dscr.DiscreteLpVector method), 136
`copy()` (odl.set.pspace.ProductSpaceVector method), 245
`copy()` (odl.set.space.LinearSpaceVector method), 265
`copy()` (odl.space.base_ntuples.FnBaseVector method), 301
`copy()` (odl.space.base_ntuples.NtuplesBaseVector method), 309
`copy()` (odl.space.cu_ntuples.CudaFnVector method), 335
`copy()` (odl.space.cu_ntuples.CudaNtuplesVector method), 347
`copy()` (odl.space.fspace.FunctionSetVector method), 355
`copy()` (odl.space.fspace.FunctionSpaceVector method), 365
`copy()` (odl.space.ntuples.FnVector method), 397
`copy()` (odl.space.ntuples.NtuplesVector method), 413
`copysign()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 536
`copysign()` (odl.util.ufuncs.DiscreteLpUFuncs method), 549
`copysign()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
`copysign()` (odl.util.ufuncs.NtuplesUFuncs method), 574
`copysign()` (odl.util.ufuncs.ProductSpaceUFuncs method), 587
`corner_grid()` (odl.dscr.grid.RegularGrid method), 107
`corner_grid()` (odl.dscr.grid.TensorGrid method), 117
`corners()` (odl.dscr.grid.RegularGrid method), 108
`corners()` (odl.dscr.grid.TensorGrid method), 117
`corners()` (odl.set.domain.IntervalProd method), 230
`cos()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
`cos()` (odl.util.ufuncs.DiscreteLpUFuncs method), 549
`cos()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
`cos()` (odl.util.ufuncs.NtuplesUFuncs method), 575
`cos()` (odl.util.ufuncs.ProductSpaceUFuncs method), 587
`cosh()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
`cosh()` (odl.util.ufuncs.DiscreteLpUFuncs method), 549
`cosh()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
`cosh()` (odl.util.ufuncs.NtuplesUFuncs method), 575
`cosh()` (odl.util.ufuncs.ProductSpaceUFuncs method), 587
`create_temporaries()` (odl.trafos.fourier.FourierTransform method), 496
`create_temporaries()` (odl.trafos.fourier.FourierTransformInverse method), 501
`cu_weighted_dist()` (in module odl.space.cu_ntuples), 349
`cu_weighted_inner()` (in module odl.space.cu_ntuples), 350
`cu_weighted_norm()` (in module odl.space.cu_ntuples), 350
`Cuboid()` (in module odl.set.domain), 234
`cuboid()` (in module odl.util.phantom), 525

`CudaFn` (class in odl.space.cu_ntuples), 310
`CudaFnConstWeighting` (class in odl.space.cu_ntuples), 321
`CudaFnCustomDist` (class in odl.space.cu_ntuples), 323
`CudaFnCustomInnerProduct` (class in odl.space.cu_ntuples), 324
`CudaFnCustomNorm` (class in odl.space.cu_ntuples), 326
`CudaFnNoWeighting` (class in odl.space.cu_ntuples), 328
`CudaFnVector` (class in odl.space.cu_ntuples), 330
`CudaFnVectorWeighting` (class in odl.space.cu_ntuples), 336
`CudaNtuples` (class in odl.space.cu_ntuples), 339
`CudaNtuplesUFuncs` (class in odl.util.ufuncs), 533
`CudaNtuplesVector` (class in odl.space.cu_ntuples), 342
`CudaRn()` (in module odl.space.cu_ntuples), 349

D

`data` (odl.space.cu_ntuples.CudaFnVector attribute), 330
`data` (odl.space.cu_ntuples.CudaNtuplesVector attribute), 342
`data` (odl.space.ntuples.FnVector attribute), 391
`data` (odl.space.ntuples.NtuplesVector attribute), 408
`data_ptr` (odl.space.cu_ntuples.CudaFnVector attribute), 331
`data_ptr` (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
`data_ptr` (odl.space.ntuples.FnVector attribute), 391
`data_ptr` (odl.space.ntuples.NtuplesVector attribute), 408
`default_dtype()` (odl.space.cu_ntuples.CudaFn static method), 316
`default_dtype()` (odl.space.ntuples.Fn static method), 373
`deg2rad()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
`deg2rad()` (odl.util.ufuncs.DiscreteLpUFuncs method), 550
`deg2rad()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
`deg2rad()` (odl.util.ufuncs.NtuplesUFuncs method), 575
`deg2rad()` (odl.util.ufuncs.ProductSpaceUFuncs method), 588
`derenzo_sources()` (in module odl.util.phantom), 526
`derivative()` (odl.diagnostics.operator.OperatorTest method), 43
`derivative()` (odl.dscr.dscr_mappings.FunctionSetMapping method), 51
`derivative()` (odl.dscr.dscr_mappings.LinearInterpolation method), 54
`derivative()` (odl.dscr.dscr_mappings.NearestInterpolation method), 58
`derivative()` (odl.dscr.dscr_mappings.PerAxisInterpolation method), 62
`derivative()` (odl.dscr.dscr_mappings.PointCollocation method), 66
`derivative()` (odl.dscr.dscr_ops.Divergence method), 69

- `derivative()` (odl.discr.discr_ops.Gradient method), 72
`derivative()` (odl.discr.discr_ops.Laplacian method), 75
`derivative()` (odl.discr.discr_ops.PartialDerivative method), 79
`derivative()` (odl.operator.default_ops.ConstantOperator method), 155
`derivative()` (odl.operator.default_ops.IdentityOperator method), 157
`derivative()` (odl.operator.default_ops.InnerProductOperator method), 161
`derivative()` (odl.operator.default_ops.LinCombOperator method), 163
`derivative()` (odl.operator.default_ops.MultiplyOperator method), 167
`derivative()` (odl.operator.default_ops.ResidualOperator method), 169
`derivative()` (odl.operator.default_ops.ScalingOperator method), 173
`derivative()` (odl.operator.default_ops.ZeroOperator method), 175
`derivative()` (odl.operator.operator.FunctionalLeftVectorMult method), 178
`derivative()` (odl.operator.operator.Operator method), 183
`derivative()` (odl.operator.operator.OperatorComp method), 186
`derivative()` (odl.operator.operator.OperatorLeftScalarMult method), 189
`derivative()` (odl.operator.operator.OperatorLeftVectorMult method), 191
`derivative()` (odl.operator.operator.OperatorPointwiseProduct method), 194
`derivative()` (odl.operator.operator.OperatorRightScalarMult method), 196
`derivative()` (odl.operator.operator.OperatorRightVectorMult method), 199
`derivative()` (odl.operator.operator.OperatorSum method), 202
`derivative()` (odl.operator.pspace_ops.BroadcastOperator method), 207
`derivative()` (odl.operator.pspace_ops.ComponentProjection method), 211
`derivative()` (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 214
`derivative()` (odl.operator.pspace_ops.ProductSpaceOperator method), 219
`derivative()` (odl.operator.pspace_ops.ReductionOperator method), 223
`derivative()` (odl.space.fspace.FunctionSetVector method), 355
`derivative()` (odl.space.fspace.FunctionSpaceVector method), 365
`derivative()` (odl.space.ntuples.MatVecOperator method), 403
`derivative()` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper method), 425
`derivative()` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper method), 428
`derivative()` (odl.tomo.operators.ray_trafo.RayBackProjection method), 478
`derivative()` (odl.tomo.operators.ray_trafo.RayTransform method), 481
`derivative()` (odl.trafos.fourier.DiscreteFourierTransform method), 487
`derivative()` (odl.trafos.fourier.DiscreteFourierTransformInverse method), 491
`derivative()` (odl.trafos.fourier.FourierTransform method), 496
`derivative()` (odl.trafos.fourier.FourierTransformInverse method), 501
`derivative()` (odl.trafos.wavelet.WaveletTransform method), 512
`derivative()` (odl.trafos.wavelet.WaveletTransformInverse method), 515
`det_grid` (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 430
`det_grid` (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435
`det_grid` (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
`det_grid` (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
`det_grid` (odl.tomo.geometry.geometry.Geometry attribute), 458
`det_grid` (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462
`det_grid` (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 465
`det_grid` (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469
`det_grid` (odl.tomo.geometry.parallel.ParallelGeometry attribute), 472
`det_params` (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 430
`det_params` (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435
`det_params` (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
`det_params` (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
`det_params` (odl.tomo.geometry.geometry.Geometry attribute), 458
`det_params` (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462
`det_params` (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 465
`det_params` (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469

det_params (odl.tomo.geometry.parallel.ParallelGeometry attribute), 472	det_refpoint() (odl.tomo.geometry.parallel.Parallel2dGeometry method), 463
det_partition (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 430	dft_compute() (odl.tomo.geometry.parallel.Parallel3dAxisGeometry method), 467
det_partition (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435	dft_compute() (odl.tomo.geometry.parallel.Parallel3dGeometry method), 471
det_partition (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450	det_refpoint() (odl.tomo.geometry.parallel.ParallelGeometry method), 474
det_partition (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455	GeometrySrc() (odl.tomo.geometry.conebeam.CircularConeFlatGeometry method), 433
det_partition (odl.tomo.geometry.geometry.Geometry attribute), 458	det_to_src() (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 437
det_partition (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462	det_to_src() (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 452
det_partition (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466	det_to_src() (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 457
det_partition (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469	det_to_src() (odl.tomo.geometry.geometry.Geometry method), 460
det_partition (odl.tomo.geometry.parallel.ParallelGeometry attribute), 472	det_to_src() (odl.tomo.geometry.parallel.Parallel2dGeometry method), 464
det_point_position() (odl.tomo.geometry.conebeam.CircularConeFlatGeometry method), 432	detector (odl.tomo.geometry.parallel.Parallel3dAxisGeometry method), 467
det_point_position() (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 437	detector (odl.tomo.geometry.parallel.Parallel3dGeometry method), 471
det_point_position() (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 451	detector_src() (odl.tomo.geometry.parallel.ParallelGeometry method), 474
det_point_position() (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 456	Detector class in odl.tomo.geometry.detector, 441
det_point_position() (odl.tomo.geometry.geometry.Geometry method), 460	detector (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
det_point_position() (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 435	detector (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435
det_point_position() (odl.tomo.geometry.parallel.Parallel2dGeometry method), 463	detector (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
det_point_position() (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 450	detector (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
det_point_position() (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 455	detector (odl.tomo.geometry.geometry.Geometry attribute), 459
det_point_position() (odl.tomo.geometry.parallel.Parallel3dGeometry method), 470	detector (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 435
det_point_position() (odl.tomo.geometry.parallel.ParallelGeometry attribute), 459	detector (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 435
det_point_position() (odl.tomo.geometry.parallel.ParallelGeometry method), 474	detector (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469
det_radius (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 430	detector (odl.tomo.geometry.parallel.ParallelGeometry attribute), 451
det_radius (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435	dft_postprocess_data() (in module odl.trafos.fourier), 503
det_radius (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450	dft_postprocess_data() (in module odl.trafos.fourier), 504
det_refpoint() (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 472	diagonal_operator() (in module odl.operator.pspace_ops), 224
det_refpoint() (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 437	discr_sequence_space() (in module odl.discr.lp_discr), 139
det_refpoint() (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 451	DiscreteFourierTransform (class in odl.trafos.fourier), 484
det_refpoint() (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 456	
det_refpoint() (odl.tomo.geometry.geometry.Geometry method), 460	

DiscreteFourierTransformInverse (class in odl.trafos.fourier), 488
 DiscreteLp (class in odl.dscr.lp_dscr), 123
 DiscreteLpUFuncs (class in odl.util.ufuncs), 545
 DiscreteLpVector (class in odl.dscr.lp_dscr), 131
 discretization, 34
 Discretization (class in odl.dscr.discretization), 82
 DiscretizationVector (class in odl.dscr.discretization), 89
 dist (odl.space.ntuples.FnCustomDist attribute), 381
 dist() (odl.diagnostics.space.SpaceTest method), 45
 dist() (odl.dscr.discretization.Discretization method), 86
 dist() (odl.dscr.discretization.DiscretizationVector method), 92
 dist() (odl.dscr.lp_dscr.DiscreteLp method), 128
 dist() (odl.dscr.lp_dscr.DiscreteLpVector method), 136
 dist() (odl.set.domain.IntervalProd method), 231
 dist() (odl.set.pspace.ProductSpace method), 238
 dist() (odl.set.pspace.ProductSpaceVector method), 245
 dist() (odl.set.space.LinearSpace method), 260
 dist() (odl.set.space.LinearSpaceVector method), 265
 dist() (odl.set.space.UniversalSpace method), 268
 dist() (odl.space.base_ntuples.FnBase method), 295
 dist() (odl.space.base_ntuples.FnBaseVector method), 301
 dist() (odl.space.base_ntuples.FnWeightingBase method), 303
 dist() (odl.space.cu_ntuples.CudaFn method), 316
 dist() (odl.space.cu_ntuples.CudaFnConstWeighting method), 322
 dist() (odl.space.cu_ntuples.CudaFnCustomDist method), 323
 dist() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 325
 dist() (odl.space.cu_ntuples.CudaFnCustomNorm method), 327
 dist() (odl.space.cu_ntuples.CudaFnNoWeighting method), 329
 dist() (odl.space.cu_ntuples.CudaFnVector method), 335
 dist() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 337
 dist() (odl.space.cu_ntuples.CudaNtuplesVector method), 347
 dist() (odl.space.fspace.FunctionSpace method), 359
 dist() (odl.space.fspace.FunctionSpaceVector method), 365
 dist() (odl.space.ntuples.Fn method), 373
 dist() (odl.space.ntuples.FnConstWeighting method), 379
 dist() (odl.space.ntuples.FnCustomInnerProduct method), 383
 dist() (odl.space.ntuples.FnCustomNorm method), 384
 dist() (odl.space.ntuples.FnMatrixWeighting method), 387
 dist() (odl.space.ntuples.FnNoWeighting method), 389
 dist() (odl.space.ntuples.FnVector method), 397
 dist() (odl.space.ntuples.FnVectorWeighting method), 400
 Divergence (class in odl.dscr.dscr_ops), 67
 DivergentBeamGeometry (class in odl.tomo.geometry.geometry), 454
 divide() (odl.dscr.discretization.Discretization method), 86
 divide() (odl.dscr.discretization.DiscretizationVector method), 92
 divide() (odl.dscr.lp_dscr.DiscreteLp method), 128
 divide() (odl.dscr.lp_dscr.DiscreteLpVector method), 136
 divide() (odl.set.pspace.ProductSpace method), 238
 divide() (odl.set.pspace.ProductSpaceVector method), 245
 divide() (odl.set.space.LinearSpace method), 260
 divide() (odl.set.space.LinearSpaceVector method), 265
 divide() (odl.set.space.UniversalSpace method), 268
 divide() (odl.space.base_ntuples.FnBase method), 296
 divide() (odl.space.base_ntuples.FnBaseVector method), 301
 divide() (odl.space.cu_ntuples.CudaFn method), 317
 divide() (odl.space.cu_ntuples.CudaFnVector method), 335
 divide() (odl.space.cu_ntuples.CudaNtuplesVector method), 347
 divide() (odl.space.fspace.FunctionSpace method), 359
 divide() (odl.space.fspace.FunctionSpaceVector method), 365
 divide() (odl.space.ntuples.Fn method), 373
 divide() (odl.space.ntuples.FnVector method), 397
 divide() (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
 divide() (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 divide() (odl.util.ufuncs.NtuplesBaseUFuncs method), 562
 divide() (odl.util.ufuncs.NtuplesUFuncs method), 575
 divide() (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 domain, 34
 domain (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 48
 domain (odl.dscr.dscr_mappings.LinearInterpolation attribute), 52
 domain (odl.dscr.dscr_mappings.NearestInterpolation attribute), 55
 domain (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 59
 domain (odl.dscr.dscr_mappings.PointCollocation attribute), 63
 domain (odl.dscr.dscr_ops.Divergence attribute), 67
 domain (odl.dscr.dscr_ops.Gradient attribute), 70
 domain (odl.dscr.dscr_ops.Laplacian attribute), 73

- domain (odl.dscr.dscr_ops.PartialDerivative attribute), 76
- domain (odl.dscr.discretization.Discretization attribute), 82
- domain (odl.dscr.discretization.RawDiscretization attribute), 95
- domain (odl.dscr.lp_dscr.DiscreteLp attribute), 124
- domain (odl.operator.default_ops.ConstantOperator attribute), 153
- domain (odl.operator.default_ops.IdentityOperator attribute), 155
- domain (odl.operator.default_ops.InnerProductOperator attribute), 159
- domain (odl.operator.default_ops.LinCombOperator attribute), 161
- domain (odl.operator.default_ops.MultiplyOperator attribute), 164
- domain (odl.operator.default_ops.ResidualOperator attribute), 167
- domain (odl.operator.default_ops.ScalingOperator attribute), 170
- domain (odl.operator.default_ops.ZeroOperator attribute), 173
- domain (odl.operator.operator.FunctionalLeftVectorMult attribute), 177
- domain (odl.operator.operator.Operator attribute), 181
- domain (odl.operator.operator.OperatorComp attribute), 184
- domain (odl.operator.operator.OperatorLeftScalarMult attribute), 187
- domain (odl.operator.operator.OperatorLeftVectorMult attribute), 190
- domain (odl.operator.operator.OperatorPointwiseProduct attribute), 192
- domain (odl.operator.operator.OperatorRightScalarMult attribute), 195
- domain (odl.operator.operator.OperatorRightVectorMult attribute), 197
- domain (odl.operator.operator.OperatorSum attribute), 200
- domain (odl.operator.pspace_ops.BroadcastOperator attribute), 205
- domain (odl.operator.pspace_ops.ComponentProjection attribute), 209
- domain (odl.operator.pspace_ops.ComponentProjectionAdj attribute), 212
- domain (odl.operator.pspace_ops.ProductSpaceOperator attribute), 217
- domain (odl.operator.pspace_ops.ReductionOperator attribute), 221
- domain (odl.space.fspace.FunctionSet attribute), 351
- domain (odl.space.fspace.FunctionSetVector attribute), 353
- domain (odl.space.fspace.FunctionSpace attribute), 356
- domain (odl.space.fspace.FunctionSpaceVector attribute), 363
- domain (odl.space.ntuples.MatVecOperator attribute), 402
- domain (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 424
- domain (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 426
- domain (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
- domain (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
- domain (odl.trafos.fourier.DiscreteFourierTransform attribute), 484
- domain (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
- domain (odl.trafos.fourier.FourierTransform attribute), 493
- domain (odl.trafos.fourier.FourierTransformInverse attribute), 498
- domain (odl.trafos.wavelet.WaveletTransform attribute), 510
- domain (odl.trafos.wavelet.WaveletTransformInverse attribute), 513
- dspace (odl.dscr.discretization.Discretization attribute), 82
- dspace (odl.dscr.discretization.RawDiscretization attribute), 95
- dspace (odl.dscr.lp_dscr.DiscreteLp attribute), 124
- dspace_type (odl.dscr.discretization.Discretization attribute), 82
- dspace_type (odl.dscr.discretization.RawDiscretization attribute), 95
- dspace_type (odl.dscr.lp_dscr.DiscreteLp attribute), 124
- dspace_type() (in module odl.dscr.discretization), 101
- dtype, 34
- dtype (odl.dscr.discretization.Discretization attribute), 83
- dtype (odl.dscr.discretization.DiscretizationVector attribute), 89
- dtype (odl.dscr.discretization.RawDiscretization attribute), 95
- dtype (odl.dscr.discretization.RawDiscretizationVector attribute), 98
- dtype (odl.dscr.lp_dscr.DiscreteLp attribute), 124
- dtype (odl.dscr.lp_dscr.DiscreteLpVector attribute), 132
- dtype (odl.space.base_ntuples.FnBase attribute), 292
- dtype (odl.space.base_ntuples.FnBaseVector attribute), 299
- dtype (odl.space.base_ntuples.NtuplesBase attribute), 305
- dtype (odl.space.base_ntuples.NtuplesBaseVector attribute), 307
- dtype (odl.space.cu_ntuples.CudaFn attribute), 311
- dtype (odl.space.cu_ntuples.CudaFnVector attribute), 331

dtype (odl.space.cu_ntuples.CudaNtuples attribute), 339
 dtype (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
 dtype (odl.space.ntuples.Fn attribute), 367
 dtype (odl.space.ntuples.FnVector attribute), 392
 dtype (odl.space.ntuples.Ntuples attribute), 404
 dtype (odl.space.ntuples.NtuplesVector attribute), 408
 dtype_repr() (in module odl.util.utility), 599

E

element, 34
 element() (odl.diagnostics.space.SpaceTest method), 45
 element() (odl.discr.discretization.Discretization method), 86
 element() (odl.discr.discretization.RawDiscretization method), 97
 element() (odl.discr.grid.RegularGrid method), 108
 element() (odl.discr.grid.TensorGrid method), 117
 element() (odl.discr.lp_discr.DiscreteLp method), 129
 element() (odl.set.domain.IntervalProd method), 231
 element() (odl.set.pspace.ProductSpace method), 238
 element() (odl.set.sets.CartesianProduct method), 248
 element() (odl.set.sets.ComplexNumbers method), 250
 element() (odl.set.sets.EmptySet method), 251
 element() (odl.set.sets.Field method), 252
 element() (odl.set.sets.Integers method), 253
 element() (odl.set.sets.RealNumbers method), 254
 element() (odl.set.sets.Set method), 255
 element() (odl.set.sets.Strings method), 256
 element() (odl.set.sets.UniversalSet method), 257
 element() (odl.set.space.LinearSpace method), 261
 element() (odl.set.space.UniversalSpace method), 269
 element() (odl.space.base_ntuples.FnBase method), 296
 element() (odl.space.base_ntuples.NtuplesBase method), 307
 element() (odl.space.cu_ntuples.CudaFn method), 317
 element() (odl.space.cu_ntuples.CudaNtuples method), 341
 element() (odl.space.fspace.FunctionSet method), 352
 element() (odl.space.fspace.FunctionSpace method), 359
 element() (odl.space.ntuples.Fn method), 374
 element() (odl.space.ntuples.Ntuples method), 406
 element-like, 34
 element_type (odl.discr.discretization.Discretization attribute), 83
 element_type (odl.discr.discretization.RawDiscretization attribute), 95
 element_type (odl.discr.lp_discr.DiscreteLp attribute), 124
 element_type (odl.set.pspace.ProductSpace attribute), 235
 element_type (odl.set.space.LinearSpace attribute), 258
 element_type (odl.set.space.UniversalSpace attribute), 266
 element_type (odl.space.base_ntuples.FnBase attribute), 292
 element_type (odl.space.base_ntuples.NtuplesBase attribute), 305
 element_type (odl.space.cu_ntuples.CudaFn attribute), 311
 element_type (odl.space.cu_ntuples.CudaNtuples attribute), 339
 element_type (odl.space.fspace.FunctionSet attribute), 351
 element_type (odl.space.fspace.FunctionSpace attribute), 356
 element_type (odl.space.ntuples.Fn attribute), 367
 element_type (odl.space.ntuples.Ntuples attribute), 404
 ellipse_phantom_2d() (in module odl.util.phantom), 526
 ellipse_phantom_3d() (in module odl.util.phantom), 526
 EmptySet (class in odl.set.sets), 250
 end (odl.discr.partition.RectPartition attribute), 146
 end (odl.set.domain.IntervalProd attribute), 225
 equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
 equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 equal() (odl.util.ufuncs.NtuplesUFuncs method), 575
 equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 equals() (odl.diagnostics.space.SpaceTest method), 45
 equiv() (odl.space.base_ntuples.FnWeightingBase method), 304
 equiv() (odl.space.cu_ntuples.CudaFnConstWeighting method), 322
 equiv() (odl.space.cu_ntuples.CudaFnCustomDist method), 324
 equiv() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 325
 equiv() (odl.space.cu_ntuples.CudaFnCustomNorm method), 327
 equiv() (odl.space.cu_ntuples.CudaFnNoWeighting method), 329
 equiv() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 338
 equiv() (odl.space.ntuples.FnConstWeighting method), 379
 equiv() (odl.space.ntuples.FnCustomDist method), 381
 equiv() (odl.space.ntuples.FnCustomInnerProduct method), 383
 equiv() (odl.space.ntuples.FnCustomNorm method), 385
 equiv() (odl.space.ntuples.FnMatrixWeighting method), 387
 equiv() (odl.space.ntuples.FnNoWeighting method), 389
 equiv() (odl.space.ntuples.FnVectorWeighting method), 400
 euler_matrix() (in module odl.tomo.util.utility), 482

- `exp()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
 - `exp()` (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 - `exp()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 - `exp()` (odl.util.ufuncs.NtuplesUFuncs method), 575
 - `exp()` (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 - `exp2()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 537
 - `exp2()` (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 - `exp2()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 - `exp2()` (odl.util.ufuncs.NtuplesUFuncs method), 575
 - `exp2()` (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 - `exp_zero_seq()` (in module odl.solvers.iterative.iterative), 280
 - `expm1()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 - `expm1()` (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 - `expm1()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 - `expm1()` (odl.util.ufuncs.NtuplesUFuncs method), 576
 - `expm1()` (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 - `exponent` (odl.dscr.lp_discr.DiscreteLp attribute), 124
 - `exponent` (odl.space.base_ntuples.FnWeightingBase attribute), 303
 - `exponent` (odl.space.cu_ntuples.CudaFn attribute), 311
 - `exponent` (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 321
 - `exponent` (odl.space.cu_ntuples.CudaFnCustomDist attribute), 323
 - `exponent` (odl.space.cu_ntuples.CudaFnCustomInnerProduct attribute), 324
 - `exponent` (odl.space.cu_ntuples.CudaFnCustomNorm attribute), 326
 - `exponent` (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 328
 - `exponent` (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 337
 - `exponent` (odl.space.ntuples.Fn attribute), 367
 - `exponent` (odl.space.ntuples.FnConstWeighting attribute), 379
 - `exponent` (odl.space.ntuples.FnCustomDist attribute), 381
 - `exponent` (odl.space.ntuples.FnCustomInnerProduct attribute), 382
 - `exponent` (odl.space.ntuples.FnCustomNorm attribute), 384
 - `exponent` (odl.space.ntuples.FnMatrixWeighting attribute), 386
 - `exponent` (odl.space.ntuples.FnNoWeighting attribute), 389
 - `exponent` (odl.space.ntuples.FnVectorWeighting attribute), 399
 - `extension`, 34
 - `extension` (odl.dscr.discretization.Discretization attribute), 83
 - `extension` (odl.dscr.discretization.DiscretizationVector attribute), 90
 - `extension` (odl.dscr.discretization.RawDiscretization attribute), 95
 - `extension` (odl.dscr.discretization.RawDiscretizationVector attribute), 98
 - `extension` (odl.dscr.lp_discr.DiscreteLp attribute), 124
 - `extension` (odl.dscr.lp_discr.DiscreteLpVector attribute), 132
 - `extent()` (odl.dscr.grid.RegularGrid method), 108
 - `extent()` (odl.dscr.grid.TensorGrid method), 118
 - `extent()` (odl.dscr.partition.RectPartition method), 147
 - `extent()` (odl.set.domain.IntervalProd method), 232
- ## F
- `fail()` (odl.util.testutils.FailCounter method), 529
 - `FailCounter` (class in odl.util.testutils), 528
 - `FanFlatGeometry` (class in odl.tomo.geometry.fanbeam), 449
 - `fast_1d_tensor_mult()` (in module odl.util.numerics), 524
 - `Field` (class in odl.set.sets), 251
 - `field` (odl.dscr.discretization.Discretization attribute), 83
 - `field` (odl.dscr.lp_discr.DiscreteLp attribute), 125
 - `field` (odl.set.pspace.ProductSpace attribute), 235
 - `field` (odl.set.sets.ComplexNumbers attribute), 249
 - `field` (odl.set.sets.Field attribute), 251
 - `field` (odl.set.sets.RealNumbers attribute), 253
 - `field` (odl.set.space.LinearSpace attribute), 258
 - `field` (odl.set.space.UniversalSpace attribute), 266
 - `field` (odl.space.base_ntuples.FnBase attribute), 292
 - `field` (odl.space.cu_ntuples.CudaFn attribute), 311
 - `field` (odl.space.fspace.FunctionSpace attribute), 356
 - `field` (odl.space.ntuples.Fn attribute), 367
 - `field()` (odl.diagnostics.space.SpaceTest method), 45
 - `finite_diff()` (in module odl.dscr.dscr_ops), 80
 - `Flat1dDetector` (class in odl.tomo.geometry.detector), 443
 - `Flat2dDetector` (class in odl.tomo.geometry.detector), 445
 - `FlatDetector` (class in odl.tomo.geometry.detector), 447
 - `floor()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 - `floor()` (odl.util.ufuncs.DiscreteLpUFuncs method), 550
 - `floor()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 - `floor()` (odl.util.ufuncs.NtuplesUFuncs method), 576
 - `floor()` (odl.util.ufuncs.ProductSpaceUFuncs method), 588
 - `floor_divide()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 - `floor_divide()` (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 - `floor_divide()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 - `floor_divide()` (odl.util.ufuncs.NtuplesUFuncs method), 576
 - `floor_divide()` (odl.util.ufuncs.ProductSpaceUFuncs method), 589

- fmax() (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 fmax() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 fmax() (odl.util.ufuncs.NtuplesBaseUFuncs method), 563
 fmax() (odl.util.ufuncs.NtuplesUFuncs method), 576
 fmax() (odl.util.ufuncs.ProductSpaceUFuncs method), 589
 fmin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 fmin() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 fmin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 fmin() (odl.util.ufuncs.NtuplesUFuncs method), 576
 fmin() (odl.util.ufuncs.ProductSpaceUFuncs method), 589
 fmod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 fmod() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 fmod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 fmod() (odl.util.ufuncs.NtuplesUFuncs method), 576
 fmod() (odl.util.ufuncs.ProductSpaceUFuncs method), 589
 Fn (class in odl.space.ntuples), 367
 FnBase (class in odl.space.base_ntuples), 292
 FnBaseVector (class in odl.space.base_ntuples), 298
 FnConstWeighting (class in odl.space.ntuples), 378
 FnCustomDist (class in odl.space.ntuples), 380
 FnCustomInnerProduct (class in odl.space.ntuples), 382
 FnCustomNorm (class in odl.space.ntuples), 384
 FnMatrixWeighting (class in odl.space.ntuples), 385
 FnNoWeighting (class in odl.space.ntuples), 388
 FnVector (class in odl.space.ntuples), 390
 FnVectorWeighting (class in odl.space.ntuples), 399
 FnWeightingBase (class in odl.space.base_ntuples), 302
 ForEachPartial (class in odl.solvers.util.partial), 287
 ForwardProjectorByBinWrapper (class in odl.tomo.backends.stir_bindings), 426
 FourierTransform (class in odl.trafos.fourier), 493
 FourierTransformInverse (class in odl.trafos.fourier), 498
 FunctionalLeftVectorMult (class in odl.operator.operator), 176
 FunctionSet (class in odl.space.fspace), 351
 FunctionSetMapping (class in odl.dscr.dscr_mappings), 48
 FunctionSetVector (class in odl.space.fspace), 353
 FunctionSpace (class in odl.space.fspace), 356
 FunctionSpaceVector (class in odl.space.fspace), 362
- ## G
- gauss_newton() (in module odl.solvers.iterative.iterative), 280
 Geometry (class in odl.tomo.geometry.geometry), 458
 geometry (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
 geometry (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
 Gradient (class in odl.dscr.dscr_ops), 70
 greater() (odl.util.ufuncs.CudaNtuplesUFuncs method), 538
 greater() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 greater() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 greater() (odl.util.ufuncs.NtuplesUFuncs method), 576
 greater() (odl.util.ufuncs.ProductSpaceUFuncs method), 589
 greater_equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 greater_equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 greater_equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 greater_equal() (odl.util.ufuncs.NtuplesUFuncs method), 577
 greater_equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 589
 grid (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 48
 grid (odl.dscr.dscr_mappings.LinearInterpolation attribute), 52
 grid (odl.dscr.dscr_mappings.NearestInterpolation attribute), 55
 grid (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
 grid (odl.dscr.dscr_mappings.PointCollocation attribute), 63
 grid (odl.dscr.lp_dscr.DiscreteLp attribute), 125
 grid (odl.dscr.partition.RectPartition attribute), 146
 grid (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
 grid (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 435
 grid (odl.tomo.geometry.detector.CircleSectionDetector attribute), 440
 grid (odl.tomo.geometry.detector.Detector attribute), 441
 grid (odl.tomo.geometry.detector.Flat1dDetector attribute), 443
 grid (odl.tomo.geometry.detector.Flat2dDetector attribute), 445
 grid (odl.tomo.geometry.detector.FlatDetector attribute), 447
 grid (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
 grid (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
 grid (odl.tomo.geometry.geometry.Geometry attribute), 459
 grid (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462

grid (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 grid (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469
 grid (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473

H

halfcomplex (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
 halfcomplex (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
 halfcomplex (odl.trafos.fourier.FourierTransform attribute), 493
 halfcomplex (odl.trafos.fourier.FourierTransformInverse attribute), 498
 HelicalConeFlatGeometry (class in odl.tomo.geometry.conebeam), 434
 hypot() (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 hypot() (odl.util.ufuncs.DiscreteLpUFuncs method), 551
 hypot() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 hypot() (odl.util.ufuncs.NtuplesUFuncs method), 577
 hypot() (odl.util.ufuncs.ProductSpaceUFuncs method), 589

I

IdentityOperator (class in odl.operator.default_ops), 155
 imag (odl.discr.lp_discr.DiscreteLpVector attribute), 133
 imag (odl.space.fspace.FunctionSpaceVector attribute), 363
 imag (odl.space.ntuples.FnVector attribute), 392
 impl (odl.space.base_ntuples.FnWeightingBase attribute), 303
 impl (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 321
 impl (odl.space.cu_ntuples.CudaFnCustomDist attribute), 323
 impl (odl.space.cu_ntuples.CudaFnCustomInnerProduct attribute), 324
 impl (odl.space.cu_ntuples.CudaFnCustomNorm attribute), 326
 impl (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 328
 impl (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 337
 impl (odl.space.ntuples.FnConstWeighting attribute), 379
 impl (odl.space.ntuples.FnCustomDist attribute), 381
 impl (odl.space.ntuples.FnCustomInnerProduct attribute), 382
 impl (odl.space.ntuples.FnCustomNorm attribute), 384
 impl (odl.space.ntuples.FnMatrixWeighting attribute), 386

impl (odl.space.ntuples.FnNoWeighting attribute), 389
 impl (odl.space.ntuples.FnVectorWeighting attribute), 399
 impl (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
 impl (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
 impl (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
 impl (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
 impl (odl.trafos.fourier.FourierTransform attribute), 493
 impl (odl.trafos.fourier.FourierTransformInverse attribute), 499
 implementation_cache (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
 implementation_cache (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
 implementation_cache (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
 implementation_cache (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
 implementation_cache (odl.tomo.geometry.geometry.Geometry attribute), 459
 implementation_cache (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462
 implementation_cache (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 implementation_cache (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 469
 implementation_cache (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
 in-place evaluation, 34
 index (odl.operator.pspace_ops.ComponentProjection attribute), 209
 index (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 212
 indicate_proj_axis() (in module odl.util.phantom), 526
 init_fftw_plan() (odl.trafos.fourier.DiscreteFourierTransform method), 487
 init_fftw_plan() (odl.trafos.fourier.DiscreteFourierTransformInverse method), 491
 init_fftw_plan() (odl.trafos.fourier.FourierTransform method), 496
 init_fftw_plan() (odl.trafos.fourier.FourierTransformInverse method), 501
 inner (odl.space.ntuples.FnCustomInnerProduct attribute), 382
 inner() (odl.diagnostics.space.SpaceTest method), 46
 inner() (odl.discr.discretization.Discretization method), 87
 inner() (odl.discr.discretization.DiscretizationVector method), 92
 inner() (odl.discr.lp_discr.DiscreteLp method), 129

- inner() (odl.discr.lp_discr.DiscreteLpVector method), 137
- inner() (odl.set.pspace.ProductSpace method), 239
- inner() (odl.set.pspace.ProductSpaceVector method), 245
- inner() (odl.set.space.LinearSpace method), 261
- inner() (odl.set.space.LinearSpaceVector method), 265
- inner() (odl.set.space.UniversalSpace method), 269
- inner() (odl.space.base_ntuples.FnBase method), 296
- inner() (odl.space.base_ntuples.FnBaseVector method), 301
- inner() (odl.space.base_ntuples.FnWeightingBase method), 304
- inner() (odl.space.cu_ntuples.CudaFn method), 318
- inner() (odl.space.cu_ntuples.CudaFnConstWeighting method), 322
- inner() (odl.space.cu_ntuples.CudaFnCustomDist method), 324
- inner() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 325
- inner() (odl.space.cu_ntuples.CudaFnCustomNorm method), 327
- inner() (odl.space.cu_ntuples.CudaFnNoWeighting method), 329
- inner() (odl.space.cu_ntuples.CudaFnVector method), 335
- inner() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 338
- inner() (odl.space.cu_ntuples.CudaNtuplesVector method), 348
- inner() (odl.space.fspace.FunctionSpace method), 360
- inner() (odl.space.fspace.FunctionSpaceVector method), 365
- inner() (odl.space.ntuples.Fn method), 375
- inner() (odl.space.ntuples.FnConstWeighting method), 380
- inner() (odl.space.ntuples.FnCustomDist method), 381
- inner() (odl.space.ntuples.FnCustomNorm method), 385
- inner() (odl.space.ntuples.FnMatrixWeighting method), 387
- inner() (odl.space.ntuples.FnNoWeighting method), 389
- inner() (odl.space.ntuples.FnVector method), 397
- inner() (odl.space.ntuples.FnVectorWeighting method), 400
- InnerProductOperator (class in odl.operator.default_ops), 158
- insert() (odl.discr.grid.RegularGrid method), 108
- insert() (odl.discr.grid.TensorGrid method), 118
- insert() (odl.discr.partition.RectPartition method), 147
- insert() (odl.set.domain.IntervalProd method), 232
- Integers (class in odl.set.sets), 252
- interp (odl.discr.lp_discr.DiscreteLp attribute), 125
- Interval() (in module odl.set.domain), 234
- IntervalProd (class in odl.set.domain), 225
- inverse (odl.discr.discr_mappings.FunctionSetMapping attribute), 49
- inverse (odl.discr.discr_mappings.LinearInterpolation attribute), 52
- inverse (odl.discr.discr_mappings.NearestInterpolation attribute), 56
- inverse (odl.discr.discr_mappings.PerAxisInterpolation attribute), 60
- inverse (odl.discr.discr_mappings.PointCollocation attribute), 63
- inverse (odl.discr.discr_ops.Divergence attribute), 67
- inverse (odl.discr.discr_ops.Gradient attribute), 70
- inverse (odl.discr.discr_ops.Laplacian attribute), 73
- inverse (odl.discr.discr_ops.PartialDerivative attribute), 76
- inverse (odl.operator.default_ops.ConstantOperator attribute), 153
- inverse (odl.operator.default_ops.IdentityOperator attribute), 155
- inverse (odl.operator.default_ops.InnerProductOperator attribute), 159
- inverse (odl.operator.default_ops.LinCombOperator attribute), 161
- inverse (odl.operator.default_ops.MultiplyOperator attribute), 165
- inverse (odl.operator.default_ops.ResidualOperator attribute), 167
- inverse (odl.operator.default_ops.ScalingOperator attribute), 170
- inverse (odl.operator.default_ops.ZeroOperator attribute), 173
- inverse (odl.operator.operator.FunctionalLeftVectorMult attribute), 177
- inverse (odl.operator.operator.Operator attribute), 181
- inverse (odl.operator.operator.OperatorComp attribute), 184
- inverse (odl.operator.operator.OperatorLeftScalarMult attribute), 187
- inverse (odl.operator.operator.OperatorLeftVectorMult attribute), 190
- inverse (odl.operator.operator.OperatorPointwiseProduct attribute), 192
- inverse (odl.operator.operator.OperatorRightScalarMult attribute), 195
- inverse (odl.operator.operator.OperatorRightVectorMult attribute), 197
- inverse (odl.operator.operator.OperatorSum attribute), 200
- inverse (odl.operator.pspace_ops.BroadcastOperator attribute), 205
- inverse (odl.operator.pspace_ops.ComponentProjection attribute), 209
- inverse (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 212
- inverse (odl.operator.pspace_ops.ProductSpaceOperator attribute), 217

- inverse (odl.operator.pspace_ops.ReductionOperator attribute), 221
- inverse (odl.space.fspace.FunctionSetVector attribute), 353
- inverse (odl.space.fspace.FunctionSpaceVector attribute), 363
- inverse (odl.space.ntuples.MatVecOperator attribute), 402
- inverse (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapping attribute), 424
- inverse (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapping attribute), 426
- inverse (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
- inverse (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
- inverse (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
- inverse (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
- inverse (odl.trafos.fourier.FourierTransform attribute), 494
- inverse (odl.trafos.fourier.FourierTransformInverse attribute), 499
- inverse (odl.trafos.wavelet.WaveletTransform attribute), 510
- inverse (odl.trafos.wavelet.WaveletTransformInverse attribute), 513
- inverse_reciprocal() (in module odl.trafos.fourier), 505
- invert() (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
- invert() (odl.util.ufuncs.DiscreteLpUFuncs method), 552
- invert() (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
- invert() (odl.util.ufuncs.NtuplesUFuncs method), 577
- invert() (odl.util.ufuncs.ProductSpaceUFuncs method), 590
- is_biorthogonal (odl.trafos.wavelet.WaveletTransform attribute), 510
- is_biorthogonal (odl.trafos.wavelet.WaveletTransformInverse attribute), 513
- is_cn (odl.dscr.discretization.Discretization attribute), 83
- is_cn (odl.dscr.lp_dscr.DiscreteLp attribute), 125
- is_cn (odl.space.base_ntuples.FnBase attribute), 293
- is_cn (odl.space.cu_ntuples.CudaFn attribute), 311
- is_cn (odl.space.ntuples.Fn attribute), 367
- is_complex_floating_dtype() (in module odl.util.utility), 599
- is_floating_dtype() (in module odl.util.utility), 599
- is_functional (odl.dscr.dscr_mappings.FunctionSetMappings attribute), 49
- is_functional (odl.dscr.dscr_mappings.LinearInterpolation attribute), 52
- is_functional (odl.dscr.dscr_mappings.NearestInterpolation attribute), 56
- is_functional (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
- is_functional (odl.dscr.dscr_mappings.PointCollocation attribute), 64
- is_functional (odl.dscr.dscr_ops.Divergence attribute), 68
- is_functional (odl.dscr.dscr_ops.Gradient attribute), 71
- is_functional (odl.dscr.dscr_ops.Laplacian attribute), 74
- is_functional (odl.dscr.dscr_ops.PartialDerivative attribute), 76
- is_functional (odl.operator.default_ops.ConstantOperator attribute), 153
- is_functional (odl.operator.default_ops.IdentityOperator attribute), 156
- is_functional (odl.operator.default_ops.InnerProductOperator attribute), 159
- is_functional (odl.operator.default_ops.LinCombOperator attribute), 162
- is_functional (odl.operator.default_ops.MultiplyOperator attribute), 165
- is_functional (odl.operator.default_ops.ResidualOperator attribute), 168
- is_functional (odl.operator.default_ops.ScalingOperator attribute), 171
- is_functional (odl.operator.default_ops.ZeroOperator attribute), 174
- is_functional (odl.operator.operator.FunctionalLeftVectorMult attribute), 177
- is_functional (odl.operator.operator.Operator attribute), 181
- is_functional (odl.operator.operator.OperatorComp attribute), 185
- is_functional (odl.operator.operator.OperatorLeftScalarMult attribute), 187
- is_functional (odl.operator.operator.OperatorLeftVectorMult attribute), 190
- is_functional (odl.operator.operator.OperatorPointwiseProduct attribute), 192
- is_functional (odl.operator.operator.OperatorRightScalarMult attribute), 195
- is_functional (odl.operator.operator.OperatorRightVectorMult attribute), 198
- is_functional (odl.operator.operator.OperatorSum attribute), 200
- is_functional (odl.operator.pspace_ops.BroadcastOperator attribute), 205
- is_functional (odl.operator.pspace_ops.ComponentProjection attribute), 209
- is_functional (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 212
- is_functional (odl.operator.pspace_ops.ProductSpaceOperator attribute), 217
- is_functional (odl.operator.pspace_ops.ReductionOperator attribute), 221

- `is_functional` (odl.space.fspace.FunctionSetVector attribute), 353
- `is_functional` (odl.space.fspace.FunctionSpaceVector attribute), 363
- `is_functional` (odl.space.ntuples.MatVecOperator attribute), 402
- `is_functional` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 424
- `is_functional` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 427
- `is_functional` (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
- `is_functional` (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
- `is_functional` (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
- `is_functional` (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
- `is_functional` (odl.trafos.fourier.FourierTransform attribute), 494
- `is_functional` (odl.trafos.fourier.FourierTransformInverse attribute), 499
- `is_functional` (odl.trafos.wavelet.WaveletTransform attribute), 510
- `is_functional` (odl.trafos.wavelet.WaveletTransformInverse attribute), 514
- `is_int_dtype()` (in module odl.util.utility), 599
- `is_linear` (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 49
- `is_linear` (odl.dscr.dscr_mappings.LinearInterpolation attribute), 52
- `is_linear` (odl.dscr.dscr_mappings.NearestInterpolation attribute), 56
- `is_linear` (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
- `is_linear` (odl.dscr.dscr_mappings.PointCollocation attribute), 64
- `is_linear` (odl.dscr.dscr_ops.Divergence attribute), 68
- `is_linear` (odl.dscr.dscr_ops.Gradient attribute), 71
- `is_linear` (odl.dscr.dscr_ops.Laplacian attribute), 74
- `is_linear` (odl.dscr.dscr_ops.PartialDerivative attribute), 76
- `is_linear` (odl.operator.default_ops.ConstantOperator attribute), 153
- `is_linear` (odl.operator.default_ops.IdentityOperator attribute), 156
- `is_linear` (odl.operator.default_ops.InnerProductOperator attribute), 159
- `is_linear` (odl.operator.default_ops.LinCombOperator attribute), 162
- `is_linear` (odl.operator.default_ops.MultiplyOperator attribute), 165
- `is_linear` (odl.operator.default_ops.ResidualOperator attribute), 168
- `is_linear` (odl.operator.default_ops.ScalingOperator attribute), 171
- `is_linear` (odl.operator.default_ops.ZeroOperator attribute), 174
- `is_linear` (odl.operator.operator.FunctionalLeftVectorMult attribute), 177
- `is_linear` (odl.operator.operator.Operator attribute), 181
- `is_linear` (odl.operator.operator.OperatorComp attribute), 184
- `is_linear` (odl.operator.operator.OperatorLeftScalarMult attribute), 187
- `is_linear` (odl.operator.operator.OperatorLeftVectorMult attribute), 190
- `is_linear` (odl.operator.operator.OperatorPointwiseProduct attribute), 192
- `is_linear` (odl.operator.operator.OperatorRightScalarMult attribute), 195
- `is_linear` (odl.operator.operator.OperatorRightVectorMult attribute), 198
- `is_linear` (odl.operator.operator.OperatorSum attribute), 200
- `is_linear` (odl.operator.pspace_ops.BroadcastOperator attribute), 205
- `is_linear` (odl.operator.pspace_ops.ComponentProjection attribute), 209
- `is_linear` (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 212
- `is_linear` (odl.operator.pspace_ops.ProductSpaceOperator attribute), 217
- `is_linear` (odl.operator.pspace_ops.ReductionOperator attribute), 222
- `is_linear` (odl.space.fspace.FunctionSetVector attribute), 354
- `is_linear` (odl.space.fspace.FunctionSpaceVector attribute), 363
- `is_linear` (odl.space.ntuples.MatVecOperator attribute), 402
- `is_linear` (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 424
- `is_linear` (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 427
- `is_linear` (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
- `is_linear` (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
- `is_linear` (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
- `is_linear` (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 489
- `is_linear` (odl.trafos.fourier.FourierTransform attribute), 494
- `is_linear` (odl.trafos.fourier.FourierTransformInverse attribute), 499

- `is_linear` (odl.trafos.wavelet.WaveletTransform attribute), 510
 - `is_linear` (odl.trafos.wavelet.WaveletTransformInverse attribute), 514
 - `is_orthogonal` (odl.trafos.wavelet.WaveletTransform attribute), 510
 - `is_orthogonal` (odl.trafos.wavelet.WaveletTransformInverse attribute), 514
 - `is_real_dtype()` (in module odl.util.utility), 599
 - `is_real_floating_dtype()` (in module odl.util.utility), 600
 - `is_regular` (odl.dscr.partition.RectPartition attribute), 146
 - `is_rn` (odl.dscr.discretization.Discretization attribute), 83
 - `is_rn` (odl.dscr.lp_discr.DiscreteLp attribute), 125
 - `is_rn` (odl.space.base_ntuples.FnBase attribute), 293
 - `is_rn` (odl.space.cu_ntuples.CudaFn attribute), 311
 - `is_rn` (odl.space.ntuples.Fn attribute), 368
 - `is_rotation_matrix()` (in module odl.tomo.util.utility), 483
 - `is_scalar_dtype()` (in module odl.util.utility), 600
 - `is_subdict()` (in module odl.util.testutils), 531
 - `is_subgrid()` (odl.dscr.grid.RegularGrid method), 109
 - `is_subgrid()` (odl.dscr.grid.TensorGrid method), 118
 - `is_valid_input_array()` (in module odl.util.vectorization), 603
 - `is_valid_input_meshgrid()` (in module odl.util.vectorization), 604
 - `is_weighted` (odl.dscr.discretization.Discretization attribute), 83
 - `is_weighted` (odl.dscr.lp_discr.DiscreteLp attribute), 125
 - `is_weighted` (odl.space.cu_ntuples.CudaFn attribute), 311
 - `is_weighted` (odl.space.ntuples.Fn attribute), 368
 - `isfinite()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 - `isfinite()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `isfinite()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 564
 - `isfinite()` (odl.util.ufuncs.NtuplesUFuncs method), 577
 - `isfinite()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `isinf()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 - `isinf()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `isinf()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
 - `isinf()` (odl.util.ufuncs.NtuplesUFuncs method), 577
 - `isinf()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `isnan()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 - `isnan()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `isnan()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
 - `isnan()` (odl.util.ufuncs.NtuplesUFuncs method), 577
 - `isnan()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `itemsizes` (odl.dscr.discretization.DiscretizationVector attribute), 90
 - `itemsizes` (odl.dscr.lp_discr.DiscreteLpVector attribute), 133
 - `itemsizes` (odl.space.base_ntuples.FnBaseVector attribute), 299
 - `itemsizes` (odl.space.base_ntuples.NtuplesBaseVector attribute), 307
 - `itemsizes` (odl.space.cu_ntuples.CudaFnVector attribute), 331
 - `itemsizes` (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
 - `itemsizes` (odl.space.ntuples.FnVector attribute), 392
 - `itemsizes` (odl.space.ntuples.NtuplesVector attribute), 408
- ## L
- `landweber()` (in module odl.solvers.iterative.iterative), 281
 - `Laplacian` (class in odl.dscr.dscr_ops), 73
 - `left_shift()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 539
 - `left_shift()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `left_shift()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
 - `left_shift()` (odl.util.ufuncs.NtuplesUFuncs method), 577
 - `left_shift()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `length` (odl.set.domain.IntervalProd attribute), 226
 - `length` (odl.set.sets.Strings attribute), 256
 - `less()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
 - `less()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `less()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
 - `less()` (odl.util.ufuncs.NtuplesUFuncs method), 578
 - `less()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `less_equal()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
 - `less_equal()` (odl.util.ufuncs.DiscreteLpUFuncs method), 552
 - `less_equal()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
 - `less_equal()` (odl.util.ufuncs.NtuplesUFuncs method), 578
 - `less_equal()` (odl.util.ufuncs.ProductSpaceUFuncs method), 590
 - `lincomb()` (odl.dscr.discretization.Discretization method), 87
 - `lincomb()` (odl.dscr.discretization.DiscretizationVector method), 93
 - `lincomb()` (odl.dscr.lp_discr.DiscreteLp method), 129
 - `lincomb()` (odl.dscr.lp_discr.DiscreteLpVector method), 137
 - `lincomb()` (odl.set.pspace.ProductSpace method), 239

- `lincomb()` (odl.set.pspace.ProductSpaceVector method), 246
- `lincomb()` (odl.set.space.LinearSpace method), 261
- `lincomb()` (odl.set.space.LinearSpaceVector method), 265
- `lincomb()` (odl.set.space.UniversalSpace method), 269
- `lincomb()` (odl.space.base_ntuples.FnBase method), 296
- `lincomb()` (odl.space.base_ntuples.FnBaseVector method), 301
- `lincomb()` (odl.space.cu_ntuples.CudaFn method), 318
- `lincomb()` (odl.space.cu_ntuples.CudaFnVector method), 335
- `lincomb()` (odl.space.cu_ntuples.CudaNtuplesVector method), 348
- `lincomb()` (odl.space.fspace.FunctionSpace method), 360
- `lincomb()` (odl.space.fspace.FunctionSpaceVector method), 365
- `lincomb()` (odl.space.ntuples.Fn method), 375
- `lincomb()` (odl.space.ntuples.FnVector method), 397
- `LinCombOperator` (class in odl.operator.default_ops), 161
- `linear()` (odl.diagnostics.operator.OperatorTest method), 43
- `LinearInterpolation` (class in odl.discr.dscr_mappings), 52
- `linearity()` (odl.diagnostics.space.SpaceTest method), 46
- `LinearSpace` (class in odl.set.space), 258
- `LinearSpaceNotImplementedError`, 263
- `LinearSpaceTypeError`, 263
- `LinearSpaceVector` (class in odl.set.space), 263
- `LineSearch` (class in odl.solvers.scalar.steplen), 285
- `log()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
- `log()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `log()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
- `log()` (odl.util.ufuncs.NtuplesUFuncs method), 578
- `log()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `log10()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
- `log10()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `log10()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 565
- `log10()` (odl.util.ufuncs.NtuplesUFuncs method), 578
- `log10()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `log1p()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
- `log1p()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `log1p()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `log1p()` (odl.util.ufuncs.NtuplesUFuncs method), 578
- `log1p()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `log2()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
- `log2()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `log2()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `log2()` (odl.util.ufuncs.NtuplesUFuncs method), 578
- `log2()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `logaddexp()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 540
- `logaddexp()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `logaddexp()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `logaddexp()` (odl.util.ufuncs.NtuplesUFuncs method), 578
- `logaddexp()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `logaddexp2()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
- `logaddexp2()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `logaddexp2()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `logaddexp2()` (odl.util.ufuncs.NtuplesUFuncs method), 579
- `logaddexp2()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `logical_and()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
- `logical_and()` (odl.util.ufuncs.DiscreteLpUFuncs method), 553
- `logical_and()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `logical_and()` (odl.util.ufuncs.NtuplesUFuncs method), 579
- `logical_and()` (odl.util.ufuncs.ProductSpaceUFuncs method), 591
- `logical_not()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
- `logical_not()` (odl.util.ufuncs.DiscreteLpUFuncs method), 554
- `logical_not()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `logical_not()` (odl.util.ufuncs.NtuplesUFuncs method), 579
- `logical_not()` (odl.util.ufuncs.ProductSpaceUFuncs method), 592
- `logical_or()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
- `logical_or()` (odl.util.ufuncs.DiscreteLpUFuncs method), 554
- `logical_or()` (odl.util.ufuncs.NtuplesBaseUFuncs method), 566
- `logical_or()` (odl.util.ufuncs.NtuplesUFuncs method), 579
- `logical_or()` (odl.util.ufuncs.ProductSpaceUFuncs method), 592
- `logical_xor()` (odl.util.ufuncs.CudaNtuplesUFuncs method), 541

logical_xor() (odl.util.ufuncs.DiscreteLpUFuncs method), 554
 logical_xor() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 logical_xor() (odl.util.ufuncs.NtuplesUFuncs method), 579
 logical_xor() (odl.util.ufuncs.ProductSpaceUFuncs method), 592

M

matrix (odl.space.ntuples.FnMatrixWeighting attribute), 386
 matrix (odl.space.ntuples.MatVecOperator attribute), 402
 matrix_issparse (odl.space.ntuples.FnMatrixWeighting attribute), 386
 matrix_issparse (odl.space.ntuples.MatVecOperator attribute), 402
 matrix_isvalid() (odl.space.ntuples.FnMatrixWeighting method), 387
 matrix_representation() (in module odl.operator.oputils), 203
 MatVecOperator (class in odl.space.ntuples), 401
 max() (odl.discr.grid.RegularGrid method), 110
 max() (odl.discr.grid.TensorGrid method), 119
 max() (odl.discr.partition.RectPartition method), 148
 max() (odl.set.domain.IntervalProd method), 232
 max() (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
 max() (odl.util.ufuncs.DiscreteLpUFuncs method), 554
 max() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 max() (odl.util.ufuncs.NtuplesUFuncs method), 579
 max() (odl.util.ufuncs.ProductSpaceUFuncs method), 592
 max_pt (odl.discr.grid.RegularGrid attribute), 103
 max_pt (odl.discr.grid.TensorGrid attribute), 113
 maximum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 541
 maximum() (odl.util.ufuncs.DiscreteLpUFuncs method), 554
 maximum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 maximum() (odl.util.ufuncs.NtuplesUFuncs method), 579
 maximum() (odl.util.ufuncs.ProductSpaceUFuncs method), 592
 measure() (odl.set.domain.IntervalProd method), 233
 meshgrid, 34
 meshgrid (odl.discr.grid.RegularGrid attribute), 103
 meshgrid (odl.discr.grid.TensorGrid attribute), 113
 meshgrid (odl.discr.lp_discr.DiscreteLp attribute), 125
 meshgrid (odl.discr.partition.RectPartition attribute), 146
 method() (in module odl.util.ufuncs), 596
 midpoint (odl.set.domain.IntervalProd attribute), 226
 min() (odl.discr.grid.RegularGrid method), 110
 min() (odl.discr.grid.TensorGrid method), 119
 min() (odl.discr.partition.RectPartition method), 148

min() (odl.set.domain.IntervalProd method), 233
 min() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 min() (odl.util.ufuncs.DiscreteLpUFuncs method), 554
 min() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 min() (odl.util.ufuncs.NtuplesUFuncs method), 580
 min() (odl.util.ufuncs.ProductSpaceUFuncs method), 592
 min_pt (odl.discr.grid.RegularGrid attribute), 104
 min_pt (odl.discr.grid.TensorGrid attribute), 113
 minimum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 minimum() (odl.util.ufuncs.DiscreteLpUFuncs method), 554
 minimum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 minimum() (odl.util.ufuncs.NtuplesUFuncs method), 580
 minimum() (odl.util.ufuncs.ProductSpaceUFuncs method), 592
 mod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 mod() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
 mod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 mod() (odl.util.ufuncs.NtuplesUFuncs method), 580
 mod() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
 modf() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 modf() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
 modf() (odl.util.ufuncs.NtuplesBaseUFuncs method), 567
 modf() (odl.util.ufuncs.NtuplesUFuncs method), 580
 modf() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
 motion_grid (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
 motion_grid (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
 motion_grid (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
 motion_grid (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 455
 motion_grid (odl.tomo.geometry.geometry.Geometry attribute), 459
 motion_grid (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 462
 motion_grid (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 motion_grid (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 470
 motion_grid (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
 motion_params (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
 motion_params (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
 motion_params (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450

motion_params (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 456
 motion_params (odl.tomo.geometry.geometry.Geometry attribute), 459
 motion_params (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 463
 motion_params (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 motion_params (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 470
 motion_params (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
 motion_partition (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
 motion_partition (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
 motion_partition (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 450
 motion_partition (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 456
 motion_partition (odl.tomo.geometry.geometry.Geometry attribute), 459
 motion_partition (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 463
 motion_partition (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 motion_partition (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 470
 motion_partition (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
 multiply() (odl.diagnostics.space.SpaceTest method), 46
 multiply() (odl.discr.discretization.Discretization method), 88
 multiply() (odl.discr.discretization.DiscretizationVector method), 93
 multiply() (odl.discr.lp_discr.DiscreteLp method), 130
 multiply() (odl.discr.lp_discr.DiscreteLpVector method), 137
 multiply() (odl.set.pspace.ProductSpace method), 240
 multiply() (odl.set.pspace.ProductSpaceVector method), 246
 multiply() (odl.set.space.LinearSpace method), 262
 multiply() (odl.set.space.LinearSpaceVector method), 265
 multiply() (odl.set.space.UniversalSpace method), 270
 multiply() (odl.space.base_ntuples.FnBase method), 297
 multiply() (odl.space.base_ntuples.FnBaseVector method), 301
 multiply() (odl.space.cu_ntuples.CudaFn method), 319
 multiply() (odl.space.cu_ntuples.CudaFnVector method), 335
 multiply() (odl.space.cu_ntuples.CudaNtuplesVector method), 348
 multiply() (odl.space.fspace.FunctionSpace method), 361
 multiply() (odl.space.fspace.FunctionSpaceVector method), 366
 multiply() (odl.space.ntuples.Fn method), 376
 multiply() (odl.space.ntuples.FnVector method), 398
 multiply() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 multiply() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
 multiply() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
 multiply() (odl.util.ufuncs.NtuplesUFuncs method), 580
 multiply() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
 MultiplyOperator (class in odl.operator.default_ops), 164
N
 nbytes (odl.discr.discretization.DiscretizationVector attribute), 90
 nbytes (odl.discr.discretization.RawDiscretizationVector attribute), 98
 nbytes (odl.discr.lp_discr.DiscreteLpVector attribute), 133
 nbytes (odl.space.base_ntuples.FnBaseVector attribute), 299
 nbytes (odl.space.base_ntuples.NtuplesBaseVector attribute), 307
 nbytes (odl.space.cu_ntuples.CudaFnVector attribute), 331
 nbytes (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
 nbytes (odl.space.ntuples.FnVector attribute), 392
 nbytes (odl.space.ntuples.NtuplesVector attribute), 408
 ndim (odl.discr.discretization.DiscretizationVector attribute), 90
 ndim (odl.discr.discretization.RawDiscretizationVector attribute), 98
 ndim (odl.discr.grid.RegularGrid attribute), 104
 ndim (odl.discr.grid.TensorGrid attribute), 114
 ndim (odl.discr.lp_discr.DiscreteLp attribute), 125
 ndim (odl.discr.lp_discr.DiscreteLpVector attribute), 133
 ndim (odl.discr.partition.RectPartition attribute), 147
 ndim (odl.set.domain.IntervalProd attribute), 226
 ndim (odl.space.base_ntuples.FnBaseVector attribute), 299
 ndim (odl.space.base_ntuples.NtuplesBaseVector attribute), 307
 ndim (odl.space.cu_ntuples.CudaFnVector attribute), 331
 ndim (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
 ndim (odl.space.ntuples.FnVector attribute), 392
 ndim (odl.space.ntuples.NtuplesVector attribute), 408
 ndim (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431

ndim (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
 ndim (odl.tomo.geometry.detector.CircleSectionDetector attribute), 440
 ndim (odl.tomo.geometry.detector.Detector attribute), 442
 ndim (odl.tomo.geometry.detector.Flat1dDetector attribute), 444
 ndim (odl.tomo.geometry.detector.Flat2dDetector attribute), 446
 ndim (odl.tomo.geometry.detector.FlatDetector attribute), 447
 ndim (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 451
 ndim (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 456
 ndim (odl.tomo.geometry.geometry.Geometry attribute), 459
 ndim (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 463
 ndim (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 466
 ndim (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 470
 ndim (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
 NearestInterpolation (class in odl.discr.discr_mappings), 55
 negative() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 negative() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
 negative() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
 negative() (odl.util.ufuncs.NtuplesUFuncs method), 580
 negative() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
 newtons_method() (in module odl.solvers.vector.newton), 291
 nn_variants (odl.discr.discr_mappings.PerAxisInterpolation attribute), 60
 norm (odl.space.ntuples.FnCustomNorm attribute), 384
 norm() (odl.diagnostics.operator.OperatorTest method), 43
 norm() (odl.diagnostics.space.SpaceTest method), 46
 norm() (odl.discr.discretization.Discretization method), 88
 norm() (odl.discr.discretization.DiscretizationVector method), 93
 norm() (odl.discr.lp_discr.DiscreteLp method), 130
 norm() (odl.discr.lp_discr.DiscreteLpVector method), 137
 norm() (odl.set.pspace.ProductSpace method), 240
 norm() (odl.set.pspace.ProductSpaceVector method), 246
 norm() (odl.set.space.LinearSpace method), 262
 norm() (odl.set.space.LinearSpaceVector method), 266
 norm() (odl.set.space.UniversalSpace method), 270
 norm() (odl.space.base_ntuples.FnBase method), 297
 norm() (odl.space.base_ntuples.FnBaseVector method), 302
 norm() (odl.space.base_ntuples.FnWeightingBase method), 304
 norm() (odl.space.cu_ntuples.CudaFn method), 319
 norm() (odl.space.cu_ntuples.CudaFnConstWeighting method), 322
 norm() (odl.space.cu_ntuples.CudaFnCustomDist method), 324
 norm() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 325
 norm() (odl.space.cu_ntuples.CudaFnCustomNorm method), 327
 norm() (odl.space.cu_ntuples.CudaFnNoWeighting method), 329
 norm() (odl.space.cu_ntuples.CudaFnVector method), 335
 norm() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 338
 norm() (odl.space.cu_ntuples.CudaNtuplesVector method), 348
 norm() (odl.space.fspace.FunctionSpace method), 361
 norm() (odl.space.fspace.FunctionSpaceVector method), 366
 norm() (odl.space.ntuples.Fn method), 376
 norm() (odl.space.ntuples.FnConstWeighting method), 380
 norm() (odl.space.ntuples.FnCustomDist method), 381
 norm() (odl.space.ntuples.FnCustomInnerProduct method), 383
 norm() (odl.space.ntuples.FnMatrixWeighting method), 387
 norm() (odl.space.ntuples.FnNoWeighting method), 390
 norm() (odl.space.ntuples.FnVector method), 398
 norm() (odl.space.ntuples.FnVectorWeighting method), 401
 normal (odl.tomo.geometry.detector.Flat1dDetector attribute), 444
 normal (odl.tomo.geometry.detector.Flat2dDetector attribute), 446
 not_equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 542
 not_equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
 not_equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
 not_equal() (odl.util.ufuncs.NtuplesUFuncs method), 580
 not_equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
 ntuple (odl.discr.discretization.DiscretizationVector attribute), 90

[ntuple \(odl.dscr.discretization.RawDiscretizationVector attribute\), 99](#)
[ntuple \(odl.dscr.lp_dscr.DiscreteLpVector attribute\), 133](#)
[Ntuples \(class in odl.space.ntuples\), 404](#)
[NtuplesBase \(class in odl.space.base_ntuples\), 305](#)
[NtuplesBaseUFuncs \(class in odl.util.ufuncs\), 558](#)
[NtuplesBaseVector \(class in odl.space.base_ntuples\), 307](#)
[NtuplesUFuncs \(class in odl.util.ufuncs\), 571](#)
[NtuplesVector \(class in odl.space.ntuples\), 407](#)

O

[one\(\) \(odl.dscr.discretization.Discretization method\), 88](#)
[one\(\) \(odl.dscr.lp_dscr.DiscreteLp method\), 130](#)
[one\(\) \(odl.set.pspace.ProductSpace method\), 240](#)
[one\(\) \(odl.set.space.LinearSpace method\), 262](#)
[one\(\) \(odl.set.space.UniversalSpace method\), 270](#)
[one\(\) \(odl.space.base_ntuples.FnBase method\), 298](#)
[one\(\) \(odl.space.cu_ntuples.CudaFn method\), 319](#)
[one\(\) \(odl.space.fspace.FunctionSpace method\), 361](#)
[one\(\) \(odl.space.ntuples.Fn method\), 376](#)
[one\(\) \(odl.space.ntuples.Ntuples method\), 407](#)
[OpDomainError, 179](#)
[operator, 34](#)
[Operator \(class in odl.operator.operator\), 179](#)
[OperatorComp \(class in odl.operator.operator\), 184](#)
[OperatorLeftScalarMult \(class in odl.operator.operator\), 186](#)
[OperatorLeftVectorMult \(class in odl.operator.operator\), 189](#)
[OperatorPointwiseProduct \(class in odl.operator.operator\), 192](#)
[OperatorRightScalarMult \(class in odl.operator.operator\), 194](#)
[OperatorRightVectorMult \(class in odl.operator.operator\), 197](#)
[operators \(odl.operator.pspace_ops.BroadcastOperator attribute\), 205](#)
[operators \(odl.operator.pspace_ops.ReductionOperator attribute\), 222](#)
[OperatorSum \(class in odl.operator.operator\), 199](#)
[OperatorTest \(class in odl.diagnostics.operator\), 42](#)
[OpNotImplementedError, 179](#)
[OpRangeError, 179](#)
[OptionalArgDecorator \(class in odl.util.vectorization\), 601](#)
[OpTypeError, 179](#)
[order, 34](#)
[order \(odl.dscr.dscr_mappings.FunctionSetMapping attribute\), 49](#)
[order \(odl.dscr.dscr_mappings.LinearInterpolation attribute\), 53](#)
[order \(odl.dscr.dscr_mappings.NearestInterpolation attribute\), 56](#)

[order \(odl.dscr.dscr_mappings.PerAxisInterpolation attribute\), 60](#)
[order \(odl.dscr.dscr_mappings.PointCollocation attribute\), 64](#)
[order \(odl.dscr.lp_dscr.DiscreteLp attribute\), 125](#)
[order \(odl.dscr.lp_dscr.DiscreteLpVector attribute\), 133](#)
[out-of-place evaluation, 34](#)
[out_dtype \(odl.space.fspace.FunctionSet attribute\), 351](#)
[out_dtype \(odl.space.fspace.FunctionSpace attribute\), 356](#)
[out_shape_from_array\(\) \(in odl.util.vectorization\), 604](#)
[out_shape_from_meshgrid\(\) \(in odl.util.vectorization\), 604](#)

P

[Parallel2dGeometry \(class in odl.tomo.geometry.parallel\), 461](#)
[Parallel3dAxisGeometry \(class in odl.tomo.geometry.parallel\), 465](#)
[Parallel3dGeometry \(class in odl.tomo.geometry.parallel\), 468](#)
[ParallelGeometry \(class in odl.tomo.geometry.parallel\), 472](#)
[params \(odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute\), 431](#)
[params \(odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute\), 436](#)
[params \(odl.tomo.geometry.detector.CircleSectionDetector attribute\), 440](#)
[params \(odl.tomo.geometry.detector.Detector attribute\), 442](#)
[params \(odl.tomo.geometry.detector.Flat1dDetector attribute\), 444](#)
[params \(odl.tomo.geometry.detector.Flat2dDetector attribute\), 446](#)
[params \(odl.tomo.geometry.detector.FlatDetector attribute\), 448](#)
[params \(odl.tomo.geometry.fanbeam.FanFlatGeometry attribute\), 451](#)
[params \(odl.tomo.geometry.geometry.DivergentBeamGeometry attribute\), 456](#)
[params \(odl.tomo.geometry.geometry.Geometry attribute\), 459](#)
[params \(odl.tomo.geometry.parallel.Parallel2dGeometry attribute\), 463](#)
[params \(odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute\), 466](#)
[params \(odl.tomo.geometry.parallel.Parallel3dGeometry attribute\), 470](#)
[params \(odl.tomo.geometry.parallel.ParallelGeometry attribute\), 473](#)
[Partial \(class in odl.solvers.util.partial\), 288](#)
[PartialDerivative \(class in odl.dscr.dscr_ops\), 76](#)

- partition (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 49
- partition (odl.dscr.dscr_mappings.LinearInterpolation attribute), 53
- partition (odl.dscr.dscr_mappings.NearestInterpolation attribute), 56
- partition (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
- partition (odl.dscr.dscr_mappings.PointCollocation attribute), 64
- partition (odl.dscr.lp_discr.DiscreteLp attribute), 125
- partition (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
- partition (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
- partition (odl.tomo.geometry.detector.CircleSectionDetector attribute), 440
- partition (odl.tomo.geometry.detector.Detector attribute), 442
- partition (odl.tomo.geometry.detector.Flat1dDetector attribute), 444
- partition (odl.tomo.geometry.detector.Flat2dDetector attribute), 446
- partition (odl.tomo.geometry.detector.FlatDetector attribute), 448
- partition (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 451
- partition (odl.tomo.geometry.geometry.DivergentBeamGeometry attribute), 456
- partition (odl.tomo.geometry.geometry.Geometry attribute), 459
- partition (odl.tomo.geometry.parallel.Parallel2dGeometry attribute), 463
- partition (odl.tomo.geometry.parallel.Parallel3dAxisGeometry attribute), 467
- partition (odl.tomo.geometry.parallel.Parallel3dGeometry attribute), 470
- partition (odl.tomo.geometry.parallel.ParallelGeometry attribute), 473
- parts (odl.set.pspace.ProductSpaceVector attribute), 243
- PerAxisInterpolation (class in odl.dscr.dscr_mappings), 59
- perpendicular_vector() (in module odl.tomo.util.utility), 483
- phantom() (in module odl.util.phantom), 527
- pitch (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 431
- pitch (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
- pitch_offset (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 432
- pitch_offset (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
- PointCollocation (class in odl.dscr.dscr_mappings), 63
- points() (odl.dscr.grid.RegularGrid method), 110
- points() (odl.dscr.grid.TensorGrid method), 120
- points() (odl.dscr.lp_discr.DiscreteLp method), 130
- points() (odl.dscr.partition.RectPartition method), 148
- power() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- power() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
- power() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
- power() (odl.util.ufuncs.NtuplesUFuncs method), 581
- power() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
- power_method_opnorm() (in module odl.operator.oputils), 204
- preload_first_arg() (in module odl.util.utility), 600
- PrintIterationPartial (class in odl.solvers.util.partial), 288
- PrintNormPartial (class in odl.solvers.util.partial), 288
- PrintTimingPartial (class in odl.solvers.util.partial), 289
- prod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- prod() (odl.util.ufuncs.DiscreteLpUFuncs method), 555
- prod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
- prod() (odl.util.ufuncs.NtuplesUFuncs method), 581
- prod() (odl.util.ufuncs.ProductSpaceUFuncs method), 593
- prod_op (odl.operator.pspace_ops.BroadcastOperator attribute), 206
- prod_op (odl.operator.pspace_ops.ReductionOperator attribute), 222
- ProductSpace (class in odl.set.pspace), 235
- ProductSpaceOperator (class in odl.operator.pspace_ops), 215
- ProductSpaceUFuncs (class in odl.util.ufuncs), 583
- ProductSpaceVector (class in odl.set.pspace), 242
- ProgressBar (class in odl.util.testutils), 529
- ProgressRange (class in odl.util.testutils), 530
- proximal_convexconjugate_1l() (in module odl.solvers.advanced.proximal_operators), 274
- proximal_convexconjugate_1l2() (in module odl.solvers.advanced.proximal_operators), 275
- proximal_nonnegativity() (in module odl.solvers.advanced.proximal_operators), 275
- proximal_zero() (in module odl.solvers.advanced.proximal_operators), 276
- pyfft_w_call() (in module odl.trafos.fourier), 506
- pywt_coeff_to_array() (in module odl.trafos.wavelet), 518

R

- rad2deg() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- rad2deg() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568
- rad2deg() (odl.util.ufuncs.NtuplesUFuncs method), 581
- rad2deg() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- range, 34
- range (odl.dscr.dscr_mappings.FunctionSetMapping attribute), 49
- range (odl.dscr.dscr_mappings.LinearInterpolation attribute), 53
- range (odl.dscr.dscr_mappings.NearestInterpolation attribute), 56
- range (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
- range (odl.dscr.dscr_mappings.PointCollocation attribute), 64
- range (odl.dscr.dscr_ops.Divergence attribute), 68
- range (odl.dscr.dscr_ops.Gradient attribute), 71
- range (odl.dscr.dscr_ops.Laplacian attribute), 74
- range (odl.dscr.dscr_ops.PartialDerivative attribute), 76
- range (odl.operator.default_ops.ConstantOperator attribute), 153
- range (odl.operator.default_ops.IdentityOperator attribute), 156
- range (odl.operator.default_ops.InnerProductOperator attribute), 159
- range (odl.operator.default_ops.LinCombOperator attribute), 162
- range (odl.operator.default_ops.MultiplyOperator attribute), 165
- range (odl.operator.default_ops.ResidualOperator attribute), 168
- range (odl.operator.default_ops.ScalingOperator attribute), 171
- range (odl.operator.default_ops.ZeroOperator attribute), 174
- range (odl.operator.operator.FunctionalLeftVectorMult attribute), 177
- range (odl.operator.operator.Operator attribute), 182
- range (odl.operator.operator.OperatorComp attribute), 185
- range (odl.operator.operator.OperatorLeftScalarMult attribute), 187
- range (odl.operator.operator.OperatorLeftVectorMult attribute), 190
- range (odl.operator.operator.OperatorPointwiseProduct attribute), 193
- range (odl.operator.operator.OperatorRightScalarMult attribute), 195
- range (odl.operator.operator.OperatorRightVectorMult attribute), 198
- range (odl.operator.operator.OperatorSum attribute), 200
- range (odl.operator.pspace_ops.BroadcastOperator attribute), 206
- range (odl.operator.pspace_ops.ComponentProjection attribute), 209
- range (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 213
- range (odl.operator.pspace_ops.ProductSpaceOperator attribute), 217
- range (odl.operator.pspace_ops.ReductionOperator attribute), 222
- range (odl.space.fspace.FunctionSet attribute), 351
- range (odl.space.fspace.FunctionSetVector attribute), 354
- range (odl.space.fspace.FunctionSpace attribute), 356
- range (odl.space.fspace.FunctionSpaceVector attribute), 363
- range (odl.space.ntuples.MatVecOperator attribute), 402
- range (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 424
- range (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 427
- range (odl.tomo.operators.ray_trafo.RayBackProjection attribute), 476
- range (odl.tomo.operators.ray_trafo.RayTransform attribute), 479
- range (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
- range (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 490
- range (odl.trafos.fourier.FourierTransform attribute), 494
- range (odl.trafos.fourier.FourierTransformInverse attribute), 499
- range (odl.trafos.wavelet.WaveletTransform attribute), 510
- range (odl.trafos.wavelet.WaveletTransformInverse attribute), 514
- RawDiscretization (class in odl.dscr.discretization), 94
- RawDiscretizationVector (class in odl.dscr.discretization), 97
- RayBackProjection (class in odl.tomo.operators.ray_trafo), 475
- RayTransform (class in odl.tomo.operators.ray_trafo), 478
- real (odl.dscr.lp_dscr.DiscreteLpVector attribute), 133
- real (odl.space.fspace.FunctionSpaceVector attribute), 363
- real (odl.space.ntuples.FnVector attribute), 392
- RealNumbers (class in odl.set.sets), 253
- reciprocal() (in module odl.trafos.fourier), 508
- reciprocal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- reciprocal() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- reciprocal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 568

- reciprocal() (odl.util.ufuncs.NtuplesUFuncs method), 581
- reciprocal() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- reciprocal_space() (in module odl.trafos.fourier), 509
- Rectangle() (in module odl.set.domain), 234
- RectPartition (class in odl.dscr.partition), 143
- ReductionOperator (class in odl.operator.pspace_ops), 220
- RegularGrid (class in odl.dscr.grid), 102
- remainder() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- remainder() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- remainder() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- remainder() (odl.util.ufuncs.NtuplesUFuncs method), 581
- remainder() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- ResidualOperator (class in odl.operator.default_ops), 167
- restriction, 34
- restriction (odl.dscr.discretization.Discretization attribute), 83
- restriction (odl.dscr.discretization.RawDiscretization attribute), 95
- restriction (odl.dscr.lp_dscr.DiscreteLp attribute), 126
- restriction() (odl.dscr.discretization.DiscretizationVector method), 93
- restriction() (odl.dscr.discretization.RawDiscretizationVector method), 100
- restriction() (odl.dscr.lp_dscr.DiscreteLpVector method), 137
- results (odl.solvers.util.partial.StorePartial attribute), 290
- right_shift() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- right_shift() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- right_shift() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- right_shift() (odl.util.ufuncs.NtuplesUFuncs method), 581
- right_shift() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- rint() (odl.util.ufuncs.CudaNtuplesUFuncs method), 543
- rint() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- rint() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- rint() (odl.util.ufuncs.NtuplesUFuncs method), 581
- rint() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- Rn() (in module odl.space.ntuples), 414
- rotation_matrix() (odl.tomo.geometry.conebeam.CircularConeFlatGeometry method), 433
- rotation_matrix() (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 438
- rotation_matrix() (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 452
- rotation_matrix() (odl.tomo.geometry.geometry.AxisOrientedGeometry method), 454
- rotation_matrix() (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 457
- rotation_matrix() (odl.tomo.geometry.geometry.Geometry method), 461
- rotation_matrix() (odl.tomo.geometry.parallel.Parallel2dGeometry method), 464
- rotation_matrix() (odl.tomo.geometry.parallel.Parallel3dAxisGeometry method), 468
- rotation_matrix() (odl.tomo.geometry.parallel.Parallel3dGeometry method), 471
- rotation_matrix() (odl.tomo.geometry.parallel.ParallelGeometry method), 475
- run_tests() (odl.diagnostics.operator.OperatorTest method), 44
- run_tests() (odl.diagnostics.space.SpaceTest method), 47
- ## S
- samples() (in module odl.diagnostics.examples), 42
- scalar_examples() (in module odl.diagnostics.examples), 42
- ScalingOperator (class in odl.operator.default_ops), 170
- schemes (odl.dscr.dscr_mappings.PerAxisInterpolation attribute), 60
- self_adjoint() (odl.diagnostics.operator.OperatorTest method), 44
- Set (class in odl.set.sets), 254
- set (odl.dscr.partition.RectPartition attribute), 147
- set_zero() (odl.dscr.discretization.DiscretizationVector method), 93
- set_zero() (odl.dscr.lp_dscr.DiscreteLpVector method), 138
- set_zero() (odl.set.pspace.ProductSpaceVector method), 246
- set_zero() (odl.set.space.LinearSpaceVector method), 266
- set_zero() (odl.space.base_ntuples.FnBaseVector method), 302
- set_zero() (odl.space.cu_ntuples.CudaFnVector method), 335
- set_zero() (odl.space.cu_ntuples.CudaNtuplesVector method), 348
- set_zero() (odl.space.fspace.FunctionSpaceVector method), 366
- set_zero() (odl.space.ntuples.FnVector method), 398
- sets (odl.set.sets.CartesianProduct attribute), 247
- shape (odl.dscr.discretization.Discretization attribute), 83
- shape (odl.dscr.discretization.DiscretizationVector attribute), 90
- shape (odl.dscr.discretization.RawDiscretization attribute), 95
- shape (odl.dscr.discretization.RawDiscretizationVector attribute), 99

- shape (odl.dscr.grid.RegularGrid attribute), 104
- shape (odl.dscr.grid.TensorGrid attribute), 114
- shape (odl.dscr.lp_dscr.DiscreteLp attribute), 126
- shape (odl.dscr.lp_dscr.DiscreteLpVector attribute), 133
- shape (odl.dscr.partition.RectPartition attribute), 147
- shape (odl.space.base_ntuples.FnBase attribute), 293
- shape (odl.space.base_ntuples.FnBaseVector attribute), 299
- shape (odl.space.base_ntuples.NtuplesBase attribute), 305
- shape (odl.space.base_ntuples.NtuplesBaseVector attribute), 308
- shape (odl.space.cu_ntuples.CudaFn attribute), 311
- shape (odl.space.cu_ntuples.CudaFnVector attribute), 331
- shape (odl.space.cu_ntuples.CudaNtuples attribute), 339
- shape (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
- shape (odl.space.ntuples.Fn attribute), 368
- shape (odl.space.ntuples.FnVector attribute), 393
- shape (odl.space.ntuples.Ntuples attribute), 404
- shape (odl.space.ntuples.NtuplesVector attribute), 409
- shape (odl.tomo.geometry.detector.CircleSectionDetector attribute), 440
- shape (odl.tomo.geometry.detector.Detector attribute), 442
- shape (odl.tomo.geometry.detector.Flat1dDetector attribute), 444
- shape (odl.tomo.geometry.detector.Flat2dDetector attribute), 446
- shape (odl.tomo.geometry.detector.FlatDetector attribute), 448
- shepp_logan() (in module odl.util.phantom), 527
- shifts (odl.trafos.fourier.FourierTransform attribute), 494
- shifts (odl.trafos.fourier.FourierTransformInverse attribute), 499
- show() (odl.dscr.discretization.DiscretizationVector method), 94
- show() (odl.dscr.discretization.RawDiscretizationVector method), 101
- show() (odl.dscr.lp_dscr.DiscreteLpVector method), 138
- show() (odl.set.pspace.ProductSpaceVector method), 246
- show() (odl.space.base_ntuples.FnBaseVector method), 302
- show() (odl.space.base_ntuples.NtuplesBaseVector method), 309
- show() (odl.space.cu_ntuples.CudaFnVector method), 336
- show() (odl.space.cu_ntuples.CudaNtuplesVector method), 348
- show() (odl.space.ntuples.FnVector method), 398
- show() (odl.space.ntuples.NtuplesVector method), 413
- show_discrete_data() (in module odl.util.graphics), 521
- ShowPartial (class in odl.solvers.util.partial), 289
- sign (odl.trafos.fourier.DiscreteFourierTransform attribute), 485
- sign (odl.trafos.fourier.DiscreteFourierTransformInverse attribute), 490
- sign (odl.trafos.fourier.FourierTransform attribute), 494
- sign (odl.trafos.fourier.FourierTransformInverse attribute), 499
- sign() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- sign() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- sign() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- sign() (odl.util.ufuncs.NtuplesUFuncs method), 582
- sign() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- signbit() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- signbit() (odl.util.ufuncs.DiscreteLpUFuncs method), 556
- signbit() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- signbit() (odl.util.ufuncs.NtuplesUFuncs method), 582
- signbit() (odl.util.ufuncs.ProductSpaceUFuncs method), 594
- simple_operator() (in module odl.operator.operator), 202
- sin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- sin() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- sin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- sin() (odl.util.ufuncs.NtuplesUFuncs method), 582
- sin() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- sinh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- sinh() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- sinh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 569
- sinh() (odl.util.ufuncs.NtuplesUFuncs method), 582
- sinh() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- size (odl.dscr.discretization.Discretization attribute), 83
- size (odl.dscr.discretization.DiscretizationVector attribute), 91
- size (odl.dscr.discretization.RawDiscretization attribute), 95
- size (odl.dscr.discretization.RawDiscretizationVector attribute), 99
- size (odl.dscr.grid.RegularGrid attribute), 104
- size (odl.dscr.grid.TensorGrid attribute), 114
- size (odl.dscr.lp_dscr.DiscreteLp attribute), 126
- size (odl.dscr.lp_dscr.DiscreteLpVector attribute), 133
- size (odl.dscr.partition.RectPartition attribute), 147
- size (odl.set.pspace.ProductSpace attribute), 235
- size (odl.set.pspace.ProductSpaceVector attribute), 243
- size (odl.space.base_ntuples.FnBase attribute), 293
- size (odl.space.base_ntuples.FnBaseVector attribute), 299
- size (odl.space.base_ntuples.NtuplesBase attribute), 305
- size (odl.space.base_ntuples.NtuplesBaseVector attribute), 308
- size (odl.space.cu_ntuples.CudaFn attribute), 312
- size (odl.space.cu_ntuples.CudaFnVector attribute), 331
- size (odl.space.cu_ntuples.CudaNtuples attribute), 339

- size (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
- size (odl.space.ntuples.Fn attribute), 368
- size (odl.space.ntuples.FnVector attribute), 393
- size (odl.space.ntuples.Ntuples attribute), 404
- size (odl.space.ntuples.NtuplesVector attribute), 409
- size (odl.tomo.geometry.detector.CircleSectionDetector attribute), 440
- size (odl.tomo.geometry.detector.Detector attribute), 442
- size (odl.tomo.geometry.detector.Flat1dDetector attribute), 444
- size (odl.tomo.geometry.detector.Flat2dDetector attribute), 446
- size (odl.tomo.geometry.detector.FlatDetector attribute), 448
- skip_if_no_benchmark() (in module odl.util.testutils), 531
- skip_if_no_cuda() (in module odl.util.testutils), 531
- skip_if_no_largyscale() (in module odl.util.testutils), 532
- skip_if_no_pyfftw() (in module odl.util.testutils), 532
- skip_if_no_pywavelets() (in module odl.util.testutils), 532
- space (odl.dscr.discretization.DiscretizationVector attribute), 91
- space (odl.dscr.discretization.RawDiscretizationVector attribute), 99
- space (odl.dscr.lp_dscr.DiscreteLpVector attribute), 133
- space (odl.set.pspace.ProductSpaceVector attribute), 243
- space (odl.set.space.LinearSpaceVector attribute), 264
- space (odl.space.base_ntuples.FnBaseVector attribute), 299
- space (odl.space.base_ntuples.NtuplesBaseVector attribute), 308
- space (odl.space.cu_ntuples.CudaFnVector attribute), 331
- space (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
- space (odl.space.fspace.FunctionSetVector attribute), 354
- space (odl.space.fspace.FunctionSpaceVector attribute), 363
- space (odl.space.ntuples.FnVector attribute), 393
- space (odl.space.ntuples.NtuplesVector attribute), 409
- spaces (odl.set.pspace.ProductSpace attribute), 235
- SpaceTest (class in odl.diagnostics.space), 44
- sparse_meshgrid() (in module odl.dscr.grid), 121
- sqrt() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- sqrt() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- sqrt() (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
- sqrt() (odl.util.ufuncs.NtuplesUFuncs method), 582
- sqrt() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- square() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- square() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- square() (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
- square() (odl.util.ufuncs.NtuplesUFuncs method), 582
- square() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- square() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- squeeze() (odl.dscr.grid.RegularGrid method), 111
- squeeze() (odl.dscr.grid.TensorGrid method), 120
- squeeze() (odl.set.domain.IntervalProd method), 233
- src_position() (odl.tomo.geometry.conebeam.CircularConeFlatGeometry method), 433
- src_position() (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry method), 438
- src_position() (odl.tomo.geometry.fanbeam.FanFlatGeometry method), 453
- src_position() (odl.tomo.geometry.geometry.DivergentBeamGeometry method), 457
- src_radius (odl.tomo.geometry.conebeam.CircularConeFlatGeometry attribute), 432
- src_radius (odl.tomo.geometry.conebeam.HelicalConeFlatGeometry attribute), 436
- src_radius (odl.tomo.geometry.fanbeam.FanFlatGeometry attribute), 451
- start() (odl.util.testutils.ProgressBar method), 530
- steepest_descent() (in module odl.solvers.scalar.gradient), 282
- StepLength (class in odl.solvers.scalar.steplen), 286
- stir_projector_from_file() (in module odl.tomo.backends.stir_bindings), 429
- StirVerbosity (class in odl.tomo.backends.stir_bindings), 428
- StorePartial (class in odl.solvers.util.partial), 290
- stride (odl.dscr.grid.RegularGrid attribute), 104
- Strings (class in odl.set.sets), 256
- submarine_phantom() (in module odl.util.phantom), 528
- subtract() (odl.util.ufuncs.CudaNtuplesUFuncs method), 544
- subtract() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- subtract() (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
- subtract() (odl.util.ufuncs.NtuplesUFuncs method), 582
- subtract() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- sum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
- sum() (odl.util.ufuncs.DiscreteLpUFuncs method), 557
- sum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
- sum() (odl.util.ufuncs.NtuplesUFuncs method), 583
- sum() (odl.util.ufuncs.ProductSpaceUFuncs method), 595
- surface() (odl.tomo.geometry.detector.CircleSectionDetector method), 440
- surface() (odl.tomo.geometry.detector.Detector method), 442
- surface() (odl.tomo.geometry.detector.Flat1dDetector method), 444
- surface() (odl.tomo.geometry.detector.Flat2dDetector method), 446

- [surface\(\)](#) (odl.tomo.geometry.detector.FlatDetector method), 448
[surface_deriv\(\)](#) (odl.tomo.geometry.detector.CircleSectionDetector method), 441
[surface_deriv\(\)](#) (odl.tomo.geometry.detector.Detector method), 442
[surface_deriv\(\)](#) (odl.tomo.geometry.detector.Flat1dDetector method), 445
[surface_deriv\(\)](#) (odl.tomo.geometry.detector.Flat2dDetector method), 447
[surface_deriv\(\)](#) (odl.tomo.geometry.detector.FlatDetector method), 448
[surface_measure\(\)](#) (odl.tomo.geometry.detector.CircleSectionDetector method), 441
[surface_measure\(\)](#) (odl.tomo.geometry.detector.Detector method), 443
[surface_measure\(\)](#) (odl.tomo.geometry.detector.Flat1dDetector method), 445
[surface_measure\(\)](#) (odl.tomo.geometry.detector.Flat2dDetector method), 447
[surface_measure\(\)](#) (odl.tomo.geometry.detector.FlatDetector method), 448
- T**
- [T](#) (odl.discr.discretization.DiscretizationVector attribute), 89
[T](#) (odl.discr.lp_discr.DiscreteLpVector attribute), 131
[T](#) (odl.operator.default_ops.InnerProductOperator attribute), 158
[T](#) (odl.set.pspace.ProductSpaceVector attribute), 242
[T](#) (odl.set.space.LinearSpaceVector attribute), 263
[T](#) (odl.space.base_ntuples.FnBaseVector attribute), 298
[T](#) (odl.space.cu_ntuples.CudaFnVector attribute), 330
[T](#) (odl.space.cu_ntuples.CudaNtuplesVector attribute), 342
[T](#) (odl.space.fspace.FunctionSpaceVector attribute), 362
[T](#) (odl.space.ntuples.FnVector attribute), 391
[tan\(\)](#) (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
[tan\(\)](#) (odl.util.ufuncs.DiscreteLpUFuncs method), 557
[tan\(\)](#) (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
[tan\(\)](#) (odl.util.ufuncs.NtuplesUFuncs method), 583
[tan\(\)](#) (odl.util.ufuncs.ProductSpaceUFuncs method), 595
[tanh\(\)](#) (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
[tanh\(\)](#) (odl.util.ufuncs.DiscreteLpUFuncs method), 558
[tanh\(\)](#) (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
[tanh\(\)](#) (odl.util.ufuncs.NtuplesUFuncs method), 583
[tanh\(\)](#) (odl.util.ufuncs.ProductSpaceUFuncs method), 596
[TensorGrid](#) (class in odl.discr.grid), 112
[timeit\(\)](#) (in module odl.util.testutils), 532
[Timer](#) (class in odl.util.testutils), 530
[to_lab_sys\(\)](#) (in module odl.tomo.util.utility), 483
[to_local_sys\(\)](#) (in module odl.tomo.util.utility), 483
- [true_divide\(\)](#) (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
[true_divide\(\)](#) (odl.util.ufuncs.DiscreteLpUFuncs method), 558
[true_divide\(\)](#) (odl.util.ufuncs.NtuplesBaseUFuncs method), 570
[true_divide\(\)](#) (odl.util.ufuncs.NtuplesUFuncs method), 583
[true_divide\(\)](#) (odl.util.ufuncs.ProductSpaceUFuncs method), 596
[true_ndim](#) (odl.set.domain.IntervalProd attribute), 226
[trunc\(\)](#) (odl.util.ufuncs.CudaNtuplesUFuncs method), 545
[trunc\(\)](#) (odl.util.ufuncs.DiscreteLpUFuncs method), 558
[trunc\(\)](#) (odl.util.ufuncs.NtuplesBaseUFuncs method), 571
[trunc\(\)](#) (odl.util.ufuncs.NtuplesUFuncs method), 583
[trunc\(\)](#) (odl.util.ufuncs.ProductSpaceUFuncs method), 596
- [ufunc](#) (odl.discr.discretization.DiscretizationVector attribute), 91
[ufunc](#) (odl.discr.discretization.RawDiscretizationVector attribute), 99
[ufunc](#) (odl.discr.lp_discr.DiscreteLpVector attribute), 134
[ufunc](#) (odl.set.pspace.ProductSpaceVector attribute), 243
[ufunc](#) (odl.space.base_ntuples.FnBaseVector attribute), 299
[ufunc](#) (odl.space.base_ntuples.NtuplesBaseVector attribute), 308
[ufunc](#) (odl.space.cu_ntuples.CudaFnVector attribute), 331
[ufunc](#) (odl.space.cu_ntuples.CudaNtuplesVector attribute), 343
[ufunc](#) (odl.space.ntuples.FnVector attribute), 393
[ufunc](#) (odl.space.ntuples.NtuplesVector attribute), 409
[uniform_discr\(\)](#) (in module odl.discr.lp_discr), 140
[uniform_discr_frompartition\(\)](#) (in module odl.discr.lp_discr), 141
[uniform_discr_fromspace\(\)](#) (in module odl.discr.lp_discr), 142
[uniform_partition\(\)](#) (in module odl.discr.partition), 148
[uniform_partition_fromgrid\(\)](#) (in module odl.discr.partition), 149
[uniform_partition_fromintv\(\)](#) (in module odl.discr.partition), 151
[uniform_sampling\(\)](#) (in module odl.discr.grid), 122
[uniform_sampling_fromintv\(\)](#) (in module odl.discr.grid), 122
[UniversalSet](#) (class in odl.set.sets), 257
[UniversalSpace](#) (class in odl.set.space), 266
[update\(\)](#) (odl.util.testutils.ProgressBar method), 530
[uspace](#) (odl.discr.discretization.Discretization attribute), 84

uspace (odl.dscr.discretization.RawDiscretization attribute), 96
uspace (odl.dscr.lp_dscr.DiscreteLp attribute), 126

V

vector (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 337
vector (odl.space.ntuples.FnVectorWeighting attribute), 399
vector() (in module odl.space.space_utils), 416
vector() (odl.diagnostics.space.SpaceTest method), 47
vector_assign() (odl.diagnostics.space.SpaceTest method), 47
vector_copy() (odl.diagnostics.space.SpaceTest method), 47
vector_equals() (odl.diagnostics.space.SpaceTest method), 47
vector_examples() (in module odl.diagnostics.examples), 42
vector_is_valid() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 338
vector_is_valid() (odl.space.ntuples.FnVectorWeighting method), 401
vector_set_zero() (odl.diagnostics.space.SpaceTest method), 47
vector_space() (odl.diagnostics.space.SpaceTest method), 47
vectorization, 34
vectorize (class in odl.util.vectorization), 602
volume (odl.set.domain.IntervalProd attribute), 226

W

wavelet_decomposition3d() (in module odl.trafos.wavelet), 519
wavelet_reconstruction3d() (in module odl.trafos.wavelet), 520
WaveletTransform (class in odl.trafos.wavelet), 509
WaveletTransformInverse (class in odl.trafos.wavelet), 513
weighted_dist() (in module odl.space.ntuples), 415
weighted_inner() (in module odl.space.ntuples), 415
weighted_norm() (in module odl.space.ntuples), 415
weighting (odl.dscr.discretization.Discretization attribute), 84
weighting (odl.dscr.lp_dscr.DiscreteLp attribute), 126
weighting (odl.space.cu_ntuples.CudaFn attribute), 312
weighting (odl.space.ntuples.Fn attribute), 368
weights (odl.set.pspace.ProductSpace attribute), 236
with_metaclass() (in module odl.util.utility), 601
wrap_reduction_base() (in module odl.util.ufuncs), 596
wrap_reduction_discretelp() (in module odl.util.ufuncs), 596
wrap_reduction_productspace() (in module odl.util.ufuncs), 597

wrap_ufunc_base() (in module odl.util.ufuncs), 597
wrap_ufunc_discretelp() (in module odl.util.ufuncs), 597
wrap_ufunc_ntuples() (in module odl.util.ufuncs), 597
wrap_ufunc_productspace() (in module odl.util.ufuncs), 597

Z

zero() (odl.dscr.discretization.Discretization method), 88
zero() (odl.dscr.lp_dscr.DiscreteLp method), 130
zero() (odl.set.pspace.ProductSpace method), 241
zero() (odl.set.space.LinearSpace method), 262
zero() (odl.set.space.UniversalSpace method), 270
zero() (odl.space.base_ntuples.FnBase method), 298
zero() (odl.space.cu_ntuples.CudaFn method), 319
zero() (odl.space.fspace.FunctionSpace method), 361
zero() (odl.space.ntuples.Fn method), 376
zero() (odl.space.ntuples.Ntuples method), 407
ZeroOperator (class in odl.operator.default_ops), 173