Menpo Documentation

Release 0.7.0

Joan Alabort-i-Medina, Epameinondas Antonakos, James Booth,

May 20, 2016

Contents

Welcome to the Menpo documentation!

Menpo is a Python package designed to make manipulating annotated data more simple. In particular, sparse locations on either images or meshes, referred to as **landmarks** within Menpo, are tightly coupled with their reference objects. For areas such as Computer Vision that involve learning models based on prior knowledge of object location (such as object detection and landmark localisation), Menpo is a very powerful toolkit.

A short example is often more illustrative than a verbose explanation. Let's assume that you want to load a set of images that have been annotated with bounding boxes, and that these bounding box locations live in text files next to the images. Here's how we would load the images and extract the areas within the bounding boxes using Menpo:

```
import menpo.io as mio

images = []
for image in mio.import_images('./images_folder'):
    images.append(image.crop_to_landmarks())
```

Where `import_images` returns a `LazyList` to keep memory usage low.

Although the above is a very simple example, we believe that being able to easily manipulate and couple landmarks with images *and* meshes, is an important problem for building powerful models in areas such as facial point localisation.

To get started, check out the User Guide for instructions on installation and some of the core concepts within Menpo.

CHAPTER 1

User Guide

The User Guide is designed to give you an overview of the key concepts within Menpo. In particular, we want to try and explain some of the design decisions that we made and demonstrate why we think they are powerful concepts for exploring visual data.

1.1 Quick Start

Here we give a very quick rundown of the basic links and information sources for the project.

1.1.1 Basic Installation

Menpo should be installable via pip on all major platforms:

```
$ pip install menpo
```

However, in the menpo team, we **strongly** advocate the usage of conda for scientific Python, as it makes installation of compiled binaries much more simple. In particular, if you wish to use any of the related Menpo projects such as *menpofit*, *menpo3d* or *menpodetect*, you will not be able to easily do so without using conda.

```
$ conda install -c menpo menpo
```

To install using conda, please see the thorough instructions for each platform on the Menpo website.

1.1.2 API Documentation

Visit API Documentation

Menpo is extensively documented on a per-method/class level and much of this documentation is reflected in the API Documentation. If any functions or classes are missing, please bring it to the attention of the developers on Github.

1.1.3 Notebooks

Explore the Menpo Notebooks

For a more thorough set of examples, we provide a set of Jupyter notebooks that demonstrate common use cases of Menpo. This concentrates on an overview of the functionality of the major classes and ideas behind Menpo.

1.1.4 User Group and Issues

If you wish to get in contact with the Menpo developers, you can do so via various channels. If you have found a bug, or if any part of Menpo behaves in a way you do not expect, please raise an issue on Github.

If you want to ask a theoretical question, or are having problems installing or setting up Menpo, please visit the user group.

1.2 Introduction

This user guide is a general introduction to Menpo, aiming to provide a bird's eye of Menpo's design. After reading this guide you should be able to go explore Menpo's extensive Notebooks and not be too suprised by what you see.

1.2.1 Core Interfaces

Menpo is an object oriented framework built around a set of core abstract interfaces, each one governing a single facet of Menpo's design. Menpo's key interfaces are:

- `Shape` - spatial data containers
- *`Vectorizable`* - efficient bi-directional conversion of types to a vector representation
- *`Targetable`* - objects that generate some spatial data
- *`Transform`* - flexible spatial transformations
- *`Landmarkable`* - objects that can be annotated with spatial labelled landmarks

1.2.2 Data containers

Most numerical data in Menpo is passed around in one of our core data containers. The features of each of the data containers is explained in great detail in the notebooks - here we just list them to give you a feel for what to expect:

- *`LazyList`* - a list that calls a function when indexed
- *`Image`* - n-dimensional image with k-channels of data
- *`MaskedImage`* - As *`Image`*, but with a boolean mask
- *`BooleanImage`* - As boolean image that is used for masking images.
- *`PointCloud`* - n-dimensional ordered point collection
- *`PointUndirectedGraph`* - n-dimensional ordered point collection with undirected connectivity
- *`PointDirectedGraph`* - n-dimensional ordered point collection with directed connectivity
- *`TriMesh`* - As *`PointCloud`*, but with a triangulation

1.3 Menpo's Data Types

Menpo is a high level software package. It is not a replacement for scikit-image, scikit-learn, or opencv - it ties all these types of packages together in to a unified framework for building and fitting deformable models. As a result, most of our algorithms take as input a higher level representation of data than simple numpy arrays.

1.3.1 Why have data types - what's wrong with numpy arrays?

Menpo's data types are thin wrappers around numpy arrays. They give semantic meaning to the underlying array through providing clearly named and consistent properties. As an example let's take a look at `PointCloud`, Menpo's workhorse for spatial data. Construction requires a numpy array:

```
x = np.random.rand(3, 2)
pc = PointCloud(x)
```

It's natural to ask the question:

> *Is this a collection of three 2D points, or two 3D points?*

In Menpo, you never do this - just look at the properties on the pointcloud:

```
pc.n_points  # 3
pc.n_dims    # 2
```

If we take a look at the properties we can see they are trivial:

```
@property
def n_points(self):
    return self.points.shape[0]

@property
def n_dims(self):
    return self.points.shape[1]
```

Using these properties makes code much more readable in algorithms accepting Menpo's types. Let's imagine a routine that does some operation on an image and a related point cloud. If it accepted numpy arrays, we might see something like this on the top line:

```
def foo_arrays(x, img):
    # preallocate the result
    y = np.zeros(x.shape[1],
                 x.shape[2],
                 img.shape[-1])
    ...
```

On first glance it is not at all apparent what `y`'s shape is semantically. Now let's take a look at the equivalent code using Menpo's types:

```
def foo_menpo(pc, img):
    # preallocate the result
    y = np.zeros(pc.n_dims,
                 pc.n_points,
                 img.n_channels)
    ...
```

This time it's immediately apparent what `y`'s shape is. Although this is a somewhat contrived example, you will find this pattern applied consistently across Menpo, and it aids greatly in keeping the code readable.

1.3.2 Key points

1. **Containers store the underlying numpy array in an easy to access attribute.** For the `PointCloud` family see the `.points` attribute. On `Image` and subclasses, the actual data array is stored at `.pixels`.

2. **Importing assets though** *menpo.io* **will result in our data containers, not numpy arrays**. This means in a lot of situations you never need to remember the Menpo conventions for ordering of array data - just ask for an image and you will get an `Image` object.

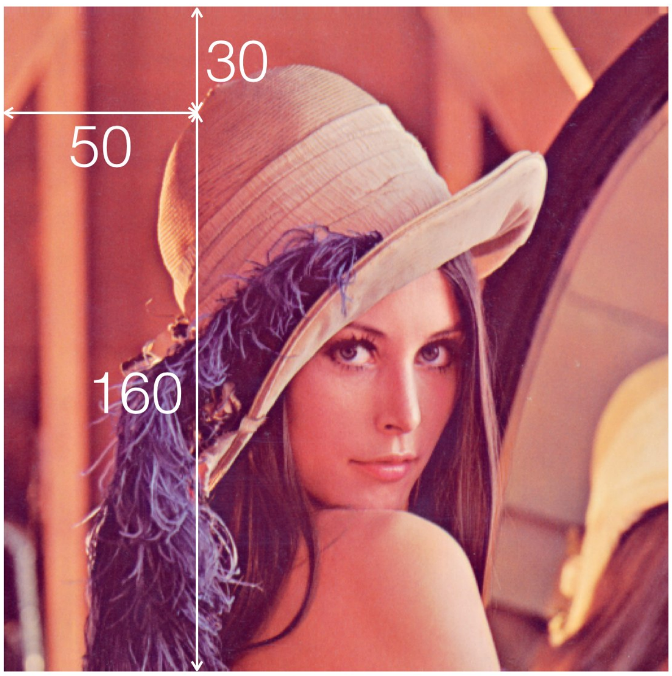
3. **All containers copy data by default**. Look for the `copy=False` keyword argument if you want to avoid copying a large numpy array for performance.

4. **Containers perform sanity checks**. This helps catch obvious bugs like misshaping an array. You can sometimes suppress them for extra performance with the `skip_checks=True` keyword argument.

1.4 Working with Images and PointClouds

Menpo takes an opinionated stance on certain issues - one of which is establishing sensible rules for how to work with spatial data and image data in the same framework.

Let's start with a quiz - which of the following is correct?



?	x	y
a	30	50
b	50	30
c	50	160
d	160	50

Most would answer **b** - images are indexed from the top left, with `x` going across and `y` going down.

Now another question - how do I access that pixel in the pixels array?

```
a: lenna[30, 50]
b: lenna[50, 30]
```

The correct answer is **b** - pixels get stored in a *y, x* order so we have to flip the points to access the array.

As Menpo blends together use of PointClouds and Images frequently this can cause a lot of confusion. You might create a *Translation* of `5` in the `y` direction as the following:

```
t = menpo.transform.Translation([0, 5])
```

And then expect to use it to warp an image:

```
img.warp_to(reference_shape, t)
```

and then some spatial data related to the image:

```
t.apply(some_data)
```

Unfortunately the meaning of `y` in these two domains is different - some code would have to flip the order of applying the translation of the transform to an image, a potential cause of confusion.

The *worst* part about this is that once we go to voxel data (which *Image* largely supports, and will fully support in the future), a z-axis is added.

There is one important caveat, unfortunately. The **first axis of an image represents the channels**. Unlike in other software, such as Matlab, which follows the fortran convention of being column major, Python and other C-like languages generally conform to a row major order. Practically this means that if you want to iterate over each channel of an image, you need the memory layout to reflect this. This means you want the pixel data of each channel to be contiguous in memory. **For row major memory, this implies that the first axis should represent an iteration over the channel data.**

Now, as was mentioned, we want to drop all the swapping business. Therefore, forgiving that the **first axis indexes the channel data**, the following axes always match the spatial data. For example, The zeroth axis of the spatial data once more corresponds with the first axis (the first axis is *after the zeroth axis representing the channel data*) of the image data. Trying to keep track of these rules muddies an otherwise very simple concept.

1.4.1 Menpo's approach

Menpo's solution to this problem is simple - **drop the insistence of calling axes x, y, and z**. Skipping the channel data, which represents the zeroth axis, the first axis of the pixel data is simply that - the first axis. It corresponds exactly with the zeroth axis on the point cloud. If you have an image with annotations provided the zeroth axis of the *PointCloud* representing the annotations will correspond with the first axis of the image. This rule makes working with images and spatial data simple - short you should never have to think about flipping axes in Menpo.

It's natural to be concerned at this point that establishing such rules must make it really difficult ingest data which follows different conventions. This is incorrect - one of the biggest strengths of the *menpo.io* module is that each asset importer normalizes the format of the data to format Menpo's rules.

1.4.2 Key Points

- **Menpo is n-dimensional**. We try and avoid speaking of `x` and `y`, because there are many different conventions in use.
- **The IO module ensures that different data formats are normalized** upon loading into Menpo. For example, *Image* types are imported as 64-bit floating point numbers normalised between [0, 1], by default.
- **axis 0 of landmarks corresponds to axis 0 of the container it is an annotation of**.
- **The first axis of image types is always the channel data. The remaining axes map exactly to the other spatial axes.** Therefore, the first image axis maps exactly to the zeroth axis of a PointCloud.

1.5 Vectorizing Objects

Computer Vision algorithms are frequently formulated as linear algebra problems in a high dimensional space, where each asset is stripped into a vector. In this high dimensional space we may perform any number of operations, but normally we can't stay in this space for the whole algorithm - we normally have to recast the vector back into it's original domain in order to perform other operations.

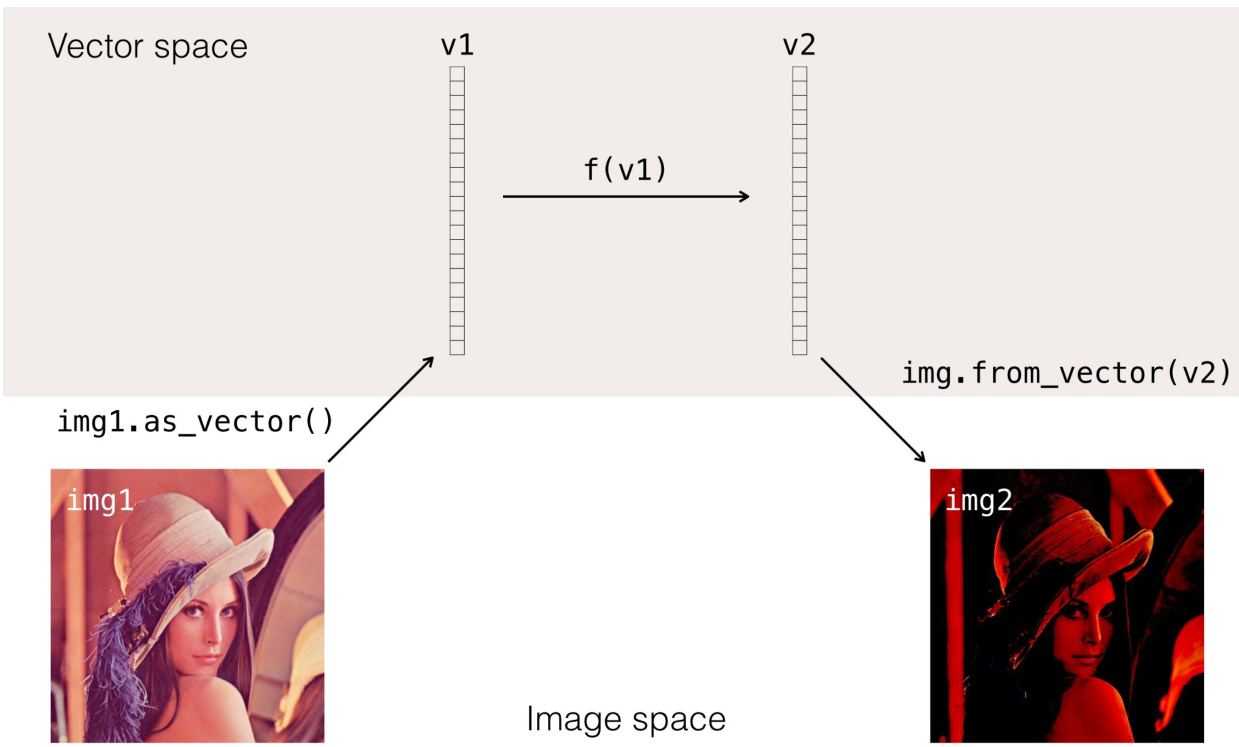


Fig. 1.1: **Figure 1:** *Vectorizing allows Menpo to have rich data types whilst simultaneously providing efficient linear algebra routines. Here an image is vectorized, and an arbitrary process* `f()` *is performed on it's vector representation. Afterwards the vector is converted the back into an image. The vector operation is completely general, and could have equally been performed on some spatial data.*

An example of this might be seen with images, where the gradient of the intensity values of an image needs to be taken. This is a complex problem to solve in a vector space representation of the image, but trivial to solve in the image domain.

Menpo bridges the gap by naively supporting bi-directional vectorisation of it's types through the `Vectorizable` interface. Through this, any type can be safely and efficiently converted to a vector form and back again. You'll find the key methods of `Vectorizable` are extensively used in Menpo. They are

- `as_vector` - generate a vector from one of our types.
- `from_vector` - rebuild one of our types from a vector
- `from_vector_inplace` - alter an object inplace to take on the new state

1.5.1 Key points

1. **Each type defines it's own form of vectorization**. Calling `as_vector` on a `Image` returns all of the pixels in a single strip, whilst on a `MaskedImage` only the true pixels are returned. This distinction means that much of Menpo's image algorithms work equally well with masked or unmasked data - it's the `Vectorizable` interface that abstracts away the difference between the two.

2. **Lots of things are vectorizable, not just images.** Pointclouds and lots of transforms are too.

3. **The length of the resulting vector of a type can be found by querying the ‘‘n_parameters‘‘ property.**

4. **The vectorized form of an object does not have to be ‘complete’.** `from_vector` and `from_vector_inplace` can use the object they are called on to rebuild a complete state. Think of vectorization more as a `parametrization` of the object, not a complete serialization.

1.6 Visualizing Objects

In Menpo, we take an opinionated stance that data exploration is a key part of working with visual data. Therefore, we tried to make the mental overhead of visualizing objects as low as possible. Therefore, we made visualization a key concept directly on our data containers, rather than requiring extra imports in order to view your data.

We also took a strong step towards simple visualization of data collections by integrating some of our core types such as `Image` with visualization widgets for the Jupyter notebook.

1.6.1 Visualizing 2D Images

Without further ado, a quick example of viewing a 2D image:

```
%matplotlib inline  # This is only needed if viewing in an IPython notebook
import menpo.io as mio

bb = mio.import_builtin_asset.breakingbad_jpg()
bb.view()
```

Viewing the image landmarks:

```
%matplotlib inline  # This is only needed if viewing in an IPython notebook
import menpo.io as mio

bb = mio.import_builtin_asset.breakingbad_jpg()
bb.view_landmarks()
```

Viewing the image with a native IPython widget:

```
%matplotlib inline  # This is only needed if viewing in an IPython notebook
import menpo.io as mio

bb = mio.import_builtin_asset.breakingbad_jpg()
bb.view_widget()
```

1.6.2 Visualizing A List Of 2D Images

Visualizing lists of images is also incredibly simple if you are using the Jupyter notebook and have the MenpoWidgets package installed:

```
%matplotlib inline
import menpo.io as mio
from menpowidgets import visualize_images

# import_images is a generator, so we must exhaust the generator before
# we can visualize the list. This is because the widget allows you to
# jump arbitrarily around the list, which cannot be done with generators.
images = list(mio.import_images('./path/to/images/*.jpg'))
visualize_images(images)
```

1.6.3 Visualizing A 2D PointCloud

Visualizing `PointCloud` objects and subclasses is a very familiar experience:

```
%matplotlib inline
from menpo.shape import PointCloud
import numpy as np

pcloud = PointCloud(np.array([[0, 0], [1, 0], [1, 1], [0, 1]]))
pcloud.view()
```

1.6.4 Visualizing In 3D

Menpo natively supports 3D objects, such as triangulated meshes, as our base classes are n-dimensional. However, as viewing in 3D is a much more complicated experience, we have segregated the 3D viewing package into one of our sub-packages: Menpo3D.

If you try to view a 3D `PointCloud` without having Menpo3D installed, you will receive an exception asking you to install it.

Menpo3D also comes with many other complicated pieces of functionality for 3D meshes such as a rasterizer. We recommend you look at Menpo3D if you want to use Menpo for 3D mesh manipulation.

1.7 Changelog

1.7.1 0.7.0 (2016/05/20)

New release that contains some minor breaking changes. In general, the biggest changes are:

- Use ImageIO rather than Pillow for basic importing of some image types. The most important aspect of this change is that we now support importing videos! Our GIF support also became much more robust. Note that importing videos is still considered to be relatively experimental due to the underlying implementation in imageio not being 100% accurate. Therefore, we warn our users that importing videos for important experiments is not advised.
- Change multi-asset importing to use a new type - the *LazyList*. Lazy lists are a generic concept for a container that holds onto a list of callables which are invoked on indexing. This means that image importing, for example, returns immediately but can be **randomly indexed**. This is in contrast to generators, which have to be sequentially accessed. This is particularly important for video support, as the frames can be accessed randomly or sliced from the end (rather than having to pay the penalty of importing the entirety of a long video just to access the last frame, for example). A simple example of using the *LazyList* to import images is as follows:

```
import menpo.io as mio
images = mio.import_images('/path/to/many/images')  # Returns immediately
image0 = images[0]  # Loading performed at access

# Example of much simpler preprocessing
preprocess_func = lambda x: x.as_greyscale()
greyscale_images = images.map(preprocess_func)  # Returns immediately
grey_image0 = greyscale_images[0]  # Loading and as_greyscale() performed at access

# Visualizing randomly is now much simpler too!
% matplotlib inline
from menpowidgets import visualize_images
visualize_images(greyscale_images)  # Can now randomly access list
```

- Move one step closer to ensuring that all image operatons are copies rather than inplace. This means breaking some methods as there was no 'non' inplace method (the break was to change them to return a copy). Likely the most common anti-pattern was code such as:

```
import menpo.io as mio
image = mio.import_builtin_asset.takeo_ppm().as_masked()
image.constrain_landmarks_to_bounds()
```

  Which now requires assigning the call to `constrain_landmarks_to_bounds` to a variable, as a copy is returned:

```
import menpo.io as mio
image = mio.import_builtin_asset.takeo_ppm().as_masked()
image = image.constrain_landmarks_to_bounds()
```

Note that this release also officially supports **Python 3.5**!

Breaking Changes

- ImageIO is used for importing. Therefore, the pixel values of some images have changed due to the difference in underlying importing code.
- Multi-asset importers are now of type `LazyList`.
- HOG previously returned negative values due to rounding errors on binning. This has been rectified, so the output values of HOG are now slightly different.
- `set_boundary_pixels` is no longer in place.
- `normalize_inplace` has been deprecated and removed. `normalize` is now a feature that abstracts out the normalisation logic.

- gaussian_pyramid and pyramid always return copies (before the first image was the original image, not copied).
- constrain_to_landmarks/constrain_to_pointcloud/constrain_mask_to_landmarks are no longer in place.
- set_patches is no longer in place.
- has_landmarks_outside_bounds is now a method.

New Features

- from_tri_mask method added to TriMesh
- LazyList type that holds a list of callables that are invoked on indexing.
- New rasterize methods. Given an image and a landmark group, return a new image with the landmarks rasterized onto the image. Useful for saving results to disk.
- Python 3.5 support!
- Better support for non float64 image types. For example, as_greyscale can be called on a uint8 image.
- New method rasterize_landmarks that allows easy image rasterization. By default, MaskedImages are masked with a black background. Use as_unmasked to change the colour/not returned masked image.
- Add bounds method to images. This is defined as ((0, 0), (height - 1, width - 1)) - the set of indices that are indexable into the image for sampling.
- Add constrain_to_bounds to PointCloud. Snaps the pointcloud exactly to the bounds given.
- init_from_pointcloud method add to Image. Allows the creation of an image that completely bounds a given pointcloud. This is useful for both viewing images of pointclouds and for creating 'reference frames' for algorithms like Active Appearance Models.
- init_from_depth_image method on PointCloud and subclasses. Allows the creation of a mesh from an image that contains pixel values that represent depth/height values. Very useful for visualising RGB-D data.
- pickle_paths method.
- Overwriting images now throws OverwriteError rather than just ValueError (OverwriteError is a subclass of ValueError) so this is not a breaking change.

Deprecated

- The previously deprecated inplace image methods **were not removed in this release**.
- set_h_matrix is deprecated for Homogeneous transforms.
- set_masked_pixels is deprecated in favor of from_vector.
- Deprecate constrain_landmarks_to_bounds on images.

Github Pull Requests

- #698 Video importing warnings. (@patricksnape)
- #697 Relex version constraints on dependencies. (@jabooth)
- #695 condaci fixes. (@patricksnape)
- #692 new OverwriteError raised specifically for overwrite errors in io.export. (@jabooth)

- #691 Add mio.pickle_paths(glob). (@jabooth)
- #690 Fix init_2d_grid for TriMesh subclasses + add init_from_depth_image. (@patricksnape)
- #687 WIP: BREAKING: Various release fixes. (@patricksnape)
- #685 GMRF mahalanobis computation with sparse precision. (@nontas)
- #684 Video importer docs and negative max_images. (@grigorisg9gr)
- #683 Bugfix: Widget imports. (@nontas)
- #682 Update the view_patches to show only the selected landmarks. (@grigorisg9gr)
- #680 Expose file extension to exporters (Fix PIL exporter bug). (@patricksnape)
- #678 Deprecate set_h_matrix and fix #677. (@patricksnape)
- #676 Implement LazyList __add__. (@patricksnape)
- #673 Fix the widgets in PCA. (@grigorisg9gr)
- #672 Use Conda environment.yml on RTD. (@patricksnape)
- #670 Rasterize 2D Landmarks Method. (@patricksnape)
- #669 BREAKING: Add LazyList - default importing is now Lazy. (@patricksnape)
- #668 Speedup as_greyscale. (@patricksnape)
- #666 Add the protocol option in exporting pickle. (@grigorisg9gr)
- #665 Fix bug with patches of different type than float64. (@patricksnape)
- #664 Python 3.5 builds. (@patricksnape)
- #661 Return labels - which maps to a KeysView as a list. (@patricksnape)
- #648 Turn coverage checking back on. (@patricksnape)
- #644 Remove label kwarg. (@patricksnape)
- #639 add from_tri_mask method to TriMesh instances. (@jabooth)
- #633 BREAKING: Imageio. (@patricksnape)
- #606 Fix negative values in HOG calculation. (@patricksnape)

1.7.2 0.6.2 (2015/12/13)

Add axes ticks option to `view_patches`.

Github Pull Requests

- #659 Add axes ticks options to view_patches (@nontas)

1.7.3 0.6.1 (2015/12/09)

Fix a nasty bug pertaining to a Diamond inheritance problem in PCA. Add the Gaussion Markov Random Field (GRMF) model. Also a couple of other bugfixes for visualization.

Github Pull Requests

- #658 PCA Diamond problem fix (@patricksnape)
- #655 Bugfix and improvements in visualize package (@nontas)
- #656 print_dynamic bugfix (@nontas)
- #635 Gaussian Markov Random Field (@nontas, @patricksnape)

1.7.4 0.6.0 (2015/11/26)

This release is another set of breaking changes for Menpo. All `in_place` methods have been deprecated to make the API clearer (always copy). The largest change is the removal of all widgets into a subpackage called menpowidgets. To continue using widgets within the Jupyter notebook, you should install menpowidgets.

Breaking Changes

- Procrustes analysis now checks for mirroring and disables it by default. This is a change in behaviour.
- The `sample_offsets` argument of *menpo.image.Image.extract_patches()* now expects a numpy array rather than a *PointCloud*.
- All widgets are removed and now exist as part of the menpowidgets project. The widgets are now only compatible with Jupyter 4.0 and above.
- Landmark labellers have been totally refactored and renamed. They have not been deprecated due to the changes. However, the new changes mean that the naming scheme of labels is now much more intuitive. Practically, the usage of labelling has only changed in that now it is possible to label not only *LandmarkGroup* but also *PointCloud* and numpy arrays directly.
- Landmarks are now warped by default, where previously they were not.
- All vlfeat features have now become optional and will not appear if cyvlfeat is not installed.
- All `label` keyword arguments have been removed. They were not found to be useful. For the same effect, you can always create a new landmark group that only contains that label and use that as the `group` key.

New Features

- New SIFT type features that return vectors rather than dense features. (*menpo.feature.vector_128_dsift()*, *menpo.feature.hellinger_vector_128_dsift()*)
- *menpo.shape.PointCloud.init_2d_grid()* static constructor for *PointCloud* and subclasses.
- Add `PCAVectorModel` class that allows performing PCA directly on arrays.
- New static constructors on PCA models for building PCA directly from covariance matrices or components (*menpo.model.PCAVectorModel.init_from_components()* and *menpo.model.PCAVectorModel.init_from_covariance_matrix()*).
- New *menpo.image.Image.mirror()* method on images.
- New *menpo.image.Image.set_patches()* methods on images.
- New *menpo.image.Image.rotate_ccw_about_centre()* method on images.
- When performing operations on images, you can now add the `return_transform` kwarg that will return both the new image **and** the transform that created the image. This can be very useful for processing landmarks after images have been cropped and rescaled for example.

Github Pull Requests

- #652 Deprecate a number of inplace methods (@jabooth)
- #653 New features (vector dsift) (@patricksnape)
- #651 remove deprecations from 0.5.0 (@jabooth)
- #650 PointCloud init_2d_grid (@patricksnape)
- #646 Add ibug_49 -> ibug_49 labelling (@patricksnape)
- #645 Add new PCAVectorModel class, refactor model package (@patricksnape, @nontas)
- #644 Remove label kwarg (@patricksnape)
- #643 Build fixes (@patricksnape)
- #638 bugfix 2D triangle areas sign was ambiguous (@jabooth)
- #634 Fixing @patricksnape and @nontas foolish errors (@yuxiang-zhou)
- #542 Add mirroring check to procrustes (@nontas, @patricksnape)
- #632 Widgets Migration (@patricksnape, @nontas)
- #631 Optional transform return on Image methods (@nontas)
- #628 Patches Visualization (@nontas)
- #629 Image counter-clockwise rotation (@nontas)
- #630 Mirror image (@nontas)
- #625 Labellers Refactoring (@patricksnape)
- #623 Fix widgets for new Jupyter/IPython 4 release (@patricksnape)
- #620 Define patches offsets as ndarray (@nontas)

1.7.5 0.5.3 (2015/08/12)

Tiny point release just fixing a typo in the `unique_edge_indices` method.

1.7.6 0.5.2 (2015/08/04)

Minor bug fixes and impovements including:

- Menpo is now better at preserving dtypes other than np.float through common operations
- Image has a new convenience constructor `init_from_rolled_channels()` to handle building images that have the channels at the back of the array.
- There are also new `crop_to_pointcloud()` and `crop_to_pointcloud_proportion()` methods to round out the Image API, and a deprecation of `rescale_to_reference_shape()` in favour of `rescale_to_pointcloud()` to make things more consistent.
- The `gradient()` method is deprecated (use `menpo.feature.gradient` instead)
- Propagation of the `.path` property when using `as_masked()` was fixed
- Fix for exporting 3D LJSON landmark files
- A new `shuffle` kwarg (default `False`) is present on all multi importers.

Github Pull Requests

- #617 add shuffle kwarg to multi import generators (@jabooth)
- #619 Ensure that LJSON landmarks are read in as floats (@jabooth)
- #618 Small image fix (@patricksnape)
- #613 Balance out rescale/crop methods (@patricksnape)
- #615 Allow exporting of 3D landmarks. (@mmcauliffe)
- #612 Type maintain (@patricksnape)
- #602 Extract patches types (@patricksnape)
- #608 Slider for selecting landmark group on widgets (@nontas)
- #605 tmp move to master condaci (@jabooth)

1.7.7 0.5.1 (2015/07/16)

A small point release that improves the Cython code (particularly extracting patches) compatibility with different data types. In particular, more floating point data types are now supported. `print_progress` was added and widgets were fixed after the Jupyter 4.0 release. Also, upgrade cyvlfeat requirement to 0.4.0.

Github Pull Requests

- #604 print_progress enhancements (@jabooth)
- #603 Fixes for new cyvlfeat (@patricksnape)
- #599 Add erode and dilate methods to MaskedImage (@jalabort)
- #601 Add sudo: false to turn on container builds (@patricksnape)
- #600 Human3.6M labels (@nontas)

1.7.8 0.5.0 (2015/06/25)

This release of Menpo makes a number of very important **BREAKING** changes to the format of Menpo's core data types. Most importantly is #524 which swaps the position of the channels on an image from the last axis to the first. This is to maintain row-major ordering and make iterating over the pixels of a channel efficient. This made a huge improvement in speed in other packages such as MenpoFit. It also makes common operations such as iterating over the pixels in an image much simpler:

```
for channels in image.pixels:
    print(channels.shape)  # This will be a (height x width) ndarray
```

Other important changes include:

- Updating all widgets to work with IPython 3
- Incremental PCA was added.
- non-inplace cropping methods
- Dense SIFT features provided by vlfeat
- The implementation of graphs was changed to use sparse matrices by default. **This may cause breaking changes.**

- Many other improvements detailed in the pull requests below!

If you have serialized data using Menpo, you will likely find you have trouble reimporting it. If this is the case, please visit the user group for advice.

Github Pull Requests

- #598 Visualize sum of channels in widgets (@nontas, @patricksnape)
- #597 test new dev tag behavior on condaci (@jabooth)
- #591 Scale around centre (@patricksnape)
- #596 Update to versioneer v0.15 (@jabooth, @patricksnape)
- #495 SIFT features (@nontas, @patricksnape, @jabooth, @jalabort)
- #595 Update mean_pointcloud (@patricksnape, @jalabort)
- #541 Add triangulation labels for ibug_face_(66/51/49) (@jalabort)
- #590 Fix centre and diagonal being properties on Images (@patricksnape)
- #592 Refactor out bounding_box method (@patricksnape)
- #566 TriMesh utilities (@jabooth)
- #593 Minor bugfix on AnimationOptionsWidget (@nontas)
- #587 promote non-inplace crop methods, crop performance improvements (@jabooth, @patricksnape)
- #586 fix as_matrix where the iterator finished early (@jabooth)
- #574 Widgets for IPython3 (@nontas, @patricksnape, @jabooth)
- #588 test condaci 0.2.1, less noisy slack notifications (@jabooth)
- #568 rescale_pixels() for rescaling the range of pixels (@jabooth)
- #585 Hotfix: suffix change led to double path resolution. (@patricksnape)
- #581 Fix the landmark importer in case the landmark file has a '.' in its filename. (@grigorisg9gr)
- #584 new print_progress visualization function (@jabooth)
- #580 export_pickle now ensures pathlib.Path save as PurePath (@jabooth)
- #582 New readers for Middlebury FLO and FRGC ABS files (@patricksnape)
- #579 Fix the image importer in case of upper case letters in the suffix (@grigorisg9gr)
- #575 Allowing expanding user paths in exporting pickle (@patricksnape)
- #577 Change to using run_test.py (@patricksnape)
- #570 Zoom (@jabooth, @patricksnape)
- #569 Add new point_in_pointcloud kwarg to constrain (@patricksnape)
- #563 TPS Updates (@patricksnape)
- #567 Optional cmaps (@jalabort)
- #559 Graphs with isolated vertices (@nontas)
- #564 Bugfix: PCAModel print (@nontas)
- #565 fixed minor typo in introduction.rst (@evanjbowling)

- #562 IPython3 widgets (@patricksnape, @jalabort)
- #558 Channel roll (@patricksnape)
- #524 BREAKING CHANGE: Channels flip (@patricksnape, @jabooth, @jalabort)
- #512 WIP: remove_all_landmarks convienience method, quick lm filter (@jabooth)
- #554 Bugfix:visualize_images (@nontas)
- #553 Transform docs fixes (@nontas)
- #533 LandmarkGroup.init_with_all_label, init_* convenience constructors (@jabooth, @patricksnape)
- #552 Many fixes for Python 3 support (@patricksnape)
- #532 Incremental PCA (@patricksnape, @jabooth, @jalabort)
- #528 New as_matrix and from_matrix methods (@patricksnape)

1.7.9 0.4.4 (2015/03/05)

A hotfix release for properly handling nan values in the landmark formats. Also, a few other bug fixes crept in:

- Fix 3D Ljson importing
- Fix trim_components on PCA
- Fix setting None key on the landmark manager
- Making mean_pointcloud faster

Also makes an important change to the build configuration that syncs this version of Menpo to IPython 2.x.

Github Pull Requests

- #560 Assorted fixes (@patricksnape)
- #557 Ljson nan fix (@patricksnape)

1.7.10 0.4.3 (2015/02/19)

Adds the concept of nan values to the landmarker format for labelling missing landmarks.

Github Pull Requests

- #556 [0.4.x] Ljson nan/null fixes (@patricksnape)

1.7.11 0.4.2 (2015/02/19)

A hotfix release for landmark groups that have no connectivity.

Github Pull Requests

- #555 don't try and build a Graph with no connectivity (@jabooth)

1.7.12 0.4.1 (2015/02/07)

A hotfix release to enable compatibility with landmarker.io.

Github Pull Requests

- #551 HOTFIX: remove incorrect tojson() methods (@jabooth)

1.7.13 0.4.0 (2015/02/04)

The 0.4.0 release (pending any currently unknown bugs), represents a very significant overhaul of Menpo from v0.3.0. In particular, Menpo has been broken into four distinct packages: Menpo, MenpoFit, Menpo3D and MenpoDetect.

Visualization has had major improvements for 2D viewing, in particular through the use of IPython widgets and explicit options on the viewing methods for common tasks (like changing the landmark marker color). This final release is a much smaller set of changes over the alpha releases, so please check the full changelog for the alphas to see all changes from v0.3.0 to v0.4.0.

Summary of changes since v0.4.0a2:

- Lots of documentation rendering fixes and style fixes including this changelog.
- Move the LJSON format to V2. V1 is now being deprecated over the next version.
- More visualization customization fixes including multiple marker colors for landmark groups.

Github Pull Requests

- #546 IO doc fixes (@jabooth)
- #545 Different marker colour per label (@nontas)
- #543 Bug fix for importing an image, case of a dot in image name. (@grigorisg9gr)
- #544 Move docs to Sphinx 1.3b2 (@patricksnape)
- #536 Docs fixes (@patricksnape)
- #530 Visualization and Widgets upgrade (@patricksnape, @nontas)
- #540 LJSON v2 (@jabooth)
- #537 fix BU3DFE connectivity, pretty JSON files (@jabooth)
- #529 BU3D-FE labeller added (@jabooth)
- #527 fixes paths for pickle importing (@jabooth)
- #525 Fix .rst doc files, auto-generation script (@jabooth)

1.7.14 v0.4.0a2 (2014/12/03)

Alpha 2 moves towards extending the graphing API so that visualization is more dependable.

Summary:

- Add graph classes, `PointUndirectedGraph`, `PointDirectedGraph`, `PointTree`. This makes visualization of landmarks much nicer looking.
- Better support of pickling menpo objects

- Add a bounding box method to *PointCloud* for calculating the correctly oriented bounding box of point clouds.
- Allow PCA to operate in place for large data matrices.

Github Pull Requests

- #522 Add bounding box method to pointclouds (@patricksnape)
- #523 HOTFIX: fix export_pickle bug, add path support (@jabooth)
- #521 menpo.io add pickle support, move to pathlib (@jabooth)
- #520 Documentation fixes (@patricksnape, @jabooth)
- #518 PCA memory improvements, inplace dot product (@jabooth)
- #519 replace wrapt with functools.wraps - we can pickle (@jabooth)
- #517 (@jabooth)
- #514 Remove the use of triplot (@patricksnape)
- #516 Fix how images are converted to PIL (@patricksnape)
- #515 Show the path in the image widgets (@patricksnape)
- #511 2D Rotation convenience constructor, Image.rotate_ccw_about_centre (@jabooth)
- #510 all menpo io glob operations are now always sorted (@jabooth)
- #508 visualize image on MaskedImage reports Mask proportion (@jabooth)
- #509 path is now preserved on image warping (@jabooth)
- #507 fix rounding issue in n_components (@jabooth)
- #506 is_tree update in Graph (@nontas)
- #505 (@nontas)
- #504 explicitly have kwarg in IO for landmark extensions (@jabooth)
- #503 Update the README (@patricksnape)

1.7.15 v0.4.0a1 (2014/10/31)

This first alpha release makes a number of large, breaking changes to Menpo from v0.3.0. The biggest change is that Menpo3D and MenpoFit were created and thus all AAM and 3D visualization/rasterization code has been moved out of the main Menpo repository. This is working towards Menpo being pip installable.

Summary:

- Fixes memory leak whereby weak references were being kept between landmarks and their host objects. The Landmark manager now no longer keeps references to its host object. This also helps with serialization.
- Use pathlib instead of strings for paths in the `io` module.
- Importing of builtin assets from a simple function
- Improve support for image importing (including ability to import without normalising)
- Add fast methods for image warping, `warp_to_mask` and `warp_to_shape` instead of `warp_to`
- Allow masking of triangle meshes

- Add IPython visualization widgets for our core types
- All expensive properties (properties that would be worth caching in a variable and are not merely a lookup) are changed to methods.

Github Pull Requests

- #502 Fixes pseudoinverse for Alignment Transforms (@jalabort, @patricksnape)
- #501 Remove menpofit widgets (@nontas)
- #500 Shapes widget (@nontas)
- #499 spin out AAM, CLM, SDM, ATM and related code to menpofit (@jabooth)
- #498 Minimum spanning tree bug fix (@nontas)
- #492 Some fixes for PIL image importing (@patricksnape)
- #494 Widgets bug fix and Active Template Model widget (@nontas)
- #491 Widgets fixes (@nontas)
- #489 remove _view, fix up color_list -> colour_list (@jabooth)
- #486 Image visualisation improvements (@patricksnape)
- #488 Move expensive image properties to methods (@jabooth)
- #487 Change expensive PCA properties to methods (@jabooth)
- #485 MeanInstanceLinearModel.mean is now a method (@jabooth)
- #452 Advanced widgets (@patricksnape, @nontas)
- #481 Remove 3D (@patricksnape)
- #480 Graphs functionality (@nontas)
- #479 Extract patches on image (@patricksnape)
- #469 Active Template Models (@nontas)
- #478 Fix residuals for AAMs (@patricksnape, @jabooth)
- #474 remove HDF5able making room for h5it (@jabooth)
- #475 Normalize norm and std of Image object (@nontas)
- #472 Daisy features (@nontas)
- #473 Fix from_mask for Trimesh subclasses (@patricksnape)
- #470 expensive properties should really be methods (@jabooth)
- #467 get a progress bar on top level feature computation (@jabooth)
- #466 Spin out rasterization and related methods to menpo3d (@jabooth)
- #465 'me_norm' error type in tests (@nontas)
- #463 goodbye ioinfo, hello path (@jabooth)
- #464 make mayavi an optional dependency (@jabooth)
- #447 Displacements in fitting result (@nontas)
- #451 AppVeyor Windows continuous builds from condaci (@jabooth)

- #445 Serialize fit results (@patricksnape)
- #444 remove pyramid_on_features from Menpo (@jabooth)
- #443 create_pyramid now applies features even if pyramid_on_features=False, SDM uses it too (@jabooth)
- #369 warp_to_mask, warp_to_shape, fast resizing of images (@nontas, @patricksnape, @jabooth)
- #442 add rescale_to_diagonal, diagonal property to Image (@jabooth)
- #441 adds constrain_to_landmarks on BooleanImage (@jabooth)
- #440 pathlib.Path can no be used in menpo.io (@jabooth)
- #439 Labelling fixes (@jabooth, @patricksnape)
- #438 extract_channels (@jabooth)
- #437 GLRasterizer becomes HDF5able (@jabooth)
- #435 import_builtin_asset.ASSET_NAME (@jabooth)
- #434 check_regression_features unified with check_features, classmethods removed from SDM (@jabooth)
- #433 tidy classifiers (@jabooth)
- #432 aam.fitter, clm.fitter, sdm.trainer packages (@jabooth)
- #431 More fitmultilevel tidying (@jabooth)
- #430 Remove classmethods from DeformableModelBuilder (@jabooth)
- #412 First visualization widgets (@jalabort, @nontas)
- #429 Masked image fixes (@patricksnape)
- #426 rename 'feature_type' to 'features throughout Menpo (@jabooth)
- #427 Adds HDF5able serialization support to Menpo (@jabooth)
- #425 Faster cached piecewise affine, Cython varient demoted (@jabooth)
- #424 (@nontas)
- #378 Fitting result fixes (@jabooth, @nontas, @jalabort)
- #423 name now displays on constrained features (@jabooth)
- #421 Travis CI now makes builds, Linux/OS X Python 2.7/3.4 (@jabooth, @patricksnape)
- #400 Features as functions (@nontas, @patricksnape, @jabooth)
- #420 move IOInfo to use pathlib (@jabooth)
- #405 import menpo is now twice as fast (@jabooth)
- #416 waffle.io Badge (@waffle-iron)
- #415 export_mesh with .OBJ exporter (@jabooth, @patricksnape)
- #410 Fix the render_labels logic (@patricksnape)
- #407 Exporters (@patricksnape)
- #406 Fix greyscale PIL images (@patricksnape)
- #404 LandmarkGroup tojson method and PointGraph (@patricksnape)
- #403 Fixes a couple of viewing problems in fitting results (@patricksnape)
- #402 Landmarks fixes (@jabooth, @patricksnape)

- #401 Dogfood landmark_resolver in menpo.io (@jabooth)
- #399 bunch of Python 3 compatibility fixes (@jabooth)
- #398 throughout Menpo. (@jabooth)
- #397 Performance improvements for Similarity family (@jabooth)
- #396 More efficient initialisations of Menpo types (@jabooth)
- #395 remove cyclic target reference from landmarks (@jabooth)
- #393 Groundwork for dense correspondence pipeline (@jabooth)
- #394 weakref to break cyclic references (@jabooth)
- #389 assorted fixes (@jabooth)
- #390 (@jabooth)
- #387 Adds landmark label for tongues (@nontas)
- #386 Adds labels for the ibug eye annotation scheme (@jalabort)
- #382 BUG fixed: block element not reset if norm=0 (@dubzzz)
- #381 Recursive globbing (@jabooth)
- #384 Adds support for odd patch shapes in function extract_local_patches_fast (@jalabort)
- #379 imported textures have ioinfo, docs improvements (@jabooth)

1.7.16 v0.3.0 (2014/05/27)

First public release of Menpo, this release coincided with submission to the ACM Multimedia Open Source Software Competition 2014. This provides the basic scaffolding for Menpo, but it is not advised to use this version over the improvements in 0.4.0.

Github Pull Requests

- #377 Simple fixes (@patricksnape)
- #375 improvements to importing multiple assets (@jabooth)
- #374 Menpo's User guide (@jabooth)

CHAPTER 2

The Menpo API

This section attempts to provide a simple browsing experience for the Menpo documentation. In Menpo, we use legible docstrings, and therefore, all documentation should be easily accessible in any sensible IDE (or IPython) via tab completion. However, this section should make most of the core classes available for viewing online.

2.1 `menpo.base`

2.1.1 Core

Core interfaces of Menpo.

Copyable

class `menpo.base.`**`Copyable`**

Bases: `object`

Efficient copying of classes containing numpy arrays.

Interface that provides a single method for copying classes very efficiently.

`copy``()`

Generate an efficient copy of this object.

Note that Numpy arrays and other *`Copyable`* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns`type(self)` – A copy of this object

Vectorizable

class `menpo.base.`**`Vectorizable`**

Bases: *`Copyable`*

Flattening of rich objects to vectors and rebuilding them back.

Interface that provides methods for 'flattening' an object into a vector, and restoring from the same vectorized form. Useful for statistical analysis of objects, which commonly requires the data to be provided as a single vector.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns`type(self)` – A copy of this object

from_vector(*vector*)
Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returnsobject (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of this object

has_nan_values()
Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returnshas_nan_values (*bool*) – If the vectorized object contains `nan` values.

n_parameters
The length of the vector that this object produces.

Type*int*

Targetable

class menpo.base.**Targetable**
Bases: *Copyable*

Interface for objects that can produce a target *PointCloud*.

This could for instance be the result of an alignment or a generation of a *PointCloud* instance from a shape model.

Implementations must define sensible behavior for:

- what a target is: see *target*
- how to set a target: see *set_target()*
- how to update the object after a target is set: see `_sync_state_from_target()`
- how to produce a new target after the changes: see `_new_target_from_state()`

Note that `_sync_target_from_state()` needs to be triggered as appropriate by subclasses e.g. when `from_vector_inplace` is called. This will in turn trigger `_new_target_from_state()`, which each subclass must implement.

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other `Copyable` objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

set_target(*new_target*)
Update this object so that it attempts to recreate the `new_target`.

Parameters **new_target** (`PointCloud`) – The new target that this object should try and regenerate.

n_dims
The number of dimensions of the `target`.

Type *int*

n_points
The number of points on the `target`.

Type *int*

target
The current `PointCloud` that this object produces.

Type `PointCloud`

LazyList

class menpo.base.**LazyList**(*callables*)
Bases: `Sequence`

An immutable sequence that provides the ability to lazily access objects. In truth, this sequence simply wraps a list of callables which are then indexed and invoked. However, if the callable represents a function that lazily access memory, then this list simply implements a lazy list paradigm.

When slicing, another *LazyList* is returned, containing the subset of callables.

Parameters **callables** (list of *callable*) – A list of *callable* objects that will be invoked if directly indexed.

count(*value*) → integer – return number of occurrences of value

index(*value*) → integer – return first index of value.
Raises ValueError if the value is not present.

classmethod init_from_index_callable(*f*, *n_elements*)
Create a lazy list from a *callable* that expects a single parameter, the index into an underlying sequence. This allows for simply creating a *LazyList* from a *callable* that likely wraps another list in a closure.

Parameters

- **f** (*callable*) – Callable expecting a single integer parameter, index. This is an index into (presumably) an underlying sequence.

• **n_elements** (*int*) – The number of elements in the underlying sequence.

Returnslazy (*LazyList*) – A LazyList where each element returns the underlying indexable object wrapped by `f`.

map (*f*)
Create a new LazyList where the passed callable `f` wraps each element.

`f` should take a single parameter, `x`, that is the result of the underlying callable - it must also return a value. Note that mapping is lazy and thus calling this function should return immediately.

Parameters`f` (*callable*) – Callable to wrap each element with.

Returnslazy (*LazyList*) – A new LazyList where each element is wrapped by `f`.

2.1.2 Convenience

menpo_src_dir_path

`menpo.base.`**`menpo_src_dir_path`**`()`
The path to the top of the menpo Python package.

Useful for locating where the data folder is stored.

Returnspath (`pathlib.Path`) – The full path to the top of the Menpo package

name_of_callable

`menpo.base.`**`name_of_callable`**`(`*c*`)`
Return the name of a callable (function or callable class) as a string. Recurses on partial function to attempt to find the wrapped methods actual name.

Parametersc (*callable*) – A callable class or function, or any valid Python object that can be wrapped with partial.

Returnsname (*str*) – The name of the passed object.

2.1.3 Convenience

MenpoDeprecationWarning

class `menpo.base.`**`MenpoDeprecationWarning`**
Bases: `Warning`

A warning that functionality in Menpo will be deprecated in a future major release.

2.2 `menpo.io`

2.2.1 Input

import_image

`menpo.io.`**`import_image`**`(`*filepath*, *landmark_resolver=<function same_name>*, *normalise=True*`)`
Single image (and associated landmarks) importer.

If an image file is found at *filepath*, returns an *Image* or subclass representing it. By default, landmark files sharing the same filename stem will be imported and attached with a group name based on the extension of the landmark file, although this behavior can be customised (see *landmark_resolver*). If the image defines a mask, this mask will be imported.

Parameters

- **filepath** (*pathlib.Path* or *str*) – A relative or absolute filepath to an image file.
- **landmark_resolver** (*function*, optional) – This function will be used to find landmarks for the image. The function should take one argument (the path to the image) and return a dictionary of the form `{'group_name': 'landmark_filepath'}` Default finds landmarks with the same name as the image file.
- **normalise** (*bool*, optional) – If `True`, normalise the image pixels between 0 and 1 and convert to floating point. If false, the native datatype of the image will be maintained (commonly *uint8*). Note that in general Menpo assumes *Image* instances contain floating point data - if you disable this flag you will have to manually convert the images you import to floating point before doing most Menpo operations. This however can be useful to save on memory usage if you only wish to view or crop images.

Returns **images** (*Image* or list of) – An instantiated *Image* or subclass thereof or a list of images.

import_images

`menpo.io.import_images`(*pattern*, *max_images=None*, *shuffle=False*, *landmark_resolver=<function same_name>*, *normalise=True*, *as_generator=False*, *verbose=False*)

Multiple image (and associated landmarks) importer.

For each image found creates an importer than returns a *Image* or subclass representing it. By default, landmark files sharing the same filename stem will be imported and attached with a group name based on the extension of the landmark file, although this behavior can be customised (see *landmark_resolver*). If the image defines a mask, this mask will be imported.

Note that this is a function returns a *LazyList*. Therefore, the function will return immediately and indexing into the returned list will load an image at run time. If all images should be loaded, then simply wrap the returned *LazyList* in a Python *list*.

Parameters

- **pattern** (*str*) – A glob path pattern to search for images. Every image found to match the glob will be imported one by one. See *image_paths* for more details of what images will be found.
- **max_images** (positive *int*, optional) – If not `None`, only import the first `max_images` found. Else, import all.
- **shuffle** (*bool*, optional) – If `True`, the order of the returned images will be randomised. If `False`, the order of the returned images will be alphanumerically ordered.
- **landmark_resolver** (*function*, optional) – This function will be used to find landmarks for the image. The function should take one argument (the image itself) and return a dictionary of the form `{'group_name': 'landmark_filepath'}` Default finds landmarks with the same name as the image file.
- **normalise** (*bool*, optional) – If `True`, normalise the image pixels between 0 and 1 and convert to floating point. If false, the native datatype of the image will be maintained (commonly *uint8*). Note that in general Menpo assumes *Image* instances contain floating point data - if you disable this flag you will have to manually convert the images you import to

floating point before doing most Menpo operations. This however can be useful to save on memory usage if you only wish to view or crop images.

- **as_generator** (*bool*, optional) – If `True`, the function returns a generator and assets will be yielded one after another when the generator is iterated over.
- **verbose** (*bool*, optional) – If `True` progress of the importing will be dynamically reported with a progress bar.

Returns **lazy_list** (`LazyList` or generator of `Image`) – A `LazyList` or generator yielding `Image` instances found to match the glob pattern provided.

Raises `ValueError` – If no images are found at the provided glob.

Examples

Import images at 20% scale from a huge collection:

```
>>> rescale_20p = lambda x: x.rescale(0.2)
>>> images =  menpo.io.import_images('./massive_image_db/*')  # Returns immediately
>>> images.map(rescale_20p)  # Returns immediately
>>> images[0]  # Get the first image, resize, lazily loaded
```

import_video

`menpo.io.import_video`(*filepath*, *landmark_resolver=<function same_name_video>*, *normalise=True*, *importer_method='ffmpeg'*)

Single video (and associated landmarks) importer.

If a video file is found at *filepath*, returns an `LazyList` wrapping all the frames of the video. By default, landmark files sharing the same filename stem will be imported and attached with a group name based on the extension of the landmark file appended with the frame number, although this behavior can be customised (see *landmark_resolver*).

> **Warning:** This method currently uses imageio to perform the importing in conjunction with the ffmpeg plugin. As of this release, and the release of imageio at the time of writing (1.5.0), the per-frame computation is not very accurate. This may cause errors when importing frames that do not actually map to valid timestamps within the image. Therefore, use this method at your own risk.

Parameters

- **filepath** (*pathlib.Path* or *str*) – A relative or absolute filepath to a video file.
- **landmark_resolver** (*function*, optional) – This function will be used to find landmarks for the video. The function should take two arguments (the path to the video and the frame number) and return a dictionary of the form `{'group_name': 'landmark_filepath'}` Default finds landmarks with the same name as the video file, appended with '_{frame_number}'.
- **normalise** (*bool*, optional) – If `True`, normalise the frame pixels between 0 and 1 and convert to floating point. If `False`, the native datatype of the image will be maintained (commonly *uint8*). Note that in general Menpo assumes `Image` instances contain floating point data - if you disable this flag you will have to manually convert the farmes you import to floating point before doing most Menpo operations. This however can be useful to save on memory usage if you only wish to view or crop the frames.
- **importer_method** (*{'ffmpeg'}, optional*) – A string representing the type of importer to use, by default ffmpeg is used.

Returnsframes (*LazyList*) – An lazy list of *Image* or subclass thereof which wraps the frames of the video. This list can be treated as a normal list, but the frame is only read when the video is indexed or iterated.

Examples

```
>>> video = menpo.io.import_video('video.avi')
>>> # Lazily load the 100th frame without reading the entire video
>>> frame100 = video[100]
```

import_videos

menpo.io.**import_videos**(*pattern*, *max_videos=None*, *shuffle=False*, *landmark_resolver=<function same_name_video>*, *normalise=True*, *importer_method='ffmpeg'*, *as_generator=False*, *verbose=False*)

Multiple video (and associated landmarks) importer.

For each video found yields a *LazyList*. By default, landmark files sharing the same filename stem will be imported and attached with a group name based on the extension of the landmark file appended with the frame number, although this behavior can be customised (see *landmark_resolver*).

Note that this is a function returns a *LazyList*. Therefore, the function will return immediately and indexing into the returned list will load an image at run time. If all images should be loaded, then simply wrap the returned *LazyList* in a Python *list*.

Warning: This method currently uses imageio to perform the importing in conjunction with the ffmpeg plugin. As of this release, and the release of imageio at the time of writing (1.5.0), the per-frame computation is not very accurate. This may cause errors when importing frames that do not actually map to valid timestamps within the image. Therefore, use this method at your own risk.

Parameters

- **pattern** (*str*) – A glob path pattern to search for videos. Every video found to match the glob will be imported one by one. See *video_paths* for more details of what videos will be found.
- **max_videos** (positive *int*, optional) – If not `None`, only import the first `max_videos` found. Else, import all.
- **shuffle** (*bool*, optional) – If `True`, the order of the returned videos will be randomised. If `False`, the order of the returned videos will be alphanumerically ordered.
- **landmark_resolver** (*function*, optional) – This function will be used to find landmarks for the video. The function should take two arguments (the path to the video and the frame number) and return a dictionary of the form `{'group_name': 'landmark_filepath'}` Default finds landmarks with the same name as the video file, appended with '_{frame_number}'.
- **normalise** (*bool*, optional) – If `True`, normalise the frame pixels between 0 and 1 and convert to floating point. If `False`, the native datatype of the image will be maintained (commonly *uint8*). Note that in general Menpo assumes *Image* instances contain floating point data - if you disable this flag you will have to manually convert the frames you import to floating point before doing most Menpo operations. This however can be useful to save on memory usage if you only wish to view or crop the frames.

- **importer_method** (*{'ffmpeg'}, optional*) – A string representing the type of importer to use, by default ffmpeg is used.
- **as_generator** (*bool*, optional) – If `True`, the function returns a generator and assets will be yielded one after another when the generator is iterated over.
- **verbose** (*bool*, optional) – If `True` progress of the importing will be dynamically reported with a progress bar.

Returns lazy_list (`LazyList` or generator of `LazyList`) – A `LazyList` or generator yielding `LazyList` instances that wrap the video object.

Raises `ValueError` – If no videos are found at the provided glob.

Examples

Import videos at and rescale every frame of each video:

```
>>> videos = []
>>> for video in menpo.io.import_videos('./set_of_videos/*'):
>>>     frames = []
>>>     for frame in video:
>>>         # rescale to a sensible size as we go
>>>         frames.append(frame.rescale(0.2))
>>>     videos.append(frames)
```

import_landmark_file

`menpo.io.`**`import_landmark_file`**(*filepath*, *asset=None*)

Single landmark group importer.

If a landmark file is found at `filepath`, returns a `LandmarkGroup` representing it.

Parameters **filepath** (*pathlib.Path* or *str*) – A relative or absolute filepath to an landmark file.

Returns landmark_group (`LandmarkGroup`) – The `LandmarkGroup` that the file format represents.

import_landmark_files

`menpo.io.`**`import_landmark_files`**(*pattern*, *max_landmarks=None*, *shuffle=False*, *as_generator=False*, *verbose=False*)

Import Multiple landmark files.

For each landmark file found returns an importer than returns a `LandmarkGroup`.

Note that this is a function returns a `LazyList`. Therefore, the function will return immediately and indexing into the returned list will load the landmarks at run time. If all landmarks should be loaded, then simply wrap the returned `LazyList` in a Python *list*.

Parameters

- **pattern** (*str*) – A glob path pattern to search for landmark files. Every landmark file found to match the glob will be imported one by one. See `landmark_file_paths` for more details of what landmark files will be found.
- **max_landmarks** (positive *int*, optional) – If not `None`, only import the first `max_landmark_files` found. Else, import all.

- **shuffle** (*bool*, optional) – If True, the order of the returned landmark files will be randomised. If False, the order of the returned landmark files will be alphanumerically ordered.
- **as_generator** (*bool*, optional) – If True, the function returns a generator and assets will be yielded one after another when the generator is iterated over.
- **verbose** (*bool*, optional) – If True progress of the importing will be dynamically reported.

Returns lazy_list (*LazyList* or generator of *LandmarkGroup*) – A *LazyList* or generator yielding *LandmarkGroup* instances found to match the glob pattern provided.

Raises ValueError – If no landmarks are found at the provided glob.

import_pickle

menpo.io.**import_pickle**(*filepath*)

Import a pickle file of arbitrary Python objects.

Menpo unambiguously uses .pkl as it's choice of extension for Pickle files. Menpo also supports automatic importing and exporting of gzip compressed pickle files - just choose a filepath ending pkl.gz and gzip compression will automatically be applied. Compression can massively reduce the filesize of a pickle file at the cost of longer import and export times.

Parameters **filepath** (*pathlib.Path* or *str*) – A relative or absolute filepath to a .pkl or .pkl.gz file.

Returns object (*object*) – Whatever Python objects are present in the Pickle file

import_pickles

menpo.io.**import_pickles**(*pattern*, *max_pickles=None*, *shuffle=False*, *as_generator=False*, *verbose=False*)

Import multiple pickle files.

Menpo unambiguously uses .pkl as it's choice of extension for pickle files. Menpo also supports automatic importing of gzip compressed pickle files - matching files with extension pkl.gz will be automatically ungzipped and imported.

Note that this is a function returns a *LazyList*. Therefore, the function will return immediately and indexing into the returned list will load the landmarks at run time. If all pickles should be loaded, then simply wrap the returned *LazyList* in a Python *list*.

Parameters

- **pattern** (*str*) – The glob path pattern to search for pickles. Every pickle file found to match the glob will be imported one by one.
- **max_pickles** (positive *int*, optional) – If not None, only import the first max_pickles found. Else, import all.
- **shuffle** (*bool*, optional) – If True, the order of the returned pickles will be randomised. If False, the order of the returned pickles will be alphanumerically ordered.
- **as_generator** (*bool*, optional) – If True, the function returns a generator and assets will be yielded one after another when the generator is iterated over.
- **verbose** (*bool*, optional) – If True progress of the importing will be dynamically reported.

Returnslazy_list (*LazyList* or generator of Python objects) – A *LazyList* or generator yielding Python objects inside the pickle files found to match the glob pattern provided.

RaisesValueError – If no pickles are found at the provided glob.

import_builtin_asset

menpo.io.**import_builtin_asset**()

This is a dynamically generated method. This method is designed to automatically generate import methods for each data file in the data folder. This method it designed to be tab completed, so you do not need to call this method explicitly. It should be treated more like a property that will dynamically generate functions that will import the shipped data. For example:

```
>>> import menpo
>>> bb_image = menpo.io.import_builtin_asset.breakingbad_jpg()
```

2.2.2 Output

export_image

menpo.io.**export_image**(*image*, *fp*, *extension=None*, *overwrite=False*)

Exports a given image. The fp argument can be either a *Path* or any Python type that acts like a file. If a file is provided, the extension kwarg **must** be provided. If no extension is provided and a *str* filepath is provided, then the export type is calculated based on the filepath extension.

Due to the mix of string and file types, an explicit overwrite argument is used which is False by default.

Parameters

- **image** (*Image*) – The image to export.
- **fp** (*Path* or *file*-like object) – The Path or file-like object to save the object at/into.
- **extension** (*str* or None, optional) – The extension to use, this must match the file path if the file path is a string. Determines the type of exporter that is used.
- **overwrite** (*bool*, optional) – Whether or not to overwrite a file if it already exists.

Raises

- ValueError – File already exists and overwrite != True
- ValueError – fp is a *str* and the extension is not None and the two extensions do not match
- ValueError – fp is a *file*-like object and extension is None
- ValueError – The provided extension does not match to an existing exporter type (the output type is not supported).

export_landmark_file

menpo.io.**export_landmark_file**(*landmark_group*, *fp*, *extension=None*, *overwrite=False*)

Exports a given landmark group. The fp argument can be either or a *str* or any Python type that acts like a file. If a file is provided, the extension kwarg **must** be provided. If no extension is provided and a *str* filepath is provided, then the export type is calculated based on the filepath extension.

Due to the mix in string and file types, an explicit overwrite argument is used which is False by default.

Parameters

- **landmark_group** (*LandmarkGroup*) – The landmark group to export.
- **fp** (*Path* or *file*-like object) – The Path or file-like object to save the object at/into.
- **extension** (*str* or None, optional) – The extension to use, this must match the file path if the file path is a string. Determines the type of exporter that is used.
- **overwrite** (*bool*, optional) – Whether or not to overwrite a file if it already exists.

Raises

- `ValueError` – File already exists and `overwrite` != `True`
- `ValueError` – `fp` is a *str* and the `extension` is not `None` and the two extensions do not match
- `ValueError` – `fp` is a *file*-like object and `extension` is `None`
- `ValueError` – The provided extension does not match to an existing exporter type (the output type is not supported).

export_pickle

`menpo.io.`**`export_pickle`**(*obj*, *fp*, *overwrite=False*, *protocol=2*)

Exports a given collection of Python objects with Pickle.

The `fp` argument can be either a *Path* or any Python type that acts like a file. If `fp` is a path, it must have the suffix *.pkl* or *.pkl.gz*. If *.pkl*, the object will be pickled using Pickle protocol 2 without compression. If *.pkl.gz* the object will be pickled using Pickle protocol 2 with gzip compression (at a fixed compression level of 3).

Note that a special exception is made for *pathlib.Path* objects - they are pickled down as a *pathlib.PurePath* so that pickles can be easily moved between different platforms.

Parameters

- **obj** (`object`) – The object to export.
- **fp** (*Path* or *file*-like object) – The string path or file-like object to save the object at/into.
- **overwrite** (*bool*, optional) – Whether or not to overwrite a file if it already exists.
- **protocol** (*int*, optional) – The Pickle protocol used to serialize the file. The protocols were introduced in different versions of python, thus it is recommended to save with the highest protocol version that your python distribution can support. The protocol refers to:

Protocol	Functionality
0	Simplest protocol for text mode, backwards compatible.
1	Protocol for binary mode, backwards compatible.
2	Wider support for classes, compatible with python >= 2.3.
3	Support for byte objects, compatible with python >= 3.0.
4	Support for large objects, compatible with python >= 3.4.

Raises

- `ValueError` – File already exists and `overwrite` != `True`
- `ValueError` – `fp` is a *file*-like object and `extension` is `None`
- `ValueError` – The provided extension does not match to an existing exporter type (the output type is not supported).

2.2.3 Path Operations

image_paths

`menpo.io.image_paths(pattern)`
Return image filepaths that Menpo can import that match the glob pattern.

landmark_file_paths

`menpo.io.landmark_file_paths(pattern)`
Return landmark file filepaths that Menpo can import that match the glob pattern.

pickle_paths

`menpo.io.pickle_paths(pattern)`
Return pickle filepaths that Menpo can import that match the glob pattern.

video_paths

`menpo.io.video_paths(pattern)`
Return video filepaths that Menpo can import that match the glob pattern.

data_path_to

`menpo.io.data_path_to(asset_filename)`
The path to a builtin asset in the ./data folder on this machine.

Parameters **asset_filename** (*str*) – The filename (with extension) of a file builtin to Menpo. The full set of allowed names is given by `ls_builtin_assets()`

Returns **data_path** (*pathlib.Path*) – The path to a given asset in the ./data folder

Raises `ValueError` – If the asset_filename doesn't exist in the *data* folder.

data_dir_path

`menpo.io.data_dir_path()`
A path to the Menpo built in ./data folder on this machine.

Returns `pathlib.Path` – The path to the local Menpo ./data folder

ls_builtin_assets

`menpo.io.ls_builtin_assets()`
List all the builtin asset examples provided in Menpo.

Returns *list of strings* – Filenames of all assets in the data directory shipped with Menpo

2.3 menpo.image

2.3.1 Image Types

Image

class menpo.image.**Image**(*image_data*, *copy=True*)

Bases: *Vectorizable*, Landmarkable, Viewable, LandmarkableViewable

An n-dimensional image.

Images are n-dimensional homogeneous regular arrays of data. Each spatially distinct location in the array is referred to as a *pixel*. At a pixel, k distinct pieces of information can be stored. Each datum at a pixel is refereed to as being in a *channel*. All pixels in the image have the same number of channels, and all channels have the same data-type (*float64*).

Parameters

- **image_data** ((C, M, N ..., Q) *ndarray*) – Array representing the image pixels, with the first axis being channels.
- **copy** (*bool*, optional) – If False, the image_data will not be copied on assignment. Note that this will miss out on additional checks. Further note that we still demand that the array is C-contiguous - if it isn't, a copy will be generated anyway. In general, this should only be used if you know what you are doing.

Raises

- Warning – If copy=False cannot be honoured
- ValueError – If the pixel array is malformed

_view_2d(*figure_id=None*, *new_figure=False*, *channels=None*, *interpolation='bilinear'*, *cmap_name=None*, *alpha=1.0*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

View the image using the default image viewer. This method will appear on the Image as view if the Image is 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **channels** (*int* or *list* of *int* or all or None) – If *int* or *list* of *int*, the specified channel(s) will be rendered. If all, all the channels will be rendered in subplots. If None and the image is RGB, it will be rendered in RGB mode. If None and the image is not RGB, it is equivalent to all.
- **interpolation** (*See Below, optional*) – The interpolation used to render the image. For example, if bilinear, the image will be smooth and if nearest, the image will be pixelated. Example options

```
{none, nearest, bilinear, bicubic, spline16, spline36,
hanning, hamming, hermite, kaiser, quadric, catrom, gaussian,
bessel, mitchell, sinc, lanczos}
```

- **cmap_name** (*str*, optional,) – If `None`, single channel and three channel images default to greyscale and rgb colormaps respectively.
- **alpha** (*float*, optional) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the Image as a percentage of the Image's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the Image as a percentage of the Image's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.

Returnsviewer (*ImageViewer*) – The image viewing object.

_view_landmarks_2d(*channels=None*, *group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *interpolation='bilinear'*, *cmap_name=None*, *alpha=1.0*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05, 1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **channels** (*int* or *list* of *int* or `all` or `None`) – If *int* or *list* of *int*, the specified channel(s) will be rendered. If `all`, all the channels will be rendered in subplots. If `None` and the image is RGB, it will be rendered in RGB mode. If `None` and the image is not RGB, it is equivalent to `all`.
- **group** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **with_labels** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.
- **without_labels** (`None` or *str* or *list* of *str*, optional) – If not `None`, show all except the given label(s). Should **not** be used with the `with_labels` kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **interpolation** (*See Below, optional*) – The interpolation used to render the image. For example, if `bilinear`, the image will be smooth and if `nearest`, the image will be pixelated. Example options

```
{none, nearest, bilinear, bicubic, spline16, spline36, hanning,
hamming, hermite, kaiser, quadric, catrom, gaussian, bessel,
mitchell, sinc, lanczos}
```

- **cmap_name** (*str*, optional,) – If `None`, single channel and three channel images default to greyscale and rgb colormaps respectively.
- **alpha** (*float*, optional) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.

- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** ({normal, italic, oblique}, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See Below, optional*) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.

- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the Image as a percentage of the Image's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the Image as a percentage of the Image's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None` optional) – The size of the figure in inches.

Raises

- `ValueError` – If both `with_labels` and `without_labels` are passed.
- `ValueError` – If the landmark manager doesn't contain the provided group label.

as_PILImage(*out_dtype=<type 'numpy.uint8'>*)

Return a PIL copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.

Parameters **out_dtype** (*np.dtype*, optional) – The dtype the output array should be.

Returns **pil_image** (*PILImage*) – PIL copy of image

Raises

- `ValueError` – If image is not 2D and has 1 channel or 3 channels.
- `ValueError` – If pixels data type is *float32* or *float64* and the pixel range is outside of `[0, 1]`
- `ValueError` – If the output dtype is unsupported. Currently uint8 is supported.

as_greyscale(*mode='luminosity'*, *channel=None*)

Returns a greyscale version of the image. If the image does *not* represent a 2D RGB image, then the `luminosity` mode will fail.

Parameters

- **mode** (`{average, luminosity, channel}`, optional) –

mode	Greyscale Algorithm
average	Equal average of all channels
luminosity	Calculates the luminance using the CCIR 601 formula:
	$$Y' = 0.2989R' + 0.5870G' + 0.1140B'$$
channel	A specific channel is chosen as the intensity value.

• **channel** (*int*, optional) – The channel to be taken. Only used if mode is `channel`.

Returns **greyscale_image** (*MaskedImage*) – A copy of this image in greyscale.

as_histogram(*keep_channels=True*, *bins='unique'*)

Histogram binning of the values of this image.

Parameters

• **keep_channels** (*bool*, optional) – If set to `False`, it returns a single histogram for all the channels of the image. If set to `True`, it returns a *list* of histograms, one for each channel.

• **bins** (`{unique}`, positive *int* or sequence of scalars, optional) – If set equal to `'unique'`, the bins of the histograms are centred on the unique values of each channel. If set equal to a positive *int*, then this is the number of bins. If set equal to a sequence of scalars, these will be used as bins centres.

Returns

• **hist** (*ndarray* or *list* with `n_channels` *ndarrays* inside) – The histogram(s). If `keep_channels=False`, then hist is an *ndarray*. If `keep_channels=True`, then hist is a *list* with `len(hist)=n_channels`.

• **bin_edges** (*ndarray* or *list* with *n_channels ndarrays* inside) – An array or a list of arrays corresponding to the above histograms that store the bins' edges.

Raises `ValueError` – Bins can be either 'unique', positive int or a sequence of scalars.

Examples

Visualizing the histogram when a list of array bin edges is provided:

```
>>> hist, bin_edges = image.as_histogram()
>>> for k in range(len(hist)):
>>>     plt.subplot(1,len(hist),k)
>>>     width = 0.7 * (bin_edges[k][1] - bin_edges[k][0])
>>>     centre = (bin_edges[k][:-1] + bin_edges[k][1:]) / 2
>>>     plt.bar(centre, hist[k], align='center', width=width)
```

---

as_imageio(*out_dtype=<type 'numpy.uint8'>*)

Return an Imageio copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.

Parameters **out_dtype** (*np.dtype*, optional) – The dtype the output array should be.

Returns **imageio_image** (*ndarray*) – Imageio image (which is just a numpy ndarray with the channels as the last axis).

Raises

• `ValueError` – If image is not 2D and has 1 channel or 3 channels.

- ValueError – If pixels data type is *float32* or *float64* and the pixel range is outside of [0, 1]
- ValueError – If the output dtype is unsupported. Currently uint8 and uint16 are supported.

as_masked(*mask=None*, *copy=True*)

Return a copy of this image with an attached mask behavior.

A custom mask may be provided, or None. See the *MaskedImage* constructor for details of how the kwargs will be handled.

Parameters

- **mask** ((self.shape) *ndarray* or *BooleanImage*) – A mask to attach to the newly generated masked image.
- **copy** (*bool*, optional) – If False, the produced *MaskedImage* will share pixels with self. Only suggested to be used for performance.

Returns**masked_image** (*MaskedImage*) – An image with the same pixels and landmarks as this one, but with a mask.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns**vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounds()

The bounds of the image, minimum is always (0, 0). The maximum is the maximum **index** that can be used to index into the image for each dimension. Therefore, bounds will be of the form: ((0, 0), (self.height - 1, self.width - 1)) for a 2D image.

Note that this is akin to supporting a nearest neighbour interpolation. Although the *actual* maximum subpixel value would be something like self.height - eps where eps is some value arbitrarily close to 0, this value at least allows sampling without worrying about floating point error.

Type*tuple*

centre()

The geometric centre of the Image - the subpixel that is in the middle.

Useful for aligning shapes and images.

Type(n_dims,) *ndarray*

constrain_landmarks_to_bounds()

Deprecated - please use the equivalent constrain_to_bounds method now on PointCloud, in conjunction with the new Image bounds() method. For example:

```
>>> im.constrain_landmarks_to_bounds()  # Equivalent to below
>>> im.landmarks['test'] = im.landmarks['test'].lms.constrain_to_bounds(im.bounds)
```

constrain_points_to_bounds(*points*)

Constrains the points provided to be within the bounds of this image.

Parameters**points** ((d,) *ndarray*) – Points to be snapped to the image boundaries.

Returns**bounded_points** ((d,) *ndarray*) – Points snapped to not stray outside the image edges.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

crop (*min_indices*, *max_indices*, *constrain_to_boundary=False*, *return_transform=False*)
Return a cropped copy of this image using the given minimum and maximum indices. Landmarks are correctly adjusted so they maintain their position relative to the newly cropped image.

Parameters

- **min_indices** (`(n_dims,)` *ndarray*) – The minimum index over each dimension.
- **max_indices** (`(n_dims,)` *ndarray*) – The maximum index over each dimension.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an `ImageBoundaryError` will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the cropping is also returned.

Returns

- **cropped_image** (*type(self)*) – A new instance of self, but cropped.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises

- `ValueError` – `min_indices` and `max_indices` both have to be of length `n_dims`. All `max_indices` must be greater than `min_indices`.
- `ImageBoundaryError` – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks (*group=None*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)
Return a copy of this image cropped so that it is bounded around a set of landmarks with an optional `n_pixel` boundary

Parameters

- **group** (*str*, optional) – The key of the landmark set that should be used. If `None` and if there is only one set of landmarks, this set will be used.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an :map‘ImageBoundaryError‘ will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the cropping is also returned.

Returns

- **image** (`Image`) – A copy of this image cropped to its landmarks.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises `ImageBoundaryError` – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks_proportion (*boundary_proportion*, *group=None*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)
Crop this image to be bounded around a set of landmarks with a border proportional to the landmark spread or range.

Parameters

- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.

- **group** (*str*, optional) – The key of the landmark set that should be used. If None and if there is only one set of landmarks, this set will be used.
- **minimum** (*bool*, optional) – If True the specified proportion is relative to the minimum value of the landmarks' per-dimension range; if False w.r.t. the maximum value of the landmarks' per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If True, the crop will be snapped to not go beyond this images boundary. If False, an ImageBoundaryError will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the Transform object that was used to perform the cropping is also returned.

Returns

- **image** (Image) – This image, cropped to its landmarks with a border proportional to the landmark spread or range.
- **transform** (Transform) – The transform that was used. It only applies if *return_transform* is True.

Raises ImageBoundaryError – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud(*pointcloud*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional n_pixel boundary.

Parameters

- **pointcloud** (PointCloud) – The pointcloud to crop around.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If True the crop will be snapped to not go beyond this images boundary. If False, an :map'ImageBoundaryError' will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the Transform object that was used to perform the cropping is also returned.

Returns

- **image** (Image) – A copy of this image cropped to the bounds of the pointcloud.
- **transform** (Transform) – The transform that was used. It only applies if *return_transform* is True.

Raises ImageBoundaryError – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud_proportion(*pointcloud*, *boundary_proportion*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional n_pixel boundary.

Parameters

- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.
- **pointcloud** (PointCloud) – The pointcloud to crop around.
- **minimum** (*bool*, optional) – If True the specified proportion is relative to the minimum value of the pointclouds' per-dimension range; if False w.r.t. the maximum value of the pointclouds' per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If True, the crop will be snapped to not go beyond this images boundary. If False, an ImageBoundaryError will be raised if an attempt is made to go beyond

the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the cropping is also returned.

Returns

- **image** (`Image`) – A copy of this image cropped to the border proportional to the pointcloud spread or range.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises `ImageBoundaryError` – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

diagonal()

The diagonal size of this image

Type *float*

extract_channels(*channels*)

A copy of this image with only the specified channels.

Parameters **channels** (*int* or *[int]*) – The channel index or *list* of channel indices to retain.

Returns **image** (*type(self)*) – A copy of this image with only the channels requested.

extract_patches(*patch_centers*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)

Extract a set of patches from an image. Given a set of patch centers and a patch size, patches are extracted from within the image, centred on the given coordinates. Sample offsets denote a set of offsets to extract from within a patch. This is very useful if you want to extract a dense set of features around a set of landmarks and simply sample the same grid of patches around the landmarks.

If sample offsets are used, to access the offsets for each patch you need to slice the resulting *list*. So for 2 offsets, the first centers offset patches would be `patches[:2]`.

Currently only 2D images are supported.

Parameters

- **patch_centers** (`PointCloud`) – The centers to extract patches around.
- **patch_shape** (`(1, n_dims)` *tuple* or *ndarray*, optional) – The size of the patch to extract
- **sample_offsets** (`(n_offsets, n_dims)` *ndarray* or `None`, optional) – The offsets to sample from within a patch. So `(0, 0)` is the centre of the patch (no offset) and `(1, 0)` would be sampling the patch from 1 pixel up the first axis away from the centre. If `None`, then no offsets are applied.
- **as_single_array** (*bool*, optional) – If `True`, an `(n_center, n_offset, n_channels, patch_shape)` *ndarray*, thus a single numpy array is returned containing each patch. If `False`, a *list* of `n_center * n_offset` `Image` objects is returned representing each patch.

Returns **patches** (*list* or *ndarray*) – Returns the extracted patches. Returns a list if `as_single_array=True` and an *ndarray* if `as_single_array=False`.

Raises `ValueError` – If image is not 2D

extract_patches_around_landmarks(*group=None*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)

Extract patches around landmarks existing on this image. Provided the group label and optionally the landmark label extract a set of patches.

See *extract_patches* for more information.

Currently only 2D images are supported.

Parameters

- **group** (*str* or `None`, optional) – The landmark group to use as patch centres.
- **patch_shape** (*tuple* or *ndarray*, optional) – The size of the patch to extract

- **sample_offsets** ((n_offsets, n_dims) *ndarray* or None, optional) – The offsets to sample from within a patch. So (0, 0) is the centre of the patch (no offset) and (1, 0) would be sampling the patch from 1 pixel up the first axis away from the centre. If None, then no offsets are applied.
- **as_single_array** (*bool*, optional) – If True, an (n_center, n_offset, n_channels, patch_shape) *ndarray*, thus a single numpy array is returned containing each patch. If False, a *list* of n_center * n_offset *Image* objects is returned representing each patch.

Returns patches (*list* or *ndarray*) – Returns the extracted patches. Returns a list if as_single_array=True and an *ndarray* if as_single_array=False.

Raises ValueError – If image is not 2D

from_vector (*vector*, *n_channels=None*, *copy=True*)

Takes a flattened vector and returns a new image formed by reshaping the vector to the correct pixels and channels.

The *n_channels* argument is useful for when we want to add an extra channel to an image but maintain the shape. For example, when calculating the gradient.

Note that landmarks are transferred in the process.

Parameters

- **vector** ((n_parameters,) *ndarray*) – A flattened vector of all pixels and channels of an image.
- **n_channels** (*int*, optional) – If given, will assume that vector is the same shape as this image, but with a possibly different number of channels.
- **copy** (*bool*, optional) – If False, the vector will not be copied in creating the new image.

Returns image (*Image*) – New image of same shape as this image and the number of specified channels.

Raises Warning – If the copy=False flag cannot be honored

from_vector_inplace (*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters vector ((n_parameters,) *ndarray*) – Flattened representation of this object

gaussian_pyramid (*n_levels=3*, *downscale=2*, *sigma=None*)

Return the gaussian pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters

- **n_levels** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **downscale** (*float*, optional) – Downscale factor.
- **sigma** (*float*, optional) – Sigma for gaussian filter. Default is downscale / 3. which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the gaussian distribution.

Yields image_pyramid (*generator*) – Generator yielding pyramid layers as *Image* objects.

has_landmarks_outside_bounds ()

Indicates whether there are landmarks located outside the image bounds.

Type *bool*

has_nan_values ()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns has_nan_values (*bool*) – If the vectorized object contains nan values.

indices()

Return the indices of all pixels in this image.

Type(n_dims, n_pixels) ndarray

classmethod **init_blank**(*shape*, *n_channels=1*, *fill=0*, *dtype=<type 'float'>*)

Returns a blank image.

Parameters

- **shape** (*tuple* or *list*) – The shape of the image. Any floating point values are rounded up to the nearest integer.
- **n_channels** (*int*, optional) – The number of channels to create the image with.
- **fill** (*int*, optional) – The value to fill all pixels with.
- **dtype** (numpy data type, optional) – The data type of the image.

Returnsblank_image (*Image*) – A new image of the requested size.

classmethod **init_from_pointcloud**(*pointcloud*, *group=None*, *boundary=0*, *n_channels=1*, *fill=0*, *dtype=<type 'float'>*, *return_transform=False*)

Create an Image that is big enough to contain the given pointcloud. The pointcloud will be translated to the origin and then translated according to its bounds in order to fit inside the new image. An optional boundary can be provided in order to increase the space around the boundary of the pointcloud. The boundary will be added to *all sides of the image* and so a boundary of 5 provides 10 pixels of boundary total for each dimension.

Parameters

- **pointcloud** (*PointCloud*) – Pointcloud to place inside the newly created image.
- **group** (*str*, optional) – If None, the pointcloud will only be used to create the image. If a *str* then the pointcloud will be attached as a landmark group to the image, with the given string as key.
- **boundary** (*float*) – A optional padding distance that is added to the pointcloud bounds. Default is 0, meaning the max/min of tightest possible containing image is returned.
- **n_channels** (*int*, optional) – The number of channels to create the image with.
- **fill** (*int*, optional) – The value to fill all pixels with.
- **dtype** (numpy data type, optional) – The data type of the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to adjust the PointCloud in order to build the image, is returned.

Returns

- **image** (type(cls) Image or subclass) – A new image with the same size as the given pointcloud, optionally with the pointcloud attached as landmarks.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

classmethod **init_from_rolled_channels**(*pixels*)

Create an Image from a set of pixels where the channels axis is on the last axis (the back). This is common in other frameworks, and therefore this method provides a convenient means of creating a menpo Image from such data. Note that a copy is always created due to the need to rearrange the data.

Parameterspixels ((M, N ..., Q, C) *ndarray*) – Array representing the image pixels, with the last axis being channels.

Returnsimage (*Image*) – A new image from the given pixels, with the FIRST axis as the channels.

mirror(*axis=1*, *return_transform=False*)

Return a copy of this image, mirrored/flipped about a certain axis.

Parameters

- **axis** (*int*, optional) – The axis about which to mirror the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the mirroring is also returned.

Returns

- **mirrored_image** (`type(self)`) – The mirrored image.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises

- `ValueError` – axis cannot be negative
- `ValueError` – axis={} but the image has {} dimensions

normalize_norm(*mode='all'*, ***kwargs*)

Returns a copy of this image normalized such that its pixel values have zero mean and its norm equals 1.

Parameters **mode** (`{all, per_channel}`, optional) – If `all`, the normalization is over all channels. If `per_channel`, each channel individually is mean centred and unit norm.

Returns **image** (`type(self)`) – A copy of this image, normalized.

normalize_std(*mode='all'*, ***kwargs*)

Returns a copy of this image normalized such that its pixel values have zero mean and unit variance.

Parameters **mode** (`{all, per_channel}`, optional) – If `all`, the normalization is over all channels. If `per_channel`, each channel individually is mean centred and normalized in variance.

Returns **image** (`type(self)`) – A copy of this image, normalized.

pixels_range()

The range of the pixel values (min and max pixel values).

Returns **min_max** (`(dtype, dtype)`) – The minimum and maximum value of the pixels array.

pyramid(*n_levels=3*, *downscale=2*)

Return a rescaled pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters

- **n_levels** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **downscale** (*float*, optional) – Downscale factor.

Yields **image_pyramid** (*generator*) – Generator yielding pyramid layers as *Image* objects.

rasterize_landmarks(*group=None*, *render_lines=True*, *line_style='-'*, *line_colour='b'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=1*, *marker_face_colour='b'*, *marker_edge_colour='b'*, *marker_edge_width=1*, *backend='matplotlib'*)

This method provides the ability to rasterize 2D landmarks onto the image. The returned image has the specified landmark groups rasterized onto the image - which is useful for things like creating result examples or rendering videos with annotations.

Since multiple landmark groups can be specified, all arguments can take lists of parameters that map to the provided groups list. Therefore, the parameters must be lists of the correct length or a single parameter to apply to every landmark group.

Multiple backends are provided, all with different strengths. The 'pillow' backend is very fast, but not very flexible. The *matplotlib* backend should be feature compatible with other Menpo rendering methods, but is much slower due to the overhead of creating a figure to render into.

Parameters

- **group** (*str* or *list* of *str*, optional) – The landmark group key, or a list of keys.
- **render_lines** (*bool*, optional) – If `True`, and the provided landmark group is a *PointDirectedGraph*, the edges are rendered.
- **line_style** (*str*, optional) – The style of the edge line. Not all backends support this argument.

- **line_colour** (*str* or *tuple*, optional) – A Matplotlib style colour or a backend dependant colour.
- **line_width** (*int*, optional) – The width of the line to rasterize.
- **render_markers** (*bool*, optional) – If `True`, render markers at the coordinates of each landmark.
- **marker_style** (*str*, optional) – A Matplotlib marker style. Not all backends support all marker styles.
- **marker_size** (*int*, optional) – The size of the marker - different backends use different scale spaces so consistent output may by difficult.
- **marker_face_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_width** (*int*, optional) – The width of the marker edge. Not all backends support this.
- **backend** (*{'matplotlib', 'pillow'}, optional*) – The backend to use.

Returns **rasterized_image** (`Image`) – The image with the landmarks rasterized directly into the pixels.

Raises

- `ValueError` – Only 2D images are supported.
- `ValueError` – Only RGB (3-channel) or Greyscale (1-channel) images are supported.

rescale(*scale*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled by a given factor. Landmarks are rescaled appropriately.

Parameters

- **scale** (*float* or *tuple* of *floats*) – The scale factor. If a tuple, the scale to apply to each dimension. If a single *float*, the scale will be applied uniformly across each dimension.
- **round** ({`ceil`, `floor`, `round`}, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (`type(self)`) – A copy of this image, rescaled.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises ValueError: – If less scales than dimensions are provided. If any scale is less than or equal to 0.

rescale_landmarks_to_diagonal_range(*diagonal_range*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the `diagonal_range` of the bounding box containing its landmarks matches the specified `diagonal_range` range.

Parameters

- **diagonal_range** ((n_dims,) *ndarray*) – The diagonal_range range that we want the landmarks of the returned image to have.
- **group** (*str*, optional) – The key of the landmark set that should be used. If None and if there is only one set of landmarks, this set will be used.
- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (type(self)) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

rescale_pixels(*minimum*, *maximum*, *per_channel=True*)

A copy of this image with pixels linearly rescaled to fit a range.

Note that the only pixels that will considered and rescaled are those that feature in the vectorized form of this image. If you want to use this routine on all the pixels in a *MaskedImage*, consider using *as_unmasked()* prior to this call.

Parameters

- **minimum** (*float*) – The minimal value of the rescaled pixels
- **maximum** (*float*) – The maximal value of the rescaled pixels
- **per_channel** (*boolean*, optional) – If True, each channel will be rescaled independently. If False, the scaling will be over all channels.

Returns **rescaled_image** (type(self)) – A copy of this image with pixels linearly rescaled to fit in the range provided.

rescale_to_diagonal(*diagonal*, *round='ceil'*, *return_transform=False*)

Return a copy of this image, rescaled so that the it's diagonal is a new size.

Parameters

- **diagonal** (*int*) – The diagonal size of the new image.
- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (*type(self)*) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

rescale_to_pointcloud(*pointcloud*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the scale of a particular group of landmarks matches the scale of the passed reference pointcloud.

Parameters

- **pointcloud** (*PointCloud*) – The reference pointcloud to which the landmarks specified by `group` will be scaled to match.
- **group** (*str*, optional) – The key of the landmark set that should be used. If `None`, and if there is only one set of landmarks, this set will be used.
- **round** (`{ceil, floor, round}`, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (`type(self)`) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

resize (*shape*, *order=1*, *return_transform=False*)

Return a copy of this image, resized to a particular shape. All image information (landmarks, and mask in the case of *MaskedImage*) is resized appropriately.

Parameters

- **shape** (*tuple*) – The new shape to resize to.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the resize is also returned.

Returns

- **resized_image** (`type(self)`) – A copy of this image, resized.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

RaisesValueError: – If the number of dimensions of the new shape does not match the number of dimensions of the image.

rolled_channels ()

Returns the pixels matrix, with the channels rolled to the back axis. This may be required for interacting with external code bases that require images to have channels as the last axis, rather than the menpo convention of channels as the first axis.

Returnsrolled_channels (*ndarray*) – Pixels with channels as the back (last) axis.

rotate_ccw_about_centre (*theta*, *degrees=True*, *retain_shape=False*, *cval=0.0*, *round='round'*, *order=1*, *return_transform=False*)

Return a copy of this image, rotated counter-clockwise about its centre.

Note that the *retain_shape* argument defines the shape of the rotated image. If `retain_shape=True`, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If `retain_shape=False`, then the returned image has the correct size so that the whole area of the current image is included.

Parameters

- **theta** (*float*) – The angle of rotation about the centre.
- **degrees** (*bool*, optional) – If `True`, *theta* is interpreted in degrees. If `False`, `theta` is interpreted as radians.
- **retain_shape** (*bool*, optional) – If `True`, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If `False`, then the returned image has the correct size so that the whole area of the current image is included.
- **cval** (*float*, optional) – The value to be set outside the rotated image boundaries.
- **round** (`{'ceil', 'floor', 'round'}`, optional) – Rounding function to be applied to floating point shapes. This is only used in case `retain_shape=True`.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range `[0,5]`. This is only used in case `retain_shape=True`.

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rotation is also returned.

Returns

- **rotated_image** (`type(self)`) – The rotated image.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises`ValueError` – Image rotation is presently only supported on 2D images

sample(*points_to_sample*, *order=1*, *mode='constant'*, *cval=0.0*)

Sample this image at the given sub-pixel accurate points. The input PointCloud should have the same number of dimensions as the image e.g. a 2D PointCloud for a 2D multi-channel image. A numpy array will be returned the has the values for every given point across each channel of the image.

Parameters

- **points_to_sample** (`PointCloud`) – Array of points to sample from the image. Should be *(n_points, n_dims)*
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]. See warp_to_shape for more information.
- **mode** (`{constant, nearest, reflect, wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.

Returns**sampled_pixels** ((*n_points*, *n_channels*) *ndarray*) – The interpolated values taken across every channel of the image.

set_patches(*patches*, *patch_centers*, *offset=None*, *offset_index=None*)

Set the values of a group of patches into the correct regions of a copy of this image. Given an array of patches and a set of patch centers, the patches' values are copied in the regions of the image that are centred on the coordinates of the given centers.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` *Image* objects

Currently only 2D images are supported.

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **patch_centers** (*PointCloud*) – The centers to set the patches around.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises

- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

set_patches_around_landmarks (*patches*, *group=None*, *offset=None*, *offset_index=None*)

Set the values of a group of patches around the landmarks existing in a copy of this image. Given an array of patches, a group and a label, the patches' values are copied in the regions of the image that are centred on the coordinates of corresponding landmarks.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` *Image* objects

Currently only 2D images are supported.

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **group** (*str* or `None` optional) – The landmark group to use as patch centres.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises

- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

view_widget (*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualizes the image object using an interactive widget. Currently only supports the rendering of 2D images.

Parameters

- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the images will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using

black and white colours.

warp_to_mask(*template_mask*, *transform*, *warp_landmarks=True*, *order=1*, *mode='constant'*, *cval=0.0*, *batch_size=None*, *return_transform=False*)

Return a copy of this image warped into a different reference space.

Note that warping into a mask is slower than warping into a full image. If you don't need a non-linear mask, consider :meth:`warp_to_shape` instead.

Parameters

- **template_mask** (`BooleanImage`) – Defines the shape of the result, and what pixels should be sampled.
- **transform** (`Transform`) – Transform **from the template space back to this image**. Defines, for each pixel location on the template, which pixel location should be sampled from on this image.
- **warp_landmarks** (*bool*, optional) – If `True`, result will have the same landmark dictionary as `self`, but with each landmark updated to the warped position.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **mode** (`{constant, nearest, reflect, wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If `None`, no batching is used and all points are warped at once.
- **return_transform** (*bool*, optional) – This argument is for internal use only. If `True`, then the `Transform` object is also returned.

Returns

- **warped_image** (`MaskedImage`) – A copy of this image, warped.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

warp_to_shape(*template_shape*, *transform*, *warp_landmarks=True*, *order=1*, *mode='constant'*, *cval=0.0*, *batch_size=None*, *return_transform=False*)

Return a copy of this image warped into a different reference space.

Parameters

- **template_shape** (*tuple* or *ndarray*) – Defines the shape of the result, and what pixel indices should be sampled (all of them).
- **transform** (`Transform`) – Transform **from the template_shape space back to this image**. Defines, for each index on template_shape, which pixel location should be sampled from on this image.
- **warp_landmarks** (*bool*, optional) – If `True`, result will have the same landmark dictionary as self, but with each landmark updated to the warped position.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **mode** ({constant, nearest, reflect, wrap}, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode constant, the value outside the image boundaries.
- **batch_size** (*int* or None, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If None, no batching is used and all points are warped at once.
- **return_transform** (*bool*, optional) – This argument is for internal use only. If True, then the Transform object is also returned.

Returns

- **warped_image** (*type(self)*) – A copy of this image, warped.
- **transform** (Transform) – The transform that was used. It only applies if *return_transform* is True.

zoom(*scale*, *cval=0.0*, *return_transform=False*)

Return a copy of this image, zoomed about the centre point. scale values greater than 1.0 denote zooming **in** to the image and values less than 1.0 denote zooming **out** of the image. The size of the image will not change, if you wish to scale an image, please see rescale().

Parameters

- **scale** (*float*) – scale > 1.0 denotes zooming in. Thus the image will appear larger and areas at the edge of the zoom will be 'cropped' out. scale < 1.0 denotes zooming out. The image will be padded by the value of cval.
- **cval** (float, optional) – The value to be set outside the rotated image boundaries.
- **return_transform** (*bool*, optional) – If True, then the Transform object that was used to perform the zooming is also returned.

Returns

- **zoomed_image** (type(self)) – A copy of this image, zoomed.
- **transform** (Transform) – The transform that was used. It only applies if *return_transform* is True.

has_landmarks

Whether the object has landmarks.

Type*bool*

height

The height of the image.

This is the height according to image semantics, and is thus the size of the **second to last** dimension.

Type*int*

landmarks

The landmarks object.

TypeLandmarkManager

n_channels

The number of channels on each pixel in the image.

Type*int*

n_dims
The number of dimensions in the image. The minimum possible `n_dims` is 2.
Type*int*

n_elements
Total number of data points in the image `(prod(shape), n_channels)`
Type*int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_pixels
Total number of pixels in the image `(prod(shape),)`
Type*int*

shape
The shape of the image (with `n_channel` values at each point).
Type*tuple*

width
The width of the image.

This is the width according to image semantics, and is thus the size of the **last** dimension.
Type*int*

BooleanImage

class `menpo.image.`**BooleanImage**(*mask_data*, *copy=True*)
Bases: `Image`

A mask image made from binary pixels. The region of the image that is left exposed by the mask is referred to as the 'masked region'. The set of 'masked' pixels is those pixels corresponding to a `True` value in the mask.

Parameters
- **mask_data** (`(M, N, ..., L)` *ndarray*) – The binary mask data. Note that there is no channel axis - a 2D Mask Image is built from just a 2D numpy array of mask_data. Automatically coerced in to boolean values.
- **copy** (*bool*, optional) – If `False`, the image_data will not be copied on assignment. Note that if the array you provide is not boolean, there **will still be copy**. In general this should only be used if you know what you are doing.

all_true()
`True` iff every element of the mask is `True`.
Type*bool*

as_PILImage(*out_dtype=<type 'numpy.uint8'>*)
Return a PIL copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.
Parameters**out_dtype** (*np.dtype*, optional) – The dtype the output array should be.
Returns**pil_image** (*PILImage*) – PIL copy of image
Raises
- `ValueError` – If image is not 2D and has 1 channel or 3 channels.

- `ValueError` – If pixels data type is *float32* or *float64* and the pixel range is outside of `[0, 1]`
- `ValueError` – If the output dtype is unsupported. Currently uint8 is supported.

as_greyscale(*mode='luminosity'*, *channel=None*)

Returns a greyscale version of the image. If the image does *not* represent a 2D RGB image, then the `luminosity` mode will fail.

Parameters

- **mode** (`{average, luminosity, channel}`, optional) –

mode	Greyscale Algorithm
average	Equal average of all channels
luminosity	Calculates the luminance using the CCIR 601 formula:
	$$Y' = 0.2989R' + 0.5870G' + 0.1140B'$$
channel	A specific channel is chosen as the intensity value.

- **channel** (*int*, optional) – The channel to be taken. Only used if mode is `channel`.

Returns **greyscale_image** (`MaskedImage`) – A copy of this image in greyscale.

as_histogram(*keep_channels=True*, *bins='unique'*)

Histogram binning of the values of this image.

Parameters

- **keep_channels** (*bool*, optional) – If set to `False`, it returns a single histogram for all the channels of the image. If set to `True`, it returns a *list* of histograms, one for each channel.
- **bins** (`{unique}`, positive *int* or sequence of scalars, optional) – If set equal to `'unique'`, the bins of the histograms are centred on the unique values of each channel. If set equal to a positive *int*, then this is the number of bins. If set equal to a sequence of scalars, these will be used as bins centres.

Returns

- **hist** (*ndarray* or *list* with `n_channels` *ndarrays* inside) – The histogram(s). If `keep_channels=False`, then hist is an *ndarray*. If `keep_channels=True`, then hist is a *list* with `len(hist)=n_channels`.
- **bin_edges** (*ndarray* or *list* with *n_channels ndarrays* inside) – An array or a list of arrays corresponding to the above histograms that store the bins' edges.

Raises `ValueError` – Bins can be either 'unique', positive int or a sequence of scalars.

Examples

Visualizing the histogram when a list of array bin edges is provided:

```
>>> hist, bin_edges = image.as_histogram()
>>> for k in range(len(hist)):
>>>     plt.subplot(1,len(hist),k)
>>>     width = 0.7 * (bin_edges[k][1] - bin_edges[k][0])
>>>     centre = (bin_edges[k][:-1] + bin_edges[k][1:]) / 2
>>>     plt.bar(centre, hist[k], align='center', width=width)
```

as_imageio(*out_dtype=<type 'numpy.uint8'>*)

Return an Imageio copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.

Parameters **out_dtype** (*np.dtype*, optional) – The dtype the output array should be.

Returns **imageio_image** (*ndarray*) – Imageio image (which is just a numpy ndarray with the channels as the last axis).

Raises

- `ValueError` – If image is not 2D and has 1 channel or 3 channels.
- `ValueError` – If pixels data type is *float32* or *float64* and the pixel range is outside of `[0, 1]`
- `ValueError` – If the output dtype is unsupported. Currently uint8 and uint16 are supported.

as_masked(*mask=None*, *copy=True*)

Impossible for a `BooleanImage` to be transformed to a `MaskedImage`.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounds()

The bounds of the image, minimum is always (0, 0). The maximum is the maximum **index** that can be used to index into the image for each dimension. Therefore, bounds will be of the form: ((0, 0), (self.height - 1, self.width - 1)) for a 2D image.

Note that this is akin to supporting a nearest neighbour interpolation. Although the *actual* maximum subpixel value would be something like `self.height - eps` where `eps` is some value arbitrarily close to 0, this value at least allows sampling without worrying about floating point error.

Type *tuple*

bounds_false(*boundary=0*, *constrain_to_bounds=True*)

Returns the minimum to maximum indices along all dimensions that the mask includes which fully surround the False mask values. In the case of a 2D Image for instance, the min and max define two corners of a rectangle bounding the False pixel values.

Parameters

- **boundary** (*int* >= 0, optional) – A number of pixels that should be added to the extent. A negative value can be used to shrink the bounds in.
- **constrain_to_bounds** (*bool*, optional) – If `True`, the bounding extent is snapped to not go beyond the edge of the image. If `False`, the bounds are left unchanged.

Returns

- **min_b** (`(D,)` *ndarray*) – The minimum extent of the `True` mask region with the boundary along each dimension. If `constrain_to_bounds=True`, is clipped to legal image bounds.
- **max_b** (`(D,)` *ndarray*) – The maximum extent of the `True` mask region with the boundary along each dimension. If `constrain_to_bounds=True`, is clipped to legal image bounds.

bounds_true(*boundary=0*, *constrain_to_bounds=True*)

Returns the minimum to maximum indices along all dimensions that the mask includes which fully surround the `True` mask values. In the case of a 2D Image for instance, the min and max define two corners of a rectangle bounding the True pixel values.

Parameters

- **boundary** (*int*, optional) – A number of pixels that should be added to the extent. A negative value can be used to shrink the bounds in.
- **constrain_to_bounds** (*bool*, optional) – If `True`, the bounding extent is snapped to not go beyond the edge of the image. If `False`, the bounds are left

unchanged.

Returns

- **min_b** (`(D,)` *ndarray*) – The minimum extent of the `True` mask region with the boundary along each dimension. If `constrain_to_bounds=True`, is clipped to legal image bounds.
- **max_b** (`(D,)` *ndarray*) – The maximum extent of the `True` mask region with the boundary along each dimension. If `constrain_to_bounds=True`, is clipped to legal image bounds.

centre()

The geometric centre of the Image - the subpixel that is in the middle.

Useful for aligning shapes and images.

Type(`n_dims`,) *ndarray*

constrain_landmarks_to_bounds()

Deprecated - please use the equivalent `constrain_to_bounds` method now on PointCloud, in conjunction with the new Image `bounds()` method. For example:

```
>>> im.constrain_landmarks_to_bounds()  # Equivalent to below
>>> im.landmarks['test'] = im.landmarks['test'].lms.constrain_to_bounds(im.bounds)
```

constrain_points_to_bounds(*points*)

Constrains the points provided to be within the bounds of this image.

Parameters **points** (`(d,)` *ndarray*) – Points to be snapped to the image boundaries.

Returns **bounded_points** (`(d,)` *ndarray*) – Points snapped to not stray outside the image edges.

constrain_to_landmarks(*group=None*, *batch_size=None*)

Returns a copy of this image whereby the `True` values in the image are restricted to be equal to the convex hull around the landmarks chosen. This is not a per-pixel convex hull, but instead relies on a triangulated approximation. If the landmarks in question are an instance of *TriMesh*, the triangulation of the landmarks will be used in the convex hull calculation. If the landmarks are an instance of *PointCloud*, Delaunay triangulation will be used to create a triangulation.

Parameters

- **group** (*str*, optional) – The key of the landmark set that should be used. If `None`, and if there is only one set of landmarks, this set will be used.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value will cause constraining to become much slower. This size indicates how many points in the image should be checked at a time, which keeps memory usage low. If `None`, no batching is used and all points are checked at once.

Returns **constrained** (*BooleanImage*) – The new boolean image, constrained by the given landmark group.

constrain_to_pointcloud(*pointcloud*, *batch_size=None*, *point_in_pointcloud='pwa'*)

Returns a copy of this image whereby the `True` values in the image are restricted to be equal to the convex hull around a pointcloud. The choice of whether a pixel is inside or outside of the pointcloud is determined by the `point_in_pointcloud` parameter. By default a Piecewise Affine transform is used to test for containment, which is useful when aligning images by their landmarks. Triangluation will be decided by Delauny - if you wish to customise it, a *TriMesh* instance can be passed for the `pointcloud` argument. In this case, the triangulation of the Trimesh will be used to define the retained region.

For large images, a faster and pixel-accurate method can be used ('convex_hull'). Here, there is no specialization for *TriMesh* instances. Alternatively, a callable can be provided to override the test. By default, the provided implementations are only valid for 2D images.

Parameters

- **pointcloud** (*PointCloud* or *TriMesh*) – The pointcloud of points that should be constrained to. See *point_in_pointcloud* for how in some cases a *TriMesh* may be used to control triangulation.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value will cause constraining to become much slower. This size indicates how many points in the image should be checked at a time, which keeps memory usage low. If `None`, no batching is used and all points are checked at once. By default, this is only used for the 'pwa' point_in_pointcloud choice.
- **point_in_pointcloud** ({'pwa', 'convex_hull'} or *callable*) – The method used to check if pixels in the image fall inside the `pointcloud` or not. If 'pwa', Menpo's *PiecewiseAffine* transform will be used to test for containment. In this case `pointcloud` should be a *TriMesh*. If it isn't, Delauny triangulation will be used to first triangulate `pointcloud` into a *TriMesh* before testing for containment. If a callable is passed, it should take two parameters, the *PointCloud* to constrain with and the pixel locations ((d, n_dims) ndarray) to test and should return a (d, 1) boolean ndarray of whether the pixels were inside (True) or outside (False) of the *PointCloud*.

Returns **constrained** (*BooleanImage*) – The new boolean image, constrained by the given pointcloud.

Raises

- `ValueError` – If the image is not 2D and a default implementation is chosen.
- `ValueError` – If the chosen `point_in_pointcloud` is unknown.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

crop(*min_indices*, *max_indices*, *constrain_to_boundary=False*, *return_transform=False*)

Return a cropped copy of this image using the given minimum and maximum indices. Landmarks are correctly adjusted so they maintain their position relative to the newly cropped image.

Parameters

- **min_indices** (`(n_dims,)` *ndarray*) – The minimum index over each dimension.
- **max_indices** (`(n_dims,)` *ndarray*) – The maximum index over each dimension.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **cropped_image** (*type(self)*) – A new instance of self, but cropped.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises

- `ValueError` – `min_indices` and `max_indices` both have to be of length `n_dims`. All `max_indices` must be greater than `min_indices`.

- *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks(*group=None*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a set of landmarks with an optional `n_pixel` boundary

Parameters

- **group** (*str*, optional) – The key of the landmark set that should be used. If `None` and if there is only one set of landmarks, this set will be used.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an :map'ImageBoundaryError' will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – A copy of this image cropped to its landmarks.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks_proportion(*boundary_proportion*, *group=None*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)

Crop this image to be bounded around a set of landmarks with a border proportional to the landmark spread or range.

Parameters

- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.
- **group** (*str*, optional) – The key of the landmark set that should be used. If `None` and if there is only one set of landmarks, this set will be used.
- **minimum** (*bool*, optional) – If `True` the specified proportion is relative to the minimum value of the landmarks' per-dimension range; if `False` w.r.t. the maximum value of the landmarks' per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If `True`, the crop will be snapped to not go beyond this images boundary. If `False`, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – This image, cropped to its landmarks with a border proportional to the landmark spread or range.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud(*pointcloud*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional `n_pixel` boundary.

Parameters

- **pointcloud** (*PointCloud*) – The pointcloud to crop around.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an :map‘ImageBoundaryError‘ will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – A copy of this image cropped to the bounds of the pointcloud.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud_proportion(*pointcloud*, *boundary_proportion*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional `n_pixel` boundary.

Parameters

- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.
- **pointcloud** (*PointCloud*) – The pointcloud to crop around.
- **minimum** (*bool*, optional) – If `True` the specified proportion is relative to the minimum value of the pointclouds’ per-dimension range; if `False` w.r.t. the maximum value of the pointclouds’ per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If `True`, the crop will be snapped to not go beyond this images boundary. If `False`, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – A copy of this image cropped to the border proportional to the pointcloud spread or range.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

diagonal()

The diagonal size of this image

Type *float*

extract_channels(*channels*)

A copy of this image with only the specified channels.

Parameters **channels** (*int* or *[int]*) – The channel index or *list* of channel indices to retain.

Returns **image** (*type(self)*) – A copy of this image with only the channels requested.

extract_patches(*patch_centers*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)

Extract a set of patches from an image. Given a set of patch centers and a patch size, patches are extracted from within the image, centred on the given coordinates. Sample offsets denote a set of offsets to extract from within a patch. This is very useful if you want to extract a dense set of features around a set of landmarks and simply sample the same grid of patches around the landmarks.

If sample offsets are used, to access the offsets for each patch you need to slice the resulting *list*. So for 2 offsets, the first centers offset patches would be `patches[:2]`.

Currently only 2D images are supported.

Parameters

- **`patch_centers`** (*`PointCloud`*) – The centers to extract patches around.
- **`patch_shape`** (`(1, n_dims)` *tuple* or *ndarray*, optional) – The size of the patch to extract
- **`sample_offsets`** (`(n_offsets, n_dims)` *ndarray* or `None`, optional) – The offsets to sample from within a patch. So `(0, 0)` is the centre of the patch (no offset) and `(1, 0)` would be sampling the patch from 1 pixel up the first axis away from the centre. If `None`, then no offsets are applied.
- **`as_single_array`** (*bool*, optional) – If `True`, an `(n_center, n_offset, n_channels, patch_shape)` *ndarray*, thus a single numpy array is returned containing each patch. If `False`, a *list* of `n_center * n_offset` *`Image`* objects is returned representing each patch.

Returns patches (*list* or *ndarray*) – Returns the extracted patches. Returns a list if `as_single_array=True` and an *ndarray* if `as_single_array=False`.

Raises `ValueError` – If image is not 2D

`extract_patches_around_landmarks`(*group=None*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)

Extract patches around landmarks existing on this image. Provided the group label and optionally the landmark label extract a set of patches.

See *extract_patches* for more information.

Currently only 2D images are supported.

Parameters

- **`group`** (*str* or `None`, optional) – The landmark group to use as patch centres.
- **`patch_shape`** (*tuple* or *ndarray*, optional) – The size of the patch to extract
- **`sample_offsets`** (`(n_offsets, n_dims)` *ndarray* or `None`, optional) – The offsets to sample from within a patch. So `(0, 0)` is the centre of the patch (no offset) and `(1, 0)` would be sampling the patch from 1 pixel up the first axis away from the centre. If `None`, then no offsets are applied.
- **`as_single_array`** (*bool*, optional) – If `True`, an `(n_center, n_offset, n_channels, patch_shape)` *ndarray*, thus a single numpy array is returned containing each patch. If `False`, a *list* of `n_center * n_offset` *`Image`* objects is returned representing each patch.

Returns patches (*list* or *ndarray*) – Returns the extracted patches. Returns a list if `as_single_array=True` and an *ndarray* if `as_single_array=False`.

Raises `ValueError` – If image is not 2D

`false_indices`()

The indices of pixels that are `Flase`.

Type `(n_dims, n_false)` *ndarray*

`from_vector`(*vector*, *copy=True*)

Takes a flattened vector and returns a new *`BooleanImage`* formed by reshaping the vector to the correct dimensions. Note that this is rebuilding a boolean image **itself** from boolean values. The mask is in no way interpreted in performing the operation, in contrast to *`MaskedImage`*, where only the masked region is used in *`from_vector()`* and :meth‘as_vector‘. Any image landmarks are transferred in the process.

Parameters

- **vector** ((n_pixels,) *bool ndarray*) – A flattened vector of all the pixels of a *BooleanImage*.
- **copy** (*bool*, optional) – If False, no copy of the vector will be taken.

Returns image (*BooleanImage*) – New BooleanImage of same shape as this image

Raises Warning – If copy=False cannot be honored.

from_vector_inplace (*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of this object

gaussian_pyramid (*n_levels=3*, *downscale=2*, *sigma=None*)

Return the gaussian pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters

- **n_levels** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **downscale** (*float*, optional) – Downscale factor.
- **sigma** (*float*, optional) – Sigma for gaussian filter. Default is downscale / 3. which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the gaussian distribution.

Yields image_pyramid (*generator*) – Generator yielding pyramid layers as *Image* objects.

has_landmarks_outside_bounds ()

Indicates whether there are landmarks located outside the image bounds.

Type *bool*

has_nan_values ()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns has_nan_values (*bool*) – If the vectorized object contains nan values.

indices ()

Return the indices of all pixels in this image.

Type (n_dims, n_pixels) ndarray

classmethod init_blank (*shape*, *fill=True*, *round='ceil'*, ***kwargs*)

Returns a blank *BooleanImage* of the requested shape

Parameters

- **shape** (*tuple* or *list*) – The shape of the image. Any floating point values are rounded according to the round kwarg.
- **fill** (*bool*, optional) – The mask value to be set everywhere.
- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.

Returns blank_image (*BooleanImage*) – A blank mask of the requested size

classmethod init_from_pointcloud (*pointcloud*, *group=None*, *boundary=0*, *constrain=True*, *fill=True*)

Create an Image that is big enough to contain the given pointcloud. The pointcloud will be translated to the origin and then translated according to its bounds in order to fit inside the new image. An optional boundary can be provided in order to increase the space around the boundary of the pointcloud. The boundary will be added to *all sides of the image* and so a boundary of 5 provides 10 pixels of boundary total for each dimension.

By default, the mask will be constrained to the convex hull of the provided pointcloud.

Parameters

- **pointcloud** (*PointCloud*) – Pointcloud to place inside the newly created image.
- **group** (*str*, optional) – If None, the pointcloud will only be used to create the image. If a *str* then the pointcloud will be attached as a landmark group to the image, with the given string as key.
- **boundary** (*float*) – A optional padding distance that is added to the pointcloud bounds. Default is 0, meaning the max/min of tightest possible containing image is returned.
- **fill** (*int*, optional) – The value to fill all pixels with.
- **constrain** (*bool*, optional) – If True, the True values will be image will be constrained to the convex hull of the provided pointcloud. If False, the mask will be the value of fill.

Returns **image** (*MaskedImage*) – A new image with the same size as the given pointcloud, optionally with the pointcloud attached as landmarks and the mask constrained to the convex hull of the pointcloud.

init_from_rolled_channels(*pixels*)

Create an Image from a set of pixels where the channels axis is on the last axis (the back). This is common in other frameworks, and therefore this method provides a convenient means of creating a menpo Image from such data. Note that a copy is always created due to the need to rearrange the data.

Parameters **pixels** (`(M, N ..., Q, C)` *ndarray*) – Array representing the image pixels, with the last axis being channels.

Returns **image** (*Image*) – A new image from the given pixels, with the FIRST axis as the channels.

invert()

Returns a copy of this boolean image, which is inverted.

Returns **inverted** (*BooleanImage*) – A copy of this boolean mask, where all True values are False and all False values are True.

mirror(*axis=1*, *return_transform=False*)

Return a copy of this image, mirrored/flipped about a certain axis.

Parameters

- **axis** (*int*, optional) – The axis about which to mirror the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the mirroring is also returned.

Returns

- **mirrored_image** (type(self)) – The mirrored image.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises

- ValueError – axis cannot be negative
- ValueError – axis={} but the image has {} dimensions

n_false()

The number of False values in the mask.

Type *int*

n_true()

The number of True values in the mask.

Type *int*

normalize_norm(*mode='all'*, ***kwargs*)

Returns a copy of this image normalized such that its pixel values have zero mean and its norm equals 1.

Parameters **mode** ({all, per_channel}, optional) – If all, the normalization is over all channels. If per_channel, each channel individually is mean centred and unit norm.

Returnsimage (`type(self)`) – A copy of this image, normalized.

`normalize_std`(*mode='all'*, ***kwargs*)

Returns a copy of this image normalized such that its pixel values have zero mean and unit variance.

Parametersmode (`{all, per_channel}`, optional) – If `all`, the normalization is over all channels. If `per_channel`, each channel individually is mean centred and normalized in variance.

Returnsimage (`type(self)`) – A copy of this image, normalized.

`pixels_range`()

The range of the pixel values (min and max pixel values).

Returnsmin_max (`(dtype, dtype)`) – The minimum and maximum value of the pixels array.

`proportion_false`()

The proportion of the mask which is `False`

Type*float*

`proportion_true`()

The proportion of the mask which is `True`.

Type*float*

`pyramid`(*n_levels=3*, *downscale=2*)

Return a rescaled pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters

- **`n_levels`** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **`downscale`** (*float*, optional) – Downscale factor.

Yieldsimage_pyramid (*generator*) – Generator yielding pyramid layers as `Image` objects.

`rasterize_landmarks`(*group=None*, *render_lines=True*, *line_style='-'*, *line_colour='b'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=1*, *marker_face_colour='b'*, *marker_edge_colour='b'*, *marker_edge_width=1*, *backend='matplotlib'*)

This method provides the ability to rasterize 2D landmarks onto the image. The returned image has the specified landmark groups rasterized onto the image - which is useful for things like creating result examples or rendering videos with annotations.

Since multiple landmark groups can be specified, all arguments can take lists of parameters that map to the provided groups list. Therefore, the parameters must be lists of the correct length or a single parameter to apply to every landmark group.

Multiple backends are provided, all with different strengths. The 'pillow' backend is very fast, but not very flexible. The *matplotlib* backend should be feature compatible with other Menpo rendering methods, but is much slower due to the overhead of creating a figure to render into.

Parameters

- **`group`** (*str* or *list* of *str*, optional) – The landmark group key, or a list of keys.
- **`render_lines`** (*bool*, optional) – If `True`, and the provided landmark group is a `PointDirectedGraph`, the edges are rendered.
- **`line_style`** (*str*, optional) – The style of the edge line. Not all backends support this argument.
- **`line_colour`** (*str* or *tuple*, optional) – A Matplotlib style colour or a backend dependant colour.
- **`line_width`** (*int*, optional) – The width of the line to rasterize.
- **`render_markers`** (*bool*, optional) – If `True`, render markers at the coordinates of each landmark.

- **marker_style** (*str*, optional) – A Matplotlib marker style. Not all backends support all marker styles.
- **marker_size** (*int*, optional) – The size of the marker - different backends use different scale spaces so consistent output may by difficult.
- **marker_face_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_width** (*int*, optional) – The width of the marker edge. Not all backends support this.
- **backend** (*{'matplotlib', 'pillow'}, optional*) – The backend to use.

Returns **rasterized_image** (*Image*) – The image with the landmarks rasterized directly into the pixels.

Raises

- ValueError – Only 2D images are supported.
- ValueError – Only RGB (3-channel) or Greyscale (1-channel) images are supported.

rescale(*scale*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled by a given factor. Landmarks are rescaled appropriately.

Parameters

- **scale** (*float* or *tuple* of *floats*) – The scale factor. If a tuple, the scale to apply to each dimension. If a single *float*, the scale will be applied uniformly across each dimension.
- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (type(self)) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises ValueError: – If less scales than dimensions are provided. If any scale is less than or equal to 0.

rescale_landmarks_to_diagonal_range(*diagonal_range*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the diagonal_range of the bounding box containing its landmarks matches the specified diagonal_range range.

Parameters

- **diagonal_range** ((n_dims,) *ndarray*) – The diagonal_range range that we want the landmarks of the returned image to have.
- **group** (*str*, optional) – The key of the landmark set that should be used. If None and if there is only one set of landmarks, this set will be used.

- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (type(self)) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

rescale_pixels(*minimum*, *maximum*, *per_channel=True*)

A copy of this image with pixels linearly rescaled to fit a range.

Note that the only pixels that will considered and rescaled are those that feature in the vectorized form of this image. If you want to use this routine on all the pixels in a *MaskedImage*, consider using *as_unmasked()* prior to this call.

Parameters

- **minimum** (*float*) – The minimal value of the rescaled pixels
- **maximum** (*float*) – The maximal value of the rescaled pixels
- **per_channel** (*boolean*, optional) – If True, each channel will be rescaled independently. If False, the scaling will be over all channels.

Returnsrescaled_image (type(self)) – A copy of this image with pixels linearly rescaled to fit in the range provided.

rescale_to_diagonal(*diagonal*, *round='ceil'*, *return_transform=False*)

Return a copy of this image, rescaled so that the it's diagonal is a new size.

Parameters

- **diagonal** (*int*) – The diagonal size of the new image.
- **round** ({ceil, floor, round}, optional) – Rounding function to be applied to floating point shapes.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (*type(self)*) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

rescale_to_pointcloud(*pointcloud*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the scale of a particular group of landmarks matches the scale of the passed reference pointcloud.

Parameters

- **pointcloud** (*PointCloud*) – The reference pointcloud to which the landmarks specified by group will be scaled to match.
- **group** (*str*, optional) – The key of the landmark set that should be used. If None, and if there is only one set of landmarks, this set will be used.
- **round** ({ceil, floor, round}, optional) – Rounding function to be ap-

plied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (`type(self)`) – A copy of this image, rescaled.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

resize(*shape*, *order=1*, *return_transform=False*)

Return a copy of this image, resized to a particular shape. All image information (landmarks, and mask in the case of `MaskedImage`) is resized appropriately.

Parameters

- **shape** (*tuple*) – The new shape to resize to.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the resize is also returned.

Returns

- **resized_image** (`type(self)`) – A copy of this image, resized.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

RaisesValueError: – If the number of dimensions of the new shape does not match the number of dimensions of the image.

rolled_channels()

Returns the pixels matrix, with the channels rolled to the back axis. This may be required for interacting with external code bases that require images to have channels as the last axis, rather than the menpo convention of channels as the first axis.

Returnsrolled_channels (*ndarray*) – Pixels with channels as the back (last) axis.

rotate_ccw_about_centre(*theta*, *degrees=True*, *retain_shape=False*, *cval=0.0*, *round='round'*, *order=1*, *return_transform=False*)

Return a copy of this image, rotated counter-clockwise about its centre.

Note that the *retain_shape* argument defines the shape of the rotated image. If `retain_shape=True`, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If `retain_shape=False`, then the returned image has the correct size so that the whole area of the current image is included.

Parameters

- **theta** (*float*) – The angle of rotation about the centre.
- **degrees** (*bool*, optional) – If `True`, *theta* is interpreted in degrees. If `False`, `theta` is interpreted as radians.
- **retain_shape** (*bool*, optional) – If `True`, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If `False`, then the returned image has the correct size so that the whole area of the current image is included.
- **cval** (*float*, optional) – The value to be set outside the rotated image boundaries.
- **round** (`{'ceil', 'floor', 'round'}`, optional) – Rounding function to be applied to floating point shapes. This is only used in case `retain_shape=True`.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range `[0,5]`. This is only used in case `retain_shape=True`.

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rotation is also returned.

Returns

- **rotated_image** (`type(self)`) – The rotated image.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

Raises`ValueError` – Image rotation is presently only supported on 2D images

sample(*points_to_sample*, *mode='constant'*, *cval=False*, ***kwargs*)

Sample this image at the given sub-pixel accurate points. The input PointCloud should have the same number of dimensions as the image e.g. a 2D PointCloud for a 2D multi-channel image. A numpy array will be returned the has the values for every given point across each channel of the image.

Parameters

- **points_to_sample** (`PointCloud`) – Array of points to sample from the image. Should be *(n_points, n_dims)*
- **mode** (`{constant, nearest, reflect, wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.

Returns**sampled_pixels** ((*n_points*, *n_channels*) *bool ndarray*) – The interpolated values taken across every channel of the image.

set_patches(*patches*, *patch_centers*, *offset=None*, *offset_index=None*)

Set the values of a group of patches into the correct regions in a copy of this image. Given an array of patches and a set of patch centers, the patches' values are copied in the regions of the image that are centred on the coordinates of the given centers.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` `Image` objects

Currently only 2D images are supported.

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *ex-*

tract_patches_around_landmarks() methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **patch_centers** (*PointCloud*) – The centers to set the patches around.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises
- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

Returns **new_image** (*BooleanImage*) – A new boolean image where the provided patch locations have been set to the provided values.

set_patches_around_landmarks(*patches*, *group=None*, *offset=None*, *offset_index=None*)

Set the values of a group of patches around the landmarks existing in a copy of this image. Given an array of patches, a group and a label, the patches' values are copied in the regions of the image that are centred on the coordinates of corresponding landmarks.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:
1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` *Image* objects

Currently only 2D images are supported.

Parameters
- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **group** (*str* or `None` optional) – The landmark group to use as patch centres.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises
- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

true_indices()

The indices of pixels that are `True`.

Type `(n_dims, n_true)` *ndarray*

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualizes the image object using an interactive widget. Currently only supports the rendering of 2D images.

Parameters
- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the images will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

warp_to_mask(*template_mask*, *transform*, *warp_landmarks=True*, *mode='constant'*, *cval=False*, *batch_size=None*, *return_transform=False*)

Return a copy of this `BooleanImage` warped into a different reference space.

Note that warping into a mask is slower than warping into a full image. If you don't need a non-linear mask, consider warp_to_shape instead.

Parameters

- **template_mask** (`BooleanImage`) – Defines the shape of the result, and what pixels should be sampled.
- **transform** (`Transform`) – Transform **from the template space back to this image**. Defines, for each pixel location on the template, which pixel location should be sampled from on this image.
- **warp_landmarks** (*bool*, optional) – If `True`, result will have the same landmark dictionary as self, but with each landmark updated to the warped position.
- **mode** (`{constant, nearest, reflect or wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If `None`, no batching is used and all points are warped at once.
- **return_transform** (*bool*, optional) – This argument is for internal use only. If `True`, then the `Transform` object is also returned.

Returns

- **warped_image** (`BooleanImage`) – A copy of this image, warped.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

warp_to_shape(*template_shape*, *transform*, *warp_landmarks=True*, *mode='constant'*, *cval=False*, *order=None*, *batch_size=None*, *return_transform=False*)

Return a copy of this `BooleanImage` warped into a different reference space.

Note that the order keyword argument is in fact ignored, as any order other than 0 makes no sense on a binary image. The keyword argument is present only for compatibility with the `Image` warp_to_shape API.

Parameters

- **template_shape** (`(n_dims, )` *tuple* or *ndarray*) – Defines the shape of the result, and what pixel indices should be sampled (all of them).
- **transform** (`Transform`) – Transform **from the template_shape space back to this image**. Defines, for each index on template_shape, which pixel location should be sampled from on this image.
- **warp_landmarks** (*bool*, optional) – If `True`, result will have the same landmark dictionary as self, but with each landmark updated to the warped position.
- **mode** (`{constant, nearest, reflect or wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If `None`, no batching is used and all points are warped at once.
- **return_transform** (*bool*, optional) – This argument is for internal use only. If `True`, then the `Transform` object is also returned.

Returns

- **warped_image** (*BooleanImage*) – A copy of this image, warped.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

zoom(*scale*, *cval=0.0*, *return_transform=False*)

Return a copy of this image, zoomed about the centre point. `scale` values greater than 1.0 denote zooming **in** to the image and values less than 1.0 denote zooming **out** of the image. The size of the image will not change, if you wish to scale an image, please see `rescale()`.

Parameters

- **scale** (*float*) – `scale > 1.0` denotes zooming in. Thus the image will appear larger and areas at the edge of the zoom will be 'cropped' out. `scale < 1.0` denotes zooming out. The image will be padded by the value of `cval`.
- **cval** (`float`, optional) – The value to be set outside the rotated image boundaries.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the zooming is also returned.

Returns

- **zoomed_image** (`type(self)`) – A copy of this image, zoomed.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

has_landmarks

Whether the object has landmarks.

Type*bool*

height

The height of the image.

This is the height according to image semantics, and is thus the size of the **second to last** dimension.

Type*int*

landmarks

The landmarks object.

Type*LandmarkManager*

mask

Returns the pixels of the mask with no channel axis. This is what should be used to mask any k-dimensional image.

Type`(M, N, ..., L)`, *bool ndarray*

n_channels

The number of channels on each pixel in the image.

Type*int*

n_dims

The number of dimensions in the image. The minimum possible `n_dims` is 2.

Type*int*

n_elements

Total number of data points in the image `(prod(shape), n_channels)`

Type*int*

n_landmark_groups

The number of landmark groups on this object.

Type*int*

n_parameters

The length of the vector that this object produces.

Type*int*

n_pixels

Total number of pixels in the image (prod(shape),)

Type*int*

shape

The shape of the image (with n_channel values at each point).

Type*tuple*

width

The width of the image.

This is the width according to image semantics, and is thus the size of the **last** dimension.

Type*int*

MaskedImage

class menpo.image.**MaskedImage**(*image_data*, *mask=None*, *copy=True*)

Bases: Image

Represents an *n*-dimensional *k*-channel image, which has a mask. Images can be masked in order to identify a region of interest. All images implicitly have a mask that is defined as the the entire image. The mask is an instance of *BooleanImage*.

Parameters

- **image_data** ((C, M, N ..., Q) *ndarray*) – The pixel data for the image, where the first axis represents the number of channels.
- **mask** ((M, N) *bool ndarray* or *BooleanImage*, optional) – A binary array representing the mask. Must be the same shape as the image. Only one mask is supported for an image (so the mask is applied to every channel equally).
- **copy** (*bool*, optional) – If False, the image_data will not be copied on assignment. If a mask is provided, this also won't be copied. In general this should only be used if you know what you are doing.

RaisesValueError – Mask is not the same shape as the image

_view_2d(*figure_id=None*, *new_figure=False*, *channels=None*, *masked=True*, *interpolation='bilinear'*, *cmap_name=None*, *alpha=1.0*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

View the image using the default image viewer. This method will appear on the Image as view if the Image is 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **channels** (*int* or *list* of *int* or all or None) – If *int* or *list* of *int*, the specified channel(s) will be rendered. If all, all the channels will be rendered in subplots. If None and the image is RGB, it will be rendered in RGB mode. If None and the image is not RGB, it is equivalent to all.
- **masked** (*bool*, optional) – If True, only the masked pixels will be rendered.
- **interpolation** (*See Below, optional*) – The interpolation used to render the image. For example, if bilinear, the image will be smooth and if nearest, the image will be pixelated. Example options

```
{none, nearest, bilinear, bicubic, spline16, spline36,
hanning, hamming, hermite, kaiser, quadric, catrom, gaussian,
bessel, mitchell, sinc, lanczos}
```

- **cmap_name** (*str*, optional,) – If `None`, single channel and three channel images default to greyscale and rgb colormaps respectively.
- **alpha** (*float*, optional) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*,*float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the Image as a percentage of the Image's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the Image as a percentage of the Image's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.

Raises `ValueError` – If Image is not 2D

`_view_landmarks_2d`(*channels=None, masked=True, group=None, with_labels=None, without_labels=None, figure_id=None, new_figure=False, interpolation='bilinear', cmap_name=None, alpha=1.0, render_lines=True, line_colour=None, line_style='-', line_width=1, render_markers=True, marker_style='o', marker_size=5, marker_face_colour=None, marker_edge_colour=None, marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_legend=False, legend_title='', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', legend_marker_scale=None, legend_location=2, legend_bbox_to_anchor=(1.05, 1.0), legend_border_axes_pad=None, legend_n_columns=1, legend_horizontal_spacing=None, legend_vertical_spacing=None, legend_border=True, legend_border_padding=None, legend_shadow=False, legend_rounded_corners=False, render_axes=False, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **channels** (*int* or *list* of *int* or all or None) – If *int* or *list* of *int*, the specified channel(s) will be rendered. If all, all the channels will be rendered in subplots. If None and the image is RGB, it will be rendered in RGB mode. If None and the image is not RGB, it is equivalent to all.
- **masked** (*bool*, optional) – If True, only the masked pixels will be rendered.
- **group** (*str* or``None`` optionals) – The landmark group to be visualized. If None and there are more than one landmark groups, an error is raised.
- **with_labels** (None or *str* or *list* of *str*, optional) – If not None, only show the given label(s). Should **not** be used with the without_labels kwarg.
- **without_labels** (None or *str* or *list* of *str*, optional) – If not None, show all except the given label(s). Should **not** be used with the with_labels kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **interpolation** (See Below, optional) – The interpolation used to render the image. For example, if bilinear, the image will be smooth and if nearest, the image will be pixelated. Example options

```
{none, nearest, bilinear, bicubic, spline16, spline36, hanning,
hamming, hermite, kaiser, quadric, catrom, gaussian, bessel,
mitchell, sinc, lanczos}
```

- **cmap_name** (*str*, optional,) – If None, single channel and three channel images default to greyscale and rgb colormaps respectively.
- **alpha** (*float*, optional) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (See Below, optional) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (See Below, optional) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (See Below, optional) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (See Below, optional) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.

- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (`See Below, optional`) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (`See Below, optional`) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (`See Below, optional`) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (`See below, optional`) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (`See Below, optional`) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

‘best’	0
‘upper right’	1
‘upper left’	2
‘lower left’	3
‘lower right’	4
‘right’	5
‘center left’	6
‘center right’	7
‘lower center’	8
‘upper center’	9
‘center’	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend’s columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame’s corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the Image as a percentage of the Image’s width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the Image as a percentage of the Image’s height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None` optional) – The size of the figure in inches.

Raises

- `ValueError` – If both `with_labels` and `without_labels` are passed.
- `ValueError` – If the landmark manager doesn't contain the provided group label.

as_PILImage(*out_dtype=<type 'numpy.uint8'>*)

Return a PIL copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.

Parameters **out_dtype** (*np.dtype*, optional) – The dtype the output array should be.

Returns **pil_image** (*PILImage*) – PIL copy of image

Raises

- `ValueError` – If image is not 2D and has 1 channel or 3 channels.
- `ValueError` – If pixels data type is *float32* or *float64* and the pixel range is outside of `[0, 1]`
- `ValueError` – If the output dtype is unsupported. Currently uint8 is supported.

as_greyscale(*mode='luminosity'*, *channel=None*)

Returns a greyscale version of the image. If the image does *not* represent a 2D RGB image, then the `luminosity` mode will fail.

Parameters

- **mode** (`{average, luminosity, channel}`, optional) –

mode	Greyscale Algorithm
average	Equal average of all channels
luminosity	Calculates the luminance using the CCIR 601 formula:
	$$Y' = 0.2989R' + 0.5870G' + 0.1140B'$$
channel	A specific channel is chosen as the intensity value.

- **channel** (*int*, optional) – The channel to be taken. Only used if mode is `channel`.

Returns **greyscale_image** (`MaskedImage`) – A copy of this image in greyscale.

as_histogram(*keep_channels=True*, *bins='unique'*)

Histogram binning of the values of this image.

Parameters

- **keep_channels** (*bool*, optional) – If set to `False`, it returns a single histogram for all the channels of the image. If set to `True`, it returns a *list* of histograms, one for each channel.
- **bins** (`{unique}`, positive *int* or sequence of scalars, optional) – If set equal to `'unique'`, the bins of the histograms are centred on the unique values of each channel. If set equal to a positive *int*, then this is the number of bins. If set equal to a sequence of scalars, these will be used as bins centres.

Returns

- **hist** (*ndarray* or *list* with `n_channels` *ndarrays* inside) – The histogram(s). If `keep_channels=False`, then hist is an *ndarray*. If `keep_channels=True`, then hist is a *list* with `len(hist)=n_channels`.
- **bin_edges** (*ndarray* or *list* with *n_channels ndarrays* inside) – An array or a list of arrays corresponding to the above histograms that store the bins' edges.

Raises `ValueError` – Bins can be either 'unique', positive int or a sequence of scalars.

Examples

Visualizing the histogram when a list of array bin edges is provided:

```
>>> hist, bin_edges = image.as_histogram()
>>> for k in range(len(hist)):
>>>     plt.subplot(1,len(hist),k)
>>>     width = 0.7 * (bin_edges[k][1] - bin_edges[k][0])
>>>     centre = (bin_edges[k][:-1] + bin_edges[k][1:]) / 2
>>>     plt.bar(centre, hist[k], align='center', width=width)
```

as_imageio(*out_dtype=<type 'numpy.uint8'>*)

Return an Imageio copy of the image scaled and cast to the correct values for the provided `out_dtype`.

Image must only have 1 or 3 channels and be 2 dimensional. Non *uint8* floating point images must be in the range `[0, 1]` to be converted.

Parameters **out_dtype** (*np.dtype*, optional) – The dtype the output array should be.

Returns **imageio_image** (*ndarray*) – Imageio image (which is just a numpy ndarray with the channels as the last axis).

Raises

- `ValueError` – If image is not 2D and has 1 channel or 3 channels.
- `ValueError` – If pixels data type is *float32* or *float64* and the pixel range is outside of `[0, 1]`
- `ValueError` – If the output dtype is unsupported. Currently uint8 and uint16 are supported.

as_masked(*mask=None*, *copy=True*)

Return a copy of this image with an attached mask behavior.

A custom mask may be provided, or `None`. See the `MaskedImage` constructor for details of how the kwargs will be handled.

Parameters

- **mask** (`(self.shape)` *ndarray* or `BooleanImage`) – A mask to attach to the newly generated masked image.
- **copy** (*bool*, optional) – If `False`, the produced `MaskedImage` will share pixels with `self`. Only suggested to be used for performance.

Returns **masked_image** (`MaskedImage`) – An image with the same pixels and landmarks as this one, but with a mask.

as_unmasked(*copy=True*, *fill=None*)

Return a copy of this image without the masking behavior.

By default the mask is simply discarded. However, there is an optional kwarg, `fill`, that can be set which will fill the **non-masked** areas with the given value.

Parameters

- **copy** (*bool*, optional) – If `False`, the produced `Image` will share pixels with `self`. Only suggested to be used for performance.
- **fill** (*float* or `(n_channels,)` iterable or `None`, optional) – If `None` the mask is simply discarded. If a scalar or iterable, the *unmasked* regions are filled with the given value.

Returns **image** (`Image`) – An image with the same pixels and landmarks as this one, but with no mask.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounds()

The bounds of the image, minimum is always (0, 0). The maximum is the maximum **index** that can be used to index into the image for each dimension. Therefore, bounds will be of the form: ((0, 0), (self.height - 1, self.width - 1)) for a 2D image.

Note that this is akin to supporting a nearest neighbour interpolation. Although the *actual* maximum subpixel value would be something like `self.height - eps` where `eps` is some value arbitrarily close to 0, this value at least allows sampling without worrying about floating point error.

Type*tuple*

build_mask_around_landmarks(*patch_shape*, *group=None*)

Deprecated - please use the equivalent *constrain_mask_to_patches_around_landmarks* method.

centre()

The geometric centre of the Image - the subpixel that is in the middle.

Useful for aligning shapes and images.

Type(`n_dims`,) *ndarray*

constrain_landmarks_to_bounds()

Deprecated - please use the equivalent `constrain_to_bounds` method now on PointCloud, in conjunction with the new Image `bounds()` method. For example:

```
>>> im.constrain_landmarks_to_bounds()  # Equivalent to below
>>> im.landmarks['test'] = im.landmarks['test'].lms.constrain_to_bounds(im.bounds)
```

constrain_mask_to_landmarks(*group=None*, *batch_size=None*, *point_in_pointcloud='pwa'*)

Returns a copy of this image whereby the mask is restricted to be equal to the convex hull around the chosen landmarks.

The choice of whether a pixel is inside or outside of the pointcloud is determined by the `point_in_pointcloud` parameter. By default a Piecewise Affine transform is used to test for containment, which is useful when building efficiently aligning images. For large images, a faster and pixel-accurate method can be used ('convex_hull'). Alternatively, a callable can be provided to override the test. By default, the provided implementations are only valid for 2D images.

Parameters

- **group** (*str*, optional) – The key of the landmark set that should be used. If `None`, and if there is only one set of landmarks, this set will be used. If the landmarks in question are an instance of *TriMesh*, the triangulation of the landmarks will be used in the convex hull calculation. If the landmarks are an instance of *PointCloud*, Delaunay triangulation will be used to create a triangulation.
- **batch_size** (*int* or `None`, optional) – This should only be considered for large images. Setting this value will cause constraining to become much slower. This size indicates how many points in the image should be checked at a time, which keeps memory usage low. If `None`, no batching is used and all points are checked at once. By default, this is only used for the 'pwa' point_in_pointcloud choice.
- **point_in_pointcloud** ({'pwa', 'convex_hull'} or *callable*) – The method used to check if pixels in the image fall inside the pointcloud or not. Can be accurate to a Piecewise Affine transform, a pixel accurate convex hull or any arbitrary callable. If a callable is passed, it should take two parameters, the *PointCloud* to constrain with and the pixel locations ((d, n_dims) ndarray) to test and should return a (d, 1) boolean ndarray of whether the pixels were inside (True) or outside (False) of the *PointCloud*.

Returnsconstrained (*MaskedImage*) – A new image where the mask is constrained by the provided landmarks.

constrain_mask_to_patches_around_landmarks (*patch_shape*, *group=None*)

Returns a copy of this image whereby the mask is restricted to be patches around each landmark in the chosen landmark group. The patch will be centred on the nearest pixel for each point in the chosen landmark group.

Parameters

- **patch_shape** (*tuple*) – The size of the patch.
- **group** (*str*, optional) – The key of the landmark set that should be used. If `None`, and if there is only one set of landmarks, this set will be used.

Returnsconstrained (*MaskedImage*) – A new image where the mask is constrained as patches centred on each point in the provided landmarks.

constrain_points_to_bounds (*points*)

Constrains the points provided to be within the bounds of this image.

Parameterspoints (`(d,)` *ndarray*) – Points to be snapped to the image boundaries.

Returnsbounded_points (`(d,)` *ndarray*) – Points snapped to not stray outside the image edges.

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns`type(self)` – A copy of this object

crop (*min_indices*, *max_indices*, *constrain_to_boundary=False*, *return_transform=False*)

Return a cropped copy of this image using the given minimum and maximum indices. Landmarks are correctly adjusted so they maintain their position relative to the newly cropped image.

Parameters

- **min_indices** (`(n_dims,)` *ndarray*) – The minimum index over each dimension.
- **max_indices** (`(n_dims,)` *ndarray*) – The maximum index over each dimension.
- **constrain_to_boundary** (*bool*, optional) – If `True` the crop will be snapped to not go beyond this images boundary. If `False`, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **cropped_image** (*type(self)*) – A new instance of self, but cropped.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises

- `ValueError` – `min_indices` and `max_indices` both have to be of length `n_dims`. All `max_indices` must be greater than `min_indices`.
- *ImageBoundaryError* – Raised if `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks (*group=None*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a set of landmarks with an optional `n_pixel` boundary

Parameters

- **group** (*str*, optional) – The key of the landmark set that should be used. If None and if there is only one set of landmarks, this set will be used.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If True the crop will be snapped to not go beyond this images boundary. If False, an :map'ImageBoundaryError' will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – A copy of this image cropped to its landmarks.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises *ImageBoundaryError* – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_landmarks_proportion(*boundary_proportion*, *group=None*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)

Crop this image to be bounded around a set of landmarks with a border proportional to the landmark spread or range.

Parameters

- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.
- **group** (*str*, optional) – The key of the landmark set that should be used. If None and if there is only one set of landmarks, this set will be used.
- **minimum** (*bool*, optional) – If True the specified proportion is relative to the minimum value of the landmarks' per-dimension range; if False w.r.t. the maximum value of the landmarks' per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If True, the crop will be snapped to not go beyond this images boundary. If False, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the cropping is also returned.

Returns

- **image** (*Image*) – This image, cropped to its landmarks with a border proportional to the landmark spread or range.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises *ImageBoundaryError* – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud(*pointcloud*, *boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional n_pixel boundary.

Parameters

- **pointcloud** (*PointCloud*) – The pointcloud to crop around.
- **boundary** (*int*, optional) – An extra padding to be added all around the landmarks bounds.
- **constrain_to_boundary** (*bool*, optional) – If True the crop will be snapped to not go beyond this images boundary. If False, an

:map'ImageBoundaryError' will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the cropping is also returned.

Returns
- **image** (*Image*) – A copy of this image cropped to the bounds of the pointcloud.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises *ImageBoundaryError* – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_pointcloud_proportion(*pointcloud*, *boundary_proportion*, *minimum=True*, *constrain_to_boundary=True*, *return_transform=False*)

Return a copy of this image cropped so that it is bounded around a pointcloud with an optional n_pixel boundary.

Parameters
- **boundary_proportion** (*float*) – Additional padding to be added all around the landmarks bounds defined as a proportion of the landmarks range. See the minimum parameter for a definition of how the range is calculated.
- **pointcloud** (*PointCloud*) – The pointcloud to crop around.
- **minimum** (*bool*, optional) – If True the specified proportion is relative to the minimum value of the pointclouds' per-dimension range; if False w.r.t. the maximum value of the pointclouds' per-dimension range.
- **constrain_to_boundary** (*bool*, optional) – If True, the crop will be snapped to not go beyond this images boundary. If False, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the cropping is also returned.

Returns
- **image** (*Image*) – A copy of this image cropped to the border proportional to the pointcloud spread or range.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises *ImageBoundaryError* – Raised if constrain_to_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.

crop_to_true_mask(*boundary=0*, *constrain_to_boundary=True*, *return_transform=False*)

Crop this image to be bounded just the *True* values of it's mask.

Parameters
- **boundary** (*int*, optional) – An extra padding to be added all around the true mask region.
- **constrain_to_boundary** (*bool*, optional) – If True the crop will be snapped to not go beyond this images boundary. If False, an *ImageBoundaryError* will be raised if an attempt is made to go beyond the edge of the image. Note that is only possible if boundary != 0.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the cropping is also returned.

Returns
- **cropped_image** (type(self)) – A copy of this image, cropped to the true mask.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

Raises *ImageBoundaryError* – Raised if 11constrain_to_boundary=False'1, and an attempt is made to crop the image in a way that violates the image bounds.

diagonal()
The diagonal size of this image

Type*float*

dilate(*n_pixels=1*)
Returns a copy of this *MaskedImage* in which its mask has been expanded by n pixels along its boundary.

Parameters **n_pixels** (*int*, optional) – The number of pixels by which we want to expand the mask along its own boundary.

Returns **dilated_image** (*MaskedImage*) – The copy of the masked image in which the mask has been expanded by n pixels along its boundary.

erode(*n_pixels=1*)
Returns a copy of this *MaskedImage* in which the mask has been shrunk by n pixels along its boundary.

Parameters **n_pixels** (*int*, optional) – The number of pixels by which we want to shrink the mask along its own boundary.

Returns **eroded_image** (*MaskedImage*) – The copy of the masked image in which the mask has been shrunk by n pixels along its boundary.

extract_channels(*channels*)
A copy of this image with only the specified channels.

Parameters **channels** (*int* or *[int]*) – The channel index or *list* of channel indices to retain.

Returns **image** (*type(self)*) – A copy of this image with only the channels requested.

extract_patches(*patch_centers*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)
Extract a set of patches from an image. Given a set of patch centers and a patch size, patches are extracted from within the image, centred on the given coordinates. Sample offsets denote a set of offsets to extract from within a patch. This is very useful if you want to extract a dense set of features around a set of landmarks and simply sample the same grid of patches around the landmarks.

If sample offsets are used, to access the offsets for each patch you need to slice the resulting *list*. So for 2 offsets, the first centers offset patches would be `patches[:2]`.

Currently only 2D images are supported.

Parameters

- **patch_centers** (*PointCloud*) – The centers to extract patches around.
- **patch_shape** (`(1, n_dims)` *tuple* or *ndarray*, optional) – The size of the patch to extract
- **sample_offsets** (`(n_offsets, n_dims)` *ndarray* or `None`, optional) – The offsets to sample from within a patch. So `(0, 0)` is the centre of the patch (no offset) and `(1, 0)` would be sampling the patch from 1 pixel up the first axis away from the centre. If `None`, then no offsets are applied.
- **as_single_array** (*bool*, optional) – If `True`, an `(n_center, n_offset, n_channels, patch_shape)` *ndarray*, thus a single numpy array is returned containing each patch. If `False`, a *list* of `n_center * n_offset` *Image* objects is returned representing each patch.

Returns **patches** (*list* or *ndarray*) – Returns the extracted patches. Returns a list if `as_single_array=True` and an *ndarray* if `as_single_array=False`.

Raises `ValueError` – If image is not 2D

extract_patches_around_landmarks(*group=None*, *patch_shape=(16, 16)*, *sample_offsets=None*, *as_single_array=True*)
Extract patches around landmarks existing on this image. Provided the group label and optionally the landmark label extract a set of patches.

See *extract_patches* for more information.

Currently only 2D images are supported.

Parameters

- **group** (*str* or `None`, optional) – The landmark group to use as patch centres.
- **patch_shape** (*tuple* or *ndarray*, optional) – The size of the patch to extract
- **sample_offsets** (`(n_offsets, n_dims)` *ndarray* or `None`, optional) – The offsets to sample from within a patch. So `(0, 0)` is the centre of the patch (no offset) and `(1, 0)` would be sampling the patch from 1 pixel up the first axis away from the centre. If `None`, then no offsets are applied.
- **as_single_array** (*bool*, optional) – If `True`, an `(n_center, n_offset, n_channels, patch_shape)` *ndarray*, thus a single numpy array is returned containing each patch. If `False`, a *list* of `n_center * n_offset` *Image* objects is returned representing each patch.

Returnspatches (*list* or *ndarray*) – Returns the extracted patches. Returns a list if `as_single_array=True` and an *ndarray* if `as_single_array=False`.

Raises`ValueError` – If image is not 2D

from_vector(*vector*, *n_channels=None*)

Takes a flattened vector and returns a new image formed by reshaping the vector to the correct pixels and channels. Note that the only region of the image that will be filled is the masked region.

On masked images, the vector is always copied.

The `n_channels` argument is useful for when we want to add an extra channel to an image but maintain the shape. For example, when calculating the gradient.

Note that landmarks are transferred in the process.

Parameters

- **vector** (`(n_pixels,)`) – A flattened vector of all pixels and channels of an image.
- **n_channels** (*int*, optional) – If given, will assume that vector is the same shape as this image, but with a possibly different number of channels.

Returnsimage (*MaskedImage*) – New image of same shape as this image and the number of specified channels.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of this object

gaussian_pyramid(*n_levels=3*, *downscale=2*, *sigma=None*)

Return the gaussian pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters

- **n_levels** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **downscale** (*float*, optional) – Downscale factor.
- **sigma** (*float*, optional) – Sigma for gaussian filter. Default is `downscale / 3.` which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the gaussian distribution.

Yieldsimage_pyramid (*generator*) – Generator yielding pyramid layers as *Image* objects.

has_landmarks_outside_bounds()

Indicates whether there are landmarks located outside the image bounds.

Type*bool*

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns has_nan_values (*bool*) – If the vectorized object contains `nan` values.

indices()

Return the indices of all true pixels in this image.

Type (n_dims, n_true_pixels) *ndarray*

classmethod init_blank(*shape*, *n_channels=1*, *fill=0*, *dtype=<type 'float'>*, *mask=None*)

Generate a blank masked image

Parameters

- **shape** (*tuple* or *list*) – The shape of the image. Any floating point values are rounded up to the nearest integer.
- **n_channels** (*int*, optional) – The number of channels to create the image with.
- **fill** (*int*, optional) – The value to fill all pixels with.
- **dtype** (*numpy datatype*, optional) – The datatype of the image.
- **mask** ((M, N) *bool ndarray* or *BooleanImage*) – An optional mask that can be applied to the image. Has to have a shape equal to that of the image.

Notes

Subclasses of *MaskedImage* need to overwrite this method and explicitly call this superclass method

```
super(SubClass, cls).init_blank(shape,**kwargs)
```

in order to appropriately propagate the subclass type to `cls`.

Returns blank_image (*MaskedImage*) – A new masked image of the requested size.

classmethod init_from_pointcloud(*pointcloud*, *group=None*, *boundary=0*, *constrain_mask=True*, *n_channels=1*, *fill=0*, *dtype=<type 'float'>*)

Create an Image that is big enough to contain the given pointcloud. The pointcloud will be translated to the origin and then translated according to its bounds in order to fit inside the new image. An optional boundary can be provided in order to increase the space around the boundary of the pointcloud. The boundary will be added to *all sides of the image* and so a boundary of 5 provides 10 pixels of boundary total for each dimension.

By default, the mask will be constrained to the convex hull of the provided pointcloud.

Parameters

- **pointcloud** (*PointCloud*) – Pointcloud to place inside the newly created image.
- **group** (*str*, optional) – If `None`, the pointcloud will only be used to create the image. If a *str* then the pointcloud will be attached as a landmark group to the image, with the given string as key.
- **boundary** (*float*) – A optional padding distance that is added to the pointcloud bounds. Default is `0`, meaning the max/min of tightest possible containing image is returned.
- **n_channels** (*int*, optional) – The number of channels to create the image with.
- **fill** (*int*, optional) – The value to fill all pixels with.
- **dtype** (*numpy data type, optional*) – The data type of the image.
- **constrain_mask** (*bool*, optional) – If `True`, the mask will be constrained to the convex hull of the provided pointcloud. If `False`, the mask will be all `True`.

Returns image (*MaskedImage*) – A new image with the same size as the given pointcloud, optionally with the pointcloud attached as landmarks and the mask constrained to the convex hull of the pointcloud.

classmethod **init_from_rolled_channels**(*pixels*, *mask=None*)
Create an Image from a set of pixels where the channels axis is on the last axis (the back). This is common in other frameworks, and therefore this method provides a convenient means of creating a menpo Image from such data. Note that a copy is always created due to the need to rearrange the data.

Parameters

- **pixels** (`(M, N ..., Q, C)` *ndarray*) – Array representing the image pixels, with the last axis being channels.
- **mask** (`(M, N)` *bool ndarray* or *BooleanImage*, optional) – A binary array representing the mask. Must be the same shape as the image. Only one mask is supported for an image (so the mask is applied to every channel equally).

Returns **image** (*Image*) – A new image from the given pixels, with the FIRST axis as the channels.

masked_pixels()
Get the pixels covered by the *True* values in the mask.

Type `(n_channels, mask.n_true)` *ndarray*

mirror(*axis=1*, *return_transform=False*)
Return a copy of this image, mirrored/flipped about a certain axis.

Parameters

- **axis** (*int*, optional) – The axis about which to mirror the image.
- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the mirroring is also returned.

Returns

- **mirrored_image** (`type(self)`) – The mirrored image.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises

- `ValueError` – axis cannot be negative
- `ValueError` – axis={} but the image has {} dimensions

n_false_elements()
The number of `False` elements of the image over all the channels.

Type *int*

n_false_pixels()
The number of `False` values in the mask.

Type *int*

n_true_elements()
The number of `True` elements of the image over all the channels.

Type *int*

n_true_pixels()
The number of `True` values in the mask.

Type *int*

normalize_norm(*mode='all'*, *limit_to_mask=True*, ***kwargs*)
Returns a copy of this image normalized such that it's pixel values have zero mean and its norm equals 1.

Parameters

- **mode** (`{all, per_channel}`, optional) – If `all`, the normalization is over all channels. If `per_channel`, each channel individually is mean centred and normalized in variance.
- **limit_to_mask** (*bool*, optional) – If `True`, the normalization is only performed wrt the masked pixels. If `False`, the normalization is wrt all pixels, regardless of their masking value.

Returns **image** (`type(self)`) – A copy of this image, normalized.

normalize_std(*mode='all'*, *limit_to_mask=True*)
Returns a copy of this image normalized such that it's pixel values have zero mean and unit variance.

Parameters
- **mode** ({all, per_channel}, optional) – If all, the normalization is over all channels. If per_channel, each channel individually is mean centred and normalized in variance.
- **limit_to_mask** (*bool*, optional) – If True, the normalization is only performed wrt the masked pixels. If False, the normalization is wrt all pixels, regardless of their masking value.

Returnsimage (type(self)) – A copy of this image, normalized.

pixels_range()
The range of the pixel values (min and max pixel values).

Returnsmin_max ((dtype, dtype)) – The minimum and maximum value of the pixels array.

pyramid(*n_levels=3*, *downscale=2*)
Return a rescaled pyramid of this image. The first image of the pyramid will be a copy of the original, unmodified, image, and counts as level 1.

Parameters
- **n_levels** (*int*, optional) – Total number of levels in the pyramid, including the original unmodified image
- **downscale** (*float*, optional) – Downscale factor.

Yieldsimage_pyramid (*generator*) – Generator yielding pyramid layers as *Image* objects.

rasterize_landmarks(*group=None*, *render_lines=True*, *line_style='-'*, *line_colour='b'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=1*, *marker_face_colour='b'*, *marker_edge_colour='b'*, *marker_edge_width=1*, *backend='matplotlib'*)
This method provides the ability to rasterize 2D landmarks onto the image. The returned image has the specified landmark groups rasterized onto the image - which is useful for things like creating result examples or rendering videos with annotations.

Since multiple landmark groups can be specified, all arguments can take lists of parameters that map to the provided groups list. Therefore, the parameters must be lists of the correct length or a single parameter to apply to every landmark group.

Multiple backends are provided, all with different strengths. The 'pillow' backend is very fast, but not very flexible. The *matplotlib* backend should be feature compatible with other Menpo rendering methods, but is much slower due to the overhead of creating a figure to render into.

Images will always be rendered masked with a black background. If an unmasked image is required, please use *as_unmasked()*.

Parameters
- **group** (*str* or *list* of *str*, optional) – The landmark group key, or a list of keys.
- **render_lines** (*bool*, optional) – If True, and the provided landmark group is a *PointDirectedGraph*, the edges are rendered.
- **line_style** (*str*, optional) – The style of the edge line. Not all backends support this argument.
- **line_colour** (*str* or *tuple*, optional) – A Matplotlib style colour or a backend dependant colour.
- **line_width** (*int*, optional) – The width of the line to rasterize.
- **render_markers** (*bool*, optional) – If True, render markers at the coordinates of each landmark.
- **marker_style** (*str*, optional) – A Matplotlib marker style. Not all backends support all marker styles.
- **marker_size** (*int*, optional) – The size of the marker - different backends use

different scale spaces so consistent output may by difficult.

- **marker_face_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_colour** (*str*, optional) – A Matplotlib style colour or a backend dependant colour.
- **marker_edge_width** (*int*, optional) – The width of the marker edge. Not all backends support this.
- **backend** (*{'matplotlib', 'pillow'}, optional*) – The backend to use.

Returns rasterized_image (*Image*) – The image with the landmarks rasterized directly into the pixels.

Raises

- `ValueError` – Only 2D images are supported.
- `ValueError` – Only RGB (3-channel) or Greyscale (1-channel) images are supported.

rescale (*scale*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled by a given factor. Landmarks are rescaled appropriately.

Parameters

- **scale** (*float* or *tuple* of *floats*) – The scale factor. If a tuple, the scale to apply to each dimension. If a single *float*, the scale will be applied uniformly across each dimension.
- **round** (`{ceil, floor, round}`, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (`type(self)`) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is `True`.

Raises ValueError: – If less scales than dimensions are provided. If any scale is less than or equal to 0.

rescale_landmarks_to_diagonal_range (*diagonal_range*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the `diagonal_range` of the bounding box containing its landmarks matches the specified `diagonal_range` range.

Parameters

- **diagonal_range** (`(n_dims,)` *ndarray*) – The diagonal_range range that we want the landmarks of the returned image to have.
- **group** (*str*, optional) – The key of the landmark set that should be used. If `None` and if there is only one set of landmarks, this set will be used.
- **round** (`{ceil, floor, round}`, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the

range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (`type(self)`) – A copy of this image, rescaled.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

rescale_pixels(*minimum*, *maximum*, *per_channel=True*)

A copy of this image with pixels linearly rescaled to fit a range.

Note that the only pixels that will considered and rescaled are those that feature in the vectorized form of this image. If you want to use this routine on all the pixels in a `MaskedImage`, consider using *as_unmasked()* prior to this call.

Parameters

- **minimum** (*float*) – The minimal value of the rescaled pixels
- **maximum** (*float*) – The maximal value of the rescaled pixels
- **per_channel** (*boolean*, optional) – If `True`, each channel will be rescaled independently. If `False`, the scaling will be over all channels.

Returnsrescaled_image (`type(self)`) – A copy of this image with pixels linearly rescaled to fit in the range provided.

rescale_to_diagonal(*diagonal*, *round='ceil'*, *return_transform=False*)

Return a copy of this image, rescaled so that the it's diagonal is a new size.

Parameters

- **diagonal** (*int*) – The diagonal size of the new image.
- **round** (`{ceil, floor, round}`, optional) – Rounding function to be applied to floating point shapes.
- **return_transform** (*bool*, optional) – If `True`, then the `Transform` object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (*type(self)*) – A copy of this image, rescaled.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

rescale_to_pointcloud(*pointcloud*, *group=None*, *round='ceil'*, *order=1*, *return_transform=False*)

Return a copy of this image, rescaled so that the scale of a particular group of landmarks matches the scale of the passed reference pointcloud.

Parameters

- **pointcloud** (`PointCloud`) – The reference pointcloud to which the landmarks specified by `group` will be scaled to match.
- **group** (*str*, optional) – The key of the landmark set that should be used. If `None`, and if there is only one set of landmarks, this set will be used.
- **round** (`{ceil, floor, round}`, optional) – Rounding function to be applied to floating point shapes.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the rescale is also returned.

Returns

- **rescaled_image** (type(self)) – A copy of this image, rescaled.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

resize(*shape*, *order=1*, *return_transform=False*)

Return a copy of this image, resized to a particular shape. All image information (landmarks, and mask in the case of *MaskedImage*) is resized appropriately.

Parameters

- **shape** (*tuple*) – The new shape to resize to.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the resize is also returned.

Returns

- **resized_image** (type(self)) – A copy of this image, resized.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

RaisesValueError: – If the number of dimensions of the new shape does not match the number of dimensions of the image.

rolled_channels()

Returns the pixels matrix, with the channels rolled to the back axis. This may be required for interacting with external code bases that require images to have channels as the last axis, rather than the menpo convention of channels as the first axis.

Returns**rolled_channels** (*ndarray*) – Pixels with channels as the back (last) axis.

rotate_ccw_about_centre(*theta*, *degrees=True*, *retain_shape=False*, *cval=0.0*, *round='round'*, *order=1*, *return_transform=False*)

Return a copy of this image, rotated counter-clockwise about its centre.

Note that the *retain_shape* argument defines the shape of the rotated image. If retain_shape=True, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If retain_shape=False, then the returned image has the correct size so that the whole area of the current image is included.

Parameters

- **theta** (*float*) – The angle of rotation about the centre.
- **degrees** (*bool*, optional) – If True, *theta* is interpreted in degrees. If False, theta is interpreted as radians.

- **retain_shape** (*bool*, optional) – If True, then the shape of the rotated image will be the same as the one of current image, so some regions will probably be cropped. If False, then the returned image has the correct size so that the whole area of the current image is included.
- **cval** (*float*, optional) – The value to be set outside the rotated image boundaries.
- **round** ({'ceil', 'floor', 'round'}, optional) – Rounding function to be applied to floating point shapes. This is only used in case retain_shape=True.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]. This is only used in case retain_shape=True.

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **return_transform** (*bool*, optional) – If True, then the Transform object that was used to perform the rotation is also returned.

Returns

- **rotated_image** (type(self)) – The rotated image.
- **transform** (Transform) – The transform that was used. It only applies if *return_transform* is True.

Raises ValueError – Image rotation is presently only supported on 2D images

sample (*points_to_sample*, *order=1*, *mode='constant'*, *cval=0.0*)

Sample this image at the given sub-pixel accurate points. The input PointCloud should have the same number of dimensions as the image e.g. a 2D PointCloud for a 2D multi-channel image. A numpy array will be returned the has the values for every given point across each channel of the image.

If the points to sample are *outside* of the mask (fall on a False value in the mask), an exception is raised. This exception contains the information of which points were outside of the mask (False) and *also* returns the sampled points.

Parameters

- **points_to_sample** (PointCloud) – Array of points to sample from the image. Should be *(n_points, n_dims)*
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]. See warp_to_shape for more information.
- **mode** ({constant, nearest, reflect, wrap}, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode constant, the value outside the image boundaries.

Returns **sampled_pixels** ((*n_points*, *n_channels*) *ndarray*) – The interpolated values taken across every channel of the image.

Raises OutOfMaskSampleError – One of the points to sample was outside of the valid area of the mask (False in the mask). This exception contains both the mask of valid sample points, **as well as** the sampled points themselves, in case you want to ignore the error.

set_boundary_pixels (*value=0.0*, *n_pixels=1*)

Returns a copy of this MaskedImage for which n pixels along the its mask boundary have been set to a particular value. This is useful in situations where there is absent data in the image which can cause, for example, erroneous computations of gradient or features.

Parameters

- **value** (*float* or (n_channels, 1) ndarray) –

- **n_pixels** (*int*, optional) – The number of pixels along the mask boundary that will be set to 0.

Returnsnew_image (*MaskedImage*) – The copy of the image for which the `n` pixels along its mask boundary have been set to a particular value.

set_masked_pixels(*pixels*, *copy=True*)

Deprecated - please use the equivalent `from_vector`

set_patches(*patches*, *patch_centers*, *offset=None*, *offset_index=None*)

Set the values of a group of patches into the correct regions of a copy of this image. Given an array of patches and a set of patch centers, the patches' values are copied in the regions of the image that are centred on the coordinates of the given centers.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` *Image* objects

Currently only 2D images are supported.

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **patch_centers** (*PointCloud*) – The centers to set the patches around.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises

- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

set_patches_around_landmarks(*patches*, *group=None*, *offset=None*, *offset_index=None*)

Set the values of a group of patches around the landmarks existing in a copy of this image. Given an array of patches, a group and a label, the patches' values are copied in the regions of the image that are centred on the coordinates of corresponding landmarks.

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` *Image* objects

Currently only 2D images are supported.

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` *Image* objects.
- **group** (*str* or `None` optional) – The landmark group to use as patch centres.
- **offset** (*list* or *tuple* or `(1, 2)` *ndarray* or `None`, optional) – The offset to apply on the patch centers within the image. If `None`, then `(0, 0)` is used.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If `None`, then `0` is used.

Raises

- `ValueError` – If image is not 2D
- `ValueError` – If offset does not have shape (1, 2)

`view_widget` (*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualizes the image object using an interactive widget. Currently only supports the rendering of 2D images.

Parameters

- **`browser_style`** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the images will have the form of plus/minus buttons or a slider.
- **`figure_size`** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **`style`** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

`warp_to_mask` (*template_mask*, *transform*, *warp_landmarks=False*, *order=1*, *mode='constant'*, *cval=0.0*, *batch_size=None*, *return_transform=False*)

Warps this image into a different reference space.

Parameters

- **`template_mask`** (`BooleanImage`) – Defines the shape of the result, and what pixels should be sampled.
- **`transform`** (`Transform`) – Transform **from the template space back to this image**. Defines, for each pixel location on the template, which pixel location should be sampled from on this image.
- **`warp_landmarks`** (*bool*, optional) – If `True`, result will have the same landmark dictionary as `self`, but with each landmark updated to the warped position.
- **`order`** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **`mode`** (`{constant, nearest, reflect, wrap}`, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **`cval`** (*float*, optional) – Used in conjunction with mode `constant`, the value outside the image boundaries.
- **`batch_size`** (*int* or `None`, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If `None`, no batching is used and all points are warped at once.
- **`return_transform`** (*bool*, optional) – This argument is for internal use only. If `True`, then the `Transform` object is also returned.

Returns

- **warped_image** (`type(self)`) – A copy of this image, warped.
- **transform** (`Transform`) – The transform that was used. It only applies if *return_transform* is `True`.

`warp_to_shape` (*template_shape*, *transform*, *warp_landmarks=False*, *order=1*, *mode='constant'*, *cval=0.0*, *batch_size=None*, *return_transform=False*)

Return a copy of this `MaskedImage` warped into a different reference space.

Parameters

- **template_shape** (*tuple* or *ndarray*) – Defines the shape of the result, and what pixel indices should be sampled (all of them).
- **transform** (*Transform*) – Transform **from the template_shape space back to this image**. Defines, for each index on template_shape, which pixel location should be sampled from on this image.
- **warp_landmarks** (*bool*, optional) – If True, result will have the same landmark dictionary as self, but with each landmark updated to the warped position.
- **order** (*int*, optional) – The order of interpolation. The order has to be in the range [0,5]

Order	Interpolation
0	Nearest-neighbor
1	Bi-linear *(default)*
2	Bi-quadratic
3	Bi-cubic
4	Bi-quartic
5	Bi-quintic

- **mode** ({constant, nearest, reflect, wrap}, optional) – Points outside the boundaries of the input are filled according to the given mode.
- **cval** (*float*, optional) – Used in conjunction with mode constant, the value outside the image boundaries.
- **batch_size** (*int* or None, optional) – This should only be considered for large images. Setting this value can cause warping to become much slower, particular for cached warps such as Piecewise Affine. This size indicates how many points in the image should be warped at a time, which keeps memory usage low. If None, no batching is used and all points are warped at once.
- **return_transform** (*bool*, optional) – This argument is for internal use only. If True, then the *Transform* object is also returned.

Returns

- **warped_image** (*MaskedImage*) – A copy of this image, warped.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

zoom(*scale*, *cval=0.0*, *return_transform=False*)

Return a copy of this image, zoomed about the centre point. scale values greater than 1.0 denote zooming **in** to the image and values less than 1.0 denote zooming **out** of the image. The size of the image will not change, if you wish to scale an image, please see *rescale()*.

Parameters

- **scale** (*float*) – scale > 1.0 denotes zooming in. Thus the image will appear larger and areas at the edge of the zoom will be 'cropped' out. scale < 1.0 denotes zooming out. The image will be padded by the value of cval.
- **cval** (float, optional) – The value to be set outside the rotated image boundaries.
- **return_transform** (*bool*, optional) – If True, then the *Transform* object that was used to perform the zooming is also returned.

Returns

- **zoomed_image** (type(self)) – A copy of this image, zoomed.
- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.

has_landmarks

Whether the object has landmarks.

Type*bool*

height

The height of the image.

This is the height according to image semantics, and is thus the size of the **second to last** dimension.

Type *int*

landmarks

The landmarks object.

Type *LandmarkManager*

n_channels

The number of channels on each pixel in the image.

Type *int*

n_dims

The number of dimensions in the image. The minimum possible `n_dims` is 2.

Type *int*

n_elements

Total number of data points in the image `(prod(shape), n_channels)`

Type *int*

n_landmark_groups

The number of landmark groups on this object.

Type *int*

n_parameters

The length of the vector that this object produces.

Type *int*

n_pixels

Total number of pixels in the image `(prod(shape),)`

Type *int*

shape

The shape of the image (with `n_channel` values at each point).

Type *tuple*

width

The width of the image.

This is the width according to image semantics, and is thus the size of the **last** dimension.

Type *int*

2.3.2 Exceptions

ImageBoundaryError

class `menpo.image.`**`ImageBoundaryError`**(*requested_min*, *requested_max*, *snapped_min*, *snapped_max*)

Bases: `ValueError`

Exception that is thrown when an attempt is made to crop an image beyond the edge of it's boundary.

Parameters

- **requested_min** (`(d,)` *ndarray*) – The per-dimension minimum index requested for the crop
- **requested_max** (`(d,)` *ndarray*) – The per-dimension maximum index requested for the crop
- **snapped_min** (`(d,)` *ndarray*) – The per-dimension minimum index that could be used if the crop was constrained to the image boundaries.

- **requested_max** – The per-dimension maximum index that could be used if the crop was constrained to the image boundaries.

OutOfMaskSampleError

class menpo.image.**OutOfMaskSampleError**(*sampled_mask*, *sampled_values*)

Bases: ValueError

Exception that is thrown when an attempt is made to sample an MaskedImage in an area that is masked out (where the mask is False).

Parameters

- **sampled_mask** (*bool ndarray*) – The sampled mask, True where the image's mask was True and False otherwise. Useful for masking out the sampling array.
- **sampled_values** (*ndarray*) – The sampled values, no attempt at masking is made.

2.4 menpo.feature

2.4.1 Features

no_op

menpo.feature.**no_op**(*image*, **args*, ***kwargs*)

A no operation feature - does nothing but return a copy of the pixels passed in.

Parameters **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.

Returns **pixels** (*Image* or subclass or (X, Y, ..., Z, C) *ndarray*) – A copy of the image that was passed in.

gradient

menpo.feature.**gradient**(*image*, **args*, ***kwargs*)

Calculates the gradient of an input image. The image is assumed to have channel information on the first axis. In the case of multiple channels, it returns the gradient over each axis over each channel as the first axis.

The gradient is computed using second order accurate central differences in the interior and first order accurate one-side (forward or backwards) differences at the boundaries.

Parameters **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array where the first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.

Returns **gradient** (*ndarray*) – The gradient over each axis over each channel. Therefore, the first axis of the gradient of a 2D, single channel image, will have length *2*. The first axis of the gradient of a 2D, 3-channel image, will have length *6*, the ordering being I[:, 0, 0] = [R0_y, G0_y, B0_y, R0_x, G0_x, B0_x]. To be clear, all the y-gradients are returned over each channel, then all the x-gradients.

gaussian_filter

menpo.feature.**gaussian_filter**(*image*, **args*, ***kwargs*)

Calculates the convolution of the input image with a multidimensional Gaussian filter.

Parameters

- **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **sigma** (*float* or *list* of *float*) – The standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a *list*, or as a single *float*, in which case it is equal for all axes.

Returns **output_image** (*Image* or subclass or (X, Y, ..., Z, C) *ndarray*) – The filtered image has the same type and size as the input `pixels`.

igo

`menpo.feature.igo`(*image*, **args*, ***kwargs*)

Extracts Image Gradient Orientation (IGO) features from the input image. The output image has `N * C` number of channels, where `N` is the number of channels of the original image and `C = 2` or `C = 4` depending on whether double angles are used.

Parameters

- **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **double_angles** (*bool*, optional) – Assume that `phi` represents the gradient orientations.

  If this flag is `False`, the features image is the concatenation of `cos(phi)` and `sin(phi)`, thus 2 channels.

  If `True`, the features image is the concatenation of `cos(phi)`, `sin(phi)`, `cos(2 * phi)`, `sin(2 * phi)`, thus 4 channels.
- **verbose** (*bool*, optional) – Flag to print IGO related information.

Returns **igo** (*Image* or subclass or (X, Y, ..., Z, C) *ndarray*) – The IGO features image. It has the same type and shape as the input `pixels`. The output number of channels depends on the `double_angles` flag.

Raises `ValueError` – Image has to be 2D in order to extract IGOs.

References

es

`menpo.feature.es`(*image*, **args*, ***kwargs*)

Extracts Edge Structure (ES) features from the input image. The output image has `N * C` number of channels, where `N` is the number of channels of the original image and `C = 2`.

Parameters

- **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either an image object itself or an array where the first axis represents the number of channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **verbose** (*bool*, optional) – Flag to print ES related information.

Returns **es** (*Image* or subclass or (X, Y, ..., Z, C) *ndarray*) – The ES features image. It has the same type and shape as the input `pixels`. The output number of channels is `C = 2`.

Raises `ValueError` – Image has to be 2D in order to extract ES features.

References

lbp

`menpo.feature.lbp`(*image*, **args*, ***kwargs*)

Extracts Local Binary Pattern (LBP) features from the input image. The output image has `N * C` number of channels, where `N` is the number of channels of the original image and `C` is the number of radius/samples values combinations that are used in the LBP computation.

Parameters

- **pixels** (*Image* or subclass or `(C, X, Y, ..., Z)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **radius** (*int* or *list* of *int* or `None`, optional) – It defines the radius of the circle (or circles) at which the sampling points will be extracted. The radius (or radii) values must be greater than zero. There must be a radius value for each samples value, thus they both need to have the same length. If `None`, then `[1, 2, 3, 4]` is used.
- **samples** (*int* or *list* of *int* or `None`, optional) – It defines the number of sampling points that will be extracted at each circle. The samples value (or values) must be greater than zero. There must be a samples value for each radius value, thus they both need to have the same length. If `None`, then `[8, 8, 8, 8]` is used.
- **mapping_type** ({`u2`, `ri`, `riu2`, `none`}, optional) – It defines the mapping type of the LBP codes. Select `u2` for uniform-2 mapping, `ri` for rotation-invariant mapping, `riu2` for uniform-2 and rotation-invariant mapping and `none` to use no mapping and only the decimal values instead.
- **window_step_vertical** (*float*, optional) – Defines the vertical step by which the window is moved, thus it controls the features density. The metric unit is defined by *window_step_unit*.
- **window_step_horizontal** (*float*, optional) – Defines the horizontal step by which the window is moved, thus it controls the features density. The metric unit is defined by *window_step_unit*.
- **window_step_unit** ({`pixels`, `window`}, optional) – Defines the metric unit of the *window_step_vertical* and *window_step_horizontal* parameters.
- **padding** (*bool*, optional) – If `True`, the output image is padded with zeros to match the input image's size.
- **verbose** (*bool*, optional) – Flag to print LBP related information.
- **skip_checks** (*bool*, optional) – If `True`, do not perform any validation of the parameters.

Returns **lbp** (*Image* or subclass or `(X, Y, ..., Z, C)` *ndarray*) – The ES features image. It has the same type and shape as the input `pixels`. The output number of channels is `C = len(radius) * len(samples)`.

Raises

- `ValueError` – Radius and samples must both be either integers or lists
- `ValueError` – Radius and samples must have the same length
- `ValueError` – Radius must be > 0
- `ValueError` – Radii must be > 0
- `ValueError` – Samples must be > 0
- `ValueError` – Mapping type must be u2, ri, riu2 or none
- `ValueError` – Horizontal window step must be > 0
- `ValueError` – Vertical window step must be > 0
- `ValueError` – Window step unit must be either pixels or window

References

hog

`menpo.feature.hog(image, *args, **kwargs)`

Extracts Histograms of Oriented Gradients (HOG) features from the input image.

Parameters

- **pixels** (`Image` or subclass or `(C, X, Y, ..., Z)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **mode** ({`dense`, `sparse`}, optional) – The `sparse` case refers to the traditional usage of HOGs, so predefined parameters values are used.

  The `sparse` case of `dalaltriggs` algorithm sets `window_height = window_width = block_size` and `window_step_horizontal = window_step_vertical = cell_size`.

  The `sparse` case of `zhuramanan` algorithm sets `window_height = window_width = 3 * cell_size` and `window_step_horizontal = window_step_vertical = cell_size`.

  In the `dense` case, the user can choose values for *window_height*, *window_width*, *window_unit*, *window_step_vertical*, *window_step_horizontal*, *window_step_unit* and *padding* to customize the HOG calculation.
- **window_height** (*float*, optional) – Defines the height of the window. The metric unit is defined by *window_unit*.
- **window_width** (*float*, optional) – Defines the width of the window. The metric unit is defined by *window_unit*.
- **window_unit** ({`blocks`, `pixels`}, optional) – Defines the metric unit of the *window_height* and *window_width* parameters.
- **window_step_vertical** (*float*, optional) – Defines the vertical step by which the window is moved, thus it controls the features' density. The metric unit is defined by *window_step_unit*.
- **window_step_horizontal** (*float*, optional) – Defines the horizontal step by which the window is moved, thus it controls the features' density. The metric unit is defined by *window_step_unit*.
- **window_step_unit** ({`pixels`, `cells`}, optional) – Defines the metric unit of the *window_step_vertical* and *window_step_horizontal* parameters.
- **padding** (*bool*, optional) – If `True`, the output image is padded with zeros to match the input image's size.
- **algorithm** ({`dalaltriggs`, `zhuramanan`}, optional) – Specifies the algorithm used to compute HOGs. `dalaltriggs` is the implementation of [1] and `zhuramanan` is the implementation of [2].
- **cell_size** (*float*, optional) – Defines the cell size in pixels. This value is set to both the width and height of the cell. This option is valid for both algorithms.
- **block_size** (*float*, optional) – Defines the block size in cells. This value is set to both the width and height of the block. This option is valid only for the `dalaltriggs` algorithm.
- **num_bins** (*float*, optional) – Defines the number of orientation histogram bins. This option is valid only for the `dalaltriggs` algorithm.
- **signed_gradient** (*bool*, optional) – Flag that defines whether we use signed or unsigned gradient angles. This option is valid only for the `dalaltriggs` algorithm.
- **l2_norm_clip** (*float*, optional) – Defines the clipping value of the gradients' L2-norm. This option is valid only for the `dalaltriggs` algorithm.
- **verbose** (*bool*, optional) – Flag to print HOG related information.

Returns **hog** (`Image` or subclass or `(X, Y, ..., Z, K)` *ndarray*) – The HOG features image. It has the same type as the input `pixels`. The output number of channels in the case of dalaltriggs is K = num_bins * block_size *block_size and K = 31 in the case of zhuramanan.

Raises

- ValueError – HOG features mode must be either dense or sparse
- ValueError – Algorithm must be either dalaltriggs or zhuramanan
- ValueError – Number of orientation bins must be > 0
- ValueError – Cell size (in pixels) must be > 0
- ValueError – Block size (in cells) must be > 0
- ValueError – Value for L2-norm clipping must be > 0.0
- ValueError – Window height must be >= block size and <= image height
- ValueError – Window width must be >= block size and <= image width
- ValueError – Window unit must be either pixels or blocks
- ValueError – Horizontal window step must be > 0
- ValueError – Vertical window step must be > 0
- ValueError – Window step unit must be either pixels or cells

References

daisy

menpo.feature.**daisy**(*image*, **args*, ***kwargs*)

Extracts Daisy features from the input image. The output image has N * C number of channels, where N is the number of channels of the original image and C is the feature channels determined by the input options. Specifically, C = (rings * histograms + 1) * orientations.

Parameters

- **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **step** (*int*, optional) – The sampling step that defines the density of the output image.
- **radius** (*int*, optional) – The radius (in pixels) of the outermost ring.
- **rings** (*int*, optional) – The number of rings to be used.
- **histograms** (*int*, optional) – The number of histograms sampled per ring.
- **orientations** (*int*, optional) – The number of orientations (bins) per histogram.
- **normalization** (*['l1', 'l2', 'daisy', None], optional*) – It defines how to normalize the descriptors If 'l1' then L1-normalization is applied at each descriptor. If 'l2' then L2-normalization is applied at each descriptor. If 'daisy' then L2-normalization is applied at individual histograms. If None then no normalization is employed.
- **sigmas** (*list* of *float* or None, optional) – Standard deviation of spatial Gaussian smoothing for the centre histogram and for each ring of histograms. The *list* of sigmas should be sorted from the centre and out. I.e. the first sigma value defines the spatial smoothing of the centre histogram and the last sigma value defines the spatial smoothing of the outermost ring. Specifying sigmas overrides the *rings* parameter by setting rings = len(sigmas) - 1.
- **ring_radii** (*list* of *float* or None, optional) – Radius (in pixels) for each ring. Specifying *ring_radii* overrides the *rings* and *radius* parameters by setting rings = len(ring_radii) and radius = ring_radii[-1].

  If both sigmas and ring_radii are given, they must satisfy

```
len(ring_radii) == len(sigmas) + 1
```

  since no radius is needed for the centre histogram.

- **verbose** (*bool*) – Flag to print Daisy related information.

Returns **daisy** (`Image` or subclass or `(X, Y, ..., Z, C)` *ndarray*) – The ES features image. It has the same type and shape as the input `pixels`. The output number of channels is `C = (rings * histograms + 1) * orientations`.

Raises

- `ValueError` – len(sigmas)-1 != len(ring_radii)
- `ValueError` – Invalid normalization method.

References

2.4.2 Optional

The following features are optional and may or may not be available depending on whether the required packages that implement them are available. If conda was used to install menpo then it is highly likely that all the optional packages will be available.

Vlfeat

Features that have been wrapped from the Vlfeat [1] project. Currently, the wrapped features are all variants on the SIFT [2] algorithm.

dsift

`menpo.feature.dsift(image, *args, **kwargs)`

Computes a 2-dimensional dense SIFT features image with `C` number of channels, where `C = num_bins_horizontal * num_bins_vertical * num_or_bins`. The dense SIFT [2] implementation is taken from Vlfeat [1].

Parameters

- **pixels** (`Image` or subclass or `(C, Y, X)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels.
- **window_step_horizontal** (*int*, optional) – Defines the horizontal step by which the window is moved, thus it controls the features density. The metric unit is pixels.
- **window_step_vertical** (*int*, optional) – Defines the vertical step by which the window is moved, thus it controls the features density. The metric unit is pixels.
- **num_bins_horizontal** (*int*, optional) – Defines the number of histogram bins in the X direction.
- **num_bins_vertical** (*int*, optional) – Defines the number of histogram bins in the Y direction.
- **num_or_bins** (*int*, optional) – Defines the number of orientation histogram bins.
- **cell_size_horizontal** (*int*, optional) – Defines cell width in pixels. The cell is the region that is covered by a spatial bin.
- **cell_size_vertical** (*int*, optional) – Defines cell height in pixels. The cell is the region that is covered by a spatial bin.

[1] Vedaldi, Andrea, and Brian Fulkerson. "VLFeat: An open and portable library of computer vision algorithms." Proceedings of the international conference on Multimedia. ACM, 2010.

[2] Lowe, David G. "Distinctive image features from scale-invariant keypoints." International journal of computer vision 60.2 (2004): 91-110.

[2] Lowe, David G. "Distinctive image features from scale-invariant keypoints." International journal of computer vision 60.2 (2004): 91-110.

[1] Vedaldi, Andrea, and Brian Fulkerson. "VLFeat: An open and portable library of computer vision algorithms." Proceedings of the international conference on Multimedia. ACM, 2010.

- **fast** (*bool*, optional) – If `True`, then the windowing function is a piecewise-flat, rather than Gaussian. While this breaks exact SIFT equivalence, in practice it is much faster to compute.
- **verbose** (*bool*, optional) – Flag to print SIFT related information.

Raises

- `ValueError` – Only 2D arrays are supported
- `ValueError` – Size must only contain positive integers.
- `ValueError` – Step must only contain positive integers.
- `ValueError` – Window size must be a positive integer.
- `ValueError` – Geometry must only contain positive integers.

References

fast_dsift

`menpo.feature.`**`fast_dsift`**`()`

Computes a 2-dimensional dense SIFT features image with `C` number of channels, where `C = num_bins_horizontal * num_bins_vertical * num_or_bins`. The dense SIFT [2] implementation is taken from Vlfeat [1].

Parameters

- **pixels** (`Image` or subclass or `(C, Y, X)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels.
- **window_step_horizontal** (*int*, optional) – Defines the horizontal step by which the window is moved, thus it controls the features density. The metric unit is pixels.
- **window_step_vertical** (*int*, optional) – Defines the vertical step by which the window is moved, thus it controls the features density. The metric unit is pixels.
- **num_bins_horizontal** (*int*, optional) – Defines the number of histogram bins in the X direction.
- **num_bins_vertical** (*int*, optional) – Defines the number of histogram bins in the Y direction.
- **num_or_bins** (*int*, optional) – Defines the number of orientation histogram bins.
- **cell_size_horizontal** (*int*, optional) – Defines cell width in pixels. The cell is the region that is covered by a spatial bin.
- **cell_size_vertical** (*int*, optional) – Defines cell height in pixels. The cell is the region that is covered by a spatial bin.
- **fast** (*bool*, optional) – If `True`, then the windowing function is a piecewise-flat, rather than Gaussian. While this breaks exact SIFT equivalence, in practice it is much faster to compute.
- **verbose** (*bool*, optional) – Flag to print SIFT related information.

Raises

- `ValueError` – Only 2D arrays are supported
- `ValueError` – Size must only contain positive integers.
- `ValueError` – Step must only contain positive integers.
- `ValueError` – Window size must be a positive integer.
- `ValueError` – Geometry must only contain positive integers.

References

[2] Lowe, David G. "Distinctive image features from scale-invariant keypoints." International journal of computer vision 60.2 (2004): 91-110.

[1] Vedaldi, Andrea, and Brian Fulkerson. "VLFeat: An open and portable library of computer vision algorithms." Proceedings of the international conference on Multimedia. ACM, 2010.

vector_128_dsift

menpo.feature.**vector_128_dsift**(*x*, *dtype=<type 'numpy.float32'>*)

Computes a SIFT feature vector from a square patch (or image). Patch **must** be square and the output vector will *always* be a `(128,)` vector. Please see `dsift()` for more information.

Parameters

- **x** (`Image` or subclass or `(C, Y, Y)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. Must be square i.e. `height == width`.
- **dtype** (`np.dtype`, optional) – The dtype of the returned vector.

Raises`ValueError` – Only square images are supported.

hellinger_vector_128_dsift

menpo.feature.**hellinger_vector_128_dsift**(*x*, *dtype=<type 'numpy.float32'>*)

Computes a SIFT feature vector from a square patch (or image). Patch **must** be square and the output vector will *always* be a `(128,)` vector. Please see `dsift()` for more information.

The output of `vector_128_dsift()` is normalised using the hellinger norm (also called the Bhattacharyya distance) which is a measure designed to quantify the similarity between two probability distributions. Since SIFT is a histogram based feature, this has been shown to improve performance. Please see [1] for more information.

Parameters

- **x** (`Image` or subclass or `(C, Y, Y)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. Must be square i.e. `height == width`.
- **dtype** (`np.dtype`, optional) – The dtype of the returned vector.

Raises`ValueError` – Only square images are supported.

References

2.4.3 Predefined (Partial Features)

The following features are are built from the features listed above, but are partial functions. This implies that some sensible parameter choices have already been made that provides a unique set of properties.

double_igo

menpo.feature.**double_igo**()

Extracts Image Gradient Orientation (IGO) features from the input image. The output image has `N * C` number of channels, where `N` is the number of channels of the original image and `C = 2` or `C = 4` depending on whether double angles are used.

Parameters

- **pixels** (`Image` or subclass or `(C, X, Y, ..., Z)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **double_angles** (*bool*, optional) – Assume that `phi` represents the gradient orientations.

[1] Arandjelovic, Relja, and Andrew Zisserman. "Three things everyone should know to improve object retrieval.", CVPR, 2012.

If this flag is `False`, the features image is the concatenation of `cos(phi)` and `sin(phi)`, thus 2 channels.

If `True`, the features image is the concatenation of `cos(phi)`, `sin(phi)`, `cos(2 * phi)`, `sin(2 * phi)`, thus 4 channels.

- **verbose** (*bool*, optional) – Flag to print IGO related information.

Returns igo (*Image* or subclass or `(X, Y, ..., Z, C)` *ndarray*) – The IGO features image. It has the same type and shape as the input `pixels`. The output number of channels depends on the `double_angles` flag.

Raises `ValueError` – Image has to be 2D in order to extract IGOs.

References

sparse_hog

`menpo.feature.sparse_hog()`

Extracts Histograms of Oriented Gradients (HOG) features from the input image.

Parameters

- **pixels** (*Image* or subclass or `(C, X, Y, ..., Z)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels. This means an N-dimensional image is represented by an N+1 dimensional array.
- **mode** (`{dense, sparse}`, optional) – The `sparse` case refers to the traditional usage of HOGs, so predefined parameters values are used.

  The `sparse` case of `dalaltriggs` algorithm sets `window_height = window_width = block_size` and `window_step_horizontal = window_step_vertical = cell_size`.

  The `sparse` case of `zhuramanan` algorithm sets `window_height = window_width = 3 * cell_size` and `window_step_horizontal = window_step_vertical = cell_size`.

  In the `dense` case, the user can choose values for *window_height*, *window_width*, *window_unit*, *window_step_vertical*, *window_step_horizontal*, *window_step_unit* and *padding* to customize the HOG calculation.
- **window_height** (*float*, optional) – Defines the height of the window. The metric unit is defined by *window_unit*.
- **window_width** (*float*, optional) – Defines the width of the window. The metric unit is defined by *window_unit*.
- **window_unit** (`{blocks, pixels}`, optional) – Defines the metric unit of the *window_height* and *window_width* parameters.
- **window_step_vertical** (*float*, optional) – Defines the vertical step by which the window is moved, thus it controls the features' density. The metric unit is defined by *window_step_unit*.
- **window_step_horizontal** (*float*, optional) – Defines the horizontal step by which the window is moved, thus it controls the features' density. The metric unit is defined by *window_step_unit*.
- **window_step_unit** (`{pixels, cells}`, optional) – Defines the metric unit of the *window_step_vertical* and *window_step_horizontal* parameters.
- **padding** (*bool*, optional) – If `True`, the output image is padded with zeros to match the input image's size.
- **algorithm** (`{dalaltriggs, zhuramanan}`, optional) – Specifies the algorithm used to compute HOGs. `dalaltriggs` is the implementation of [1] and `zhuramanan` is the implementation of [2].

- **cell_size** (*float*, optional) – Defines the cell size in pixels. This value is set to both the width and height of the cell. This option is valid for both algorithms.
- **block_size** (*float*, optional) – Defines the block size in cells. This value is set to both the width and height of the block. This option is valid only for the `dalaltriggs` algorithm.
- **num_bins** (*float*, optional) – Defines the number of orientation histogram bins. This option is valid only for the `dalaltriggs` algorithm.
- **signed_gradient** (*bool*, optional) – Flag that defines whether we use signed or unsigned gradient angles. This option is valid only for the `dalaltriggs` algorithm.
- **l2_norm_clip** (*float*, optional) – Defines the clipping value of the gradients' L2-norm. This option is valid only for the `dalaltriggs` algorithm.
- **verbose** (*bool*, optional) – Flag to print HOG related information.

Returns **hog** (*Image* or subclass or `(X, Y, ..., Z, K)` *ndarray*) – The HOG features image. It has the same type as the input `pixels`. The output number of channels in the case of `dalaltriggs` is `K = num_bins * block_size *block_size` and `K = 31` in the case of `zhuramanan`.

Raises

- `ValueError` – HOG features mode must be either dense or sparse
- `ValueError` – Algorithm must be either dalaltriggs or zhuramanan
- `ValueError` – Number of orientation bins must be > 0
- `ValueError` – Cell size (in pixels) must be > 0
- `ValueError` – Block size (in cells) must be > 0
- `ValueError` – Value for L2-norm clipping must be > 0.0
- `ValueError` – Window height must be >= block size and <= image height
- `ValueError` – Window width must be >= block size and <= image width
- `ValueError` – Window unit must be either pixels or blocks
- `ValueError` – Horizontal window step must be > 0
- `ValueError` – Vertical window step must be > 0
- `ValueError` – Window step unit must be either pixels or cells

References

2.4.4 Visualization

glyph

`menpo.feature.glyph`(*image*, **args*, ***kwargs*)

Create the glyph of a feature image that can be used for visualization. If *pixels* have negative values, the *use_negative* flag controls whether there will be created a glyph of both positive and negative values concatenated the one on top of the other.

Parameters

- **pixels** (*Image* or subclass or `(C, X, Y, ..., Z)` *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels.
- **vectors_block_size** (*int*) – Defines the size of each block with vectors of the glyph image.
- **use_negative** (*bool*) – Defines whether to take into account possible negative values of feature_data.
- **channels** (*list* of *int* or `None`) – The list of channels to be used. If `None`, then all the channels are employed.

sum_channels

menpo.feature.**sum_channels**(*image*, **args*, ***kwargs*)

Create the sum of the channels of an image that can be used for visualization.

Parameters

- **pixels** (*Image* or subclass or (C, X, Y, ..., Z) *ndarray*) – Either the image object itself or an array with the pixels. The first dimension is interpreted as channels.
- **channels** (*list* of *int* or None) – The list of channels to be used. If None, then all the channels are employed.

2.4.5 Widget

features_selection_widget

menpo.feature.**features_selection_widget**()

Widget that allows for easy selection of a features function and its options. It also has a 'preview' tab for visual inspection. It returns a *list* of length 1 with the selected features function closure.

Returns

features_function (*list* of length 1) – The function closure of the features function using *functools.partial*. So the function can be called as:

```
features_image = features_function[0](image)
```

Examples

The widget can be invoked as

```
from menpo.feature import features_selection_widget
features_fun = features_selection_widget()
```

And the returned function can be used as

```
import menpo.io as mio
image = mio.import_builtin_asset.lenna_png()
features_image = features_fun[0](image)
```

2.4.6 References

2.5 menpo.landmark

2.5.1 Abstract Classes

Landmarkable

class menpo.landmark.**Landmarkable**

Bases: *Copyable*

Abstract interface for object that can have landmarks attached to them. Landmarkable objects have a public dictionary of landmarks which are managed by a *LandmarkManager*. This means that different sets of landmarks can be attached to the same object. Landmarks can be N-dimensional and are expected to be some subclass of `PointCloud`. These landmarks are wrapped inside a `LandmarkGroup` object that performs useful tasks like label filtering and viewing.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other `Copyable` objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns`type(self)` – A copy of this object

n_dims()

The total number of dimensions.

Type*int*

has_landmarks

Whether the object has landmarks.

Type*bool*

landmarks

The landmarks object.

Type`LandmarkManager`

n_landmark_groups

The number of landmark groups on this object.

Type*int*

2.5.2 Exceptions

LabellingError

class `menpo.landmark.`**LabellingError**

Bases: `Exception`

Raised when labelling a landmark manager and the set of landmarks does not match the expected semantic layout.

2.5.3 Landmarks & Labeller

LandmarkManager

class `menpo.landmark.`**LandmarkManager**

Bases: `MutableMapping`, `Transformable`

Store for `LandmarkGroup` instances associated with an object

Every `Landmarkable` instance has an instance of this class available at the `.landmarks` property. It is through this class that all access to landmarks attached to instances is handled. In general the `LandmarkManager` provides a dictionary-like interface for storing landmarks. `LandmarkGroup` instances are stored under string keys - these keys are refereed to as the **group name**. A special case is where there is a single unambiguous `LandmarkGroup` attached to a `LandmarkManager` - in this case `None` can be used as a key to access the sole group.

Note that all landmarks stored on a `Landmarkable` in it's attached `LandmarkManager` are automatically transformed and copied with their parent object.

clear() → None. Remove all items from D.

copy()
Generate an efficient copy of this *LandmarkManager*.
Returns`type(self)` – A copy of this object

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

items_matching(*glob_pattern*)
Yield only items `(group, LandmarkGroup)` where the key matches a given glob.
Parameters**glob_pattern** (*str*) – A glob pattern e.g. 'frontal_face_*'
Yields**item** (`(group, LandmarkGroup)`) – Tuple of group, LandmarkGroup where the group matches the glob

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

keys_matching(*glob_pattern*)
Yield only landmark group names (keys) matching a given glob.
Parameters**glob_pattern** (*str*) – A glob pattern e.g. 'frontal_face_*'
Yields**keys** (*group labels that match the glob pattern*)

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)
Visualizes the landmark manager object using an interactive widget.
Parameters
- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the landmark managers will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

group_labels
All the labels for the landmark set.
Type*list* of *str*

has_landmarks
Whether the object has landmarks or not
Type*int*

n_dims
The total number of dimensions.
Type*int*

n_groups
Total number of labels.
Type*int*

LandmarkGroup

class menpo.landmark.**LandmarkGroup**(*pointcloud*, *labels_to_masks*, *copy=True*)
Bases: MutableMapping, *Copyable*, Viewable

An immutable object that holds a *PointCloud* (or a subclass) and stores labels for each point. These labels are defined via masks on the *PointCloud*. For this reason, the *PointCloud* is considered to be immutable.

The labels to masks must be within an *OrderedDict* so that semantic ordering can be maintained.

Parameters
- **pointcloud** (*PointCloud*) – The pointcloud representing the landmarks.
- **labels_to_masks** (*ordereddict* {*str* -> *bool ndarray*}) – For each label, the mask that specifies the indices in to the pointcloud that belong to the label.
- **copy** (*bool*, optional) – If True, a copy of the *PointCloud* is stored on the group.

Raises
- ValueError – If *dict* passed instead of *OrderedDict*
- ValueError – If no set of label masks is passed.
- ValueError – If any of the label masks differs in size to the pointcloud.
- ValueError – If there exists any point in the pointcloud that is not covered by a label.

clear() → None. Remove all items from D.

copy()
Generate an efficient copy of this *LandmarkGroup*.
Returnstype(self) – A copy of this object

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_nan_values()
Tests if the LandmarkGroup contains nan values or not. This is particularly useful for annotations with unknown values or non-visible landmarks that have been mapped to nan values.
Returnshas_nan_values (*bool*) – If the LandmarkGroup contains nan values.

classmethod init_from_indices_mapping(*pointcloud*, *labels_to_indices*, *copy=True*)
Static constructor to create a *LandmarkGroup* from an ordered dictionary that maps a set of indices .

Parameters
- **pointcloud** (*PointCloud*) – The pointcloud representing the landmarks.
- **labels_to_indices** (*ordereddict* {*str* -> *int ndarray*}) – For each label, the indices in to the pointcloud that belong to the label.
- **copy** (*boolean*, optional) – If True, a copy of the *PointCloud* is stored on the group.

Returnslmark_group (*LandmarkGroup*) – Landmark group wrapping the given pointcloud with the given semantic labels applied.

Raises
- ValueError – If *dict* passed instead of *OrderedDict*
- ValueError – If any of the label masks differs in size to the pointcloud.
- ValueError – If there exists any point in the pointcloud that is not covered by a label.

classmethod **init_with_all_label** (*pointcloud*, *copy=True*)
Static constructor to create a *LandmarkGroup* with a single default 'all' label that covers all points.
Parameters
- **pointcloud** (*PointCloud*) – The pointcloud representing the landmarks.
- **copy** (*boolean*, optional) – If True, a copy of the *PointCloud* is stored on the group.

Returns **lmark_group** (*LandmarkGroup*) – Landmark group wrapping the given pointcloud with a single label called 'all' that is True for all points.

items () → list of D's (key, value) pairs, as 2-tuples

iteritems () → an iterator over the (key, value) items of D

iterkeys () → an iterator over the keys of D

itervalues () → an iterator over the values of D

keys () → list of D's keys

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem () → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

tojson ()
Convert this *LandmarkGroup* to a dictionary JSON representation.
Returns **json** (dict) – Dictionary conforming to the LJSON v2 specification.

update ([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values () → list of D's values

view_widget (*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)
Visualizes the landmark group object using an interactive widget.
Parameters
- **browser_style** ({'buttons', 'slider'}, optional) – It defines whether the selector of the landmark managers will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **style** ({'coloured', 'minimal'}, optional) – If 'coloured', then the style of the widget will be coloured. If minimal, then the style is simple using black and white colours.

with_labels (*labels=None*)
A new landmark group that contains only the certain labels
Parameters **labels** (*str* or *list* of *str*, optional) – Labels that should be kept in the returned landmark group. If None is passed, and if there is only one label on this group, the label will be substituted automatically.
Returns **landmark_group** (*LandmarkGroup*) – A new landmark group with the same group label but containing only the given label.

without_labels (*labels*)
A new landmark group that excludes certain labels label.
Parameters **labels** (*str* or *list* of *str*) – Labels that should be excluded in the returned landmark group.

Returns landmark_group (*LandmarkGroup*) – A new landmark group with the same group label but containing all labels except the given label.

labels
The list of labels that belong to this group.
Type *list* of *str*

lms
The pointcloud representing all the landmarks in the group.
Type *PointCloud*

n_dims
The dimensionality of these landmarks.
Type *int*

n_labels
Number of labels in the group.
Type *int*

n_landmarks
The total number of landmarks in the group.
Type *int*

labeller

`menpo.landmark.labeller(landmarkable, group, label_func)`
Re-label an existing landmark group on a *Landmarkable* object with a new label set.

Parameters
- **landmarkable** (*Landmarkable*) – *Landmarkable* that will have it's *LandmarkManager* augmented with a new *LandmarkGroup*
- **group** (*str*) – The group label of the existing landmark group that should be re-labelled. A copy of this group will be attached to it's landmark manager with new labels. The group label of this new group and the labels it will have is determined by `label_func`
- **label_func** (*func -> (str, LandmarkGroup)*) – A labelling function taken from this module, Takes as input a *LandmarkGroup* and returns a tuple of (new group label, new LandmarkGroup with semantic labels applied).

Returns landmarkable (*Landmarkable*) – Augmented `landmarkable` (this is just for convenience, the object will actually be modified in place)

2.5.4 Bounding Box Labels

bounding_box_mirrored_to_bounding_box

`menpo.landmark.bounding_box_mirrored_to_bounding_box(x, return_mapping=False)`
Apply a single 'all' label to a given bounding box that has been mirrored around the vertical axis (flipped around the Y-axis). This bounding box must be as specified by the `bounding_box` method (but mirrored).

Parameters
- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This

parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises*: :map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

bounding_box_to_bounding_box

`menpo.landmark.`**`bounding_box_to_bounding_box`**`(`*x*, *return_mapping=False*`)`

Apply a single 'all' label to a given bounding box. This bounding box must be as specified by the `bounding_box` method.

Parameters

- **`x`** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **`return_mapping`** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises*: :map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.5 Face Labels

face_ibug_68_to_face_ibug_49

`menpo.landmark.`**`face_ibug_68_to_face_ibug_49`**`(`*x*, *return_mapping=False*`)`

Apply the IBUG 49-point semantic labels, but removing the annotations corresponding to the jaw region and the 2 describing the inner mouth corners.

The semantic labels applied are as follows:

- left_eyebrow

- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises*: :map:‘LabellingError‘* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_49_trimesh

menpo.landmark.**face_ibug_68_to_face_ibug_49_trimesh**(*x*, *return_mapping=False*)

Apply the IBUG 49-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific

labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).

- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_51

`menpo.landmark.`**`face_ibug_68_to_face_ibug_51`**(*x*, *return_mapping=False*)

Apply the IBUG 51-point semantic labels, but removing the annotations corresponding to the jaw region.

The semantic labels applied are as follows:

- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_51_trimesh

`menpo.landmark.`**`face_ibug_68_to_face_ibug_51_trimesh`**(*x*, *return_mapping=False*)

Apply the IBUG 51-point semantic labels, with trimesh connectivity..

The semantic labels applied are as follows:

- tri

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_65

`menpo.landmark.`**`face_ibug_68_to_face_ibug_65`**(*x*, *return_mapping=False*)

Apply the IBUG 68 point semantic labels, but ignore the 3 points that are coincident for a closed mouth (bottom of the inner mouth).

The semantic labels applied are as follows:

- jaw
- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This

parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_66

menpo.landmark.**face_ibug_68_to_face_ibug_66**(*x*, *return_mapping=False*)

Apply the IBUG 66-point semantic labels, but ignoring the 2 points describing the inner mouth corners).

The semantic labels applied are as follows:

- jaw
- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_66_trimesh

menpo.landmark.**face_ibug_68_to_face_ibug_66_trimesh**(*x*, *return_mapping=False*)

Apply the IBUG 66-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_68

menpo.landmark.**face_ibug_68_to_face_ibug_68**(*x*, *return_mapping=False*)

Apply the IBUG 68-point semantic labels.

The semantic labels are as follows:

- jaw
- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).

- **return_mapping** (*bool*, optional) – Only applicable if a `PointCloud` or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting `PointCloud` (which is then used to for building a `LandmarkGroup`. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (`LandmarkGroup` or `PointCloud`) – If a `LandmarkGroup` was passed, a `LandmarkGroup` is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a `PointCloud` was passed, a `PointCloud` is returned. Only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_to_face_ibug_68_trimesh

menpo.landmark.**face_ibug_68_to_face_ibug_68_trimesh**(*x*, *return_mapping=False*)

Apply the IBUG 68-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

References

Parameters

- **x** (`LandmarkGroup` or `PointCloud` or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a `PointCloud` or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting `PointCloud` (which is then used to for building a `LandmarkGroup`. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (`LandmarkGroup` or `PointCloud`) – If a `LandmarkGroup` was passed, a `LandmarkGroup` is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a `PointCloud` was passed, a `PointCloud` is returned. Only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_68_mirrored_to_face_ibug_68

menpo.landmark.**face_ibug_68_mirrored_to_face_ibug_68**(*x*, *return_mapping=False*)

Apply the IBUG 68-point semantic labels, on a pointcloud that has been mirrored around the vertical axis (flipped around the Y-axis). Thus, on the flipped image the jaw etc would be the wrong way around. This rectifies that and returns a new PointCloud whereby all the points are oriented correctly.

The semantic labels applied are as follows:

- jaw
- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_ibug_49_to_face_ibug_49

menpo.landmark.**face_ibug_49_to_face_ibug_49**(*x*, *return_mapping=False*)

Apply the IBUG 49-point semantic labels.

The semantic labels applied are as follows:

- left_eyebrow
- right_eyebrow
- nose
- left_eye
- right_eye
- mouth

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_imm_58_to_face_imm_58

`menpo.landmark.`**`face_imm_58_to_face_imm_58`**(*x*, *return_mapping=False*)

Apply the 58-point semantic labels from the IMM dataset.

The semantic labels applied are as follows:

- jaw
- left_eye
- right_eye
- left_eyebrow
- right_eyebrow
- mouth
- nose

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_lfpw_29_to_face_lfpw_29

`menpo.landmark.face_lfpw_29_to_face_lfpw_29(x, return_mapping=False)`

Apply the 29-point semantic labels from the original LFPW dataset.

The semantic labels applied are as follows:

- chin
- left_eye
- right_eye
- left_eyebrow
- right_eyebrow
- mouth
- nose

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

face_bu3dfe_83_to_face_bu3dfe_83

menpo.landmark.**face_bu3dfe_83_to_face_bu3dfe_83**(*x*, *return_mapping=False*)

Apply the BU-3DFE (Binghamton University 3D Facial Expression) Database 83-point facial semantic labels.

The semantic labels applied are as follows:

- right_eye
- left_eye
- right_eyebrow
- left_eyebrow
- right_nose
- left_nose
- nostrils
- outer_mouth
- inner_mouth
- jaw

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.6 Eyes Labels

eye_ibug_close_17_to_eye_ibug_close_17

menpo.landmark.**eye_ibug_close_17_to_eye_ibug_close_17**(*x*, *return_mapping=False*)

Apply the IBUG 17-point close eye semantic labels.

The semantic labels applied are as follows:

- upper_eyelid
- lower_eyelid

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

eye_ibug_close_17_to_eye_ibug_close_17_trimesh

menpo.landmark.**eye_ibug_close_17_to_eye_ibug_close_17_trimesh**(*x*, *return_mapping=False*)

Apply the IBUG 17-point close eye semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

eye_ibug_open_38_to_eye_ibug_open_38

menpo.landmark.**eye_ibug_open_38_to_eye_ibug_open_38**(*x*, *return_mapping=False*)

Apply the IBUG 38-point open eye semantic labels.

The semantic labels applied are as follows:

- upper_eyelid
- lower_eyelid
- iris
- pupil
- sclera

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

eye_ibug_open_38_to_eye_ibug_open_38_trimesh

menpo.landmark.**eye_ibug_open_38_to_eye_ibug_open_38_trimesh**(*x*, *return_mapping=False*)

Apply the IBUG 38-point open eye semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific

labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).

- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.7 Hands Labels

hand_ibug_39_to_hand_ibug_39

`menpo.landmark.`**`hand_ibug_39_to_hand_ibug_39`**(*x*, *return_mapping=False*)

Apply the IBUG 39-point semantic labels.

The semantic labels applied are as follows:

- thumb
- index
- middle
- ring
- pinky
- palm

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).

- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.8 Pose Labels

pose_flic_11_to_pose_flic_11

menpo.landmark.**pose_flic_11_to_pose_flic_11**(*x*, *return_mapping=False*)

Apply the flic 11-point semantic labels.

The semantic labels applied are as follows:

- left_arm
- right_arm
- hips
- face

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

pose_human36M_32_to_pose_human36M_17

menpo.landmark.**pose_human36M_32_to_pose_human36M_17**(*x*, *return_mapping=False*)

Apply the human3.6M 17-point semantic labels (based on the original semantic labels of Human3.6 but removing the annotations corresponding to duplicate points, soles and palms), originally 32-points.

The semantic labels applied are as follows:

- pelvis
- right_leg
- left_leg
- spine
- head
- left_arm
- right_arm
- torso

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

pose_human36M_32_to_pose_human36M_32

menpo.landmark.**pose_human36M_32_to_pose_human36M_32**(*x*, *return_mapping=False*)

Apply the human3.6M 32-point semantic labels.

The semantic labels applied are as follows:

- pelvis
- right_leg
- left_leg
- spine
- head
- left_arm
- left_hand
- right_arm
- right_hand
- torso

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This

parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

pose_lsp_14_to_pose_lsp_14

`menpo.landmark.`**`pose_lsp_14_to_pose_lsp_14`**(*x*, *return_mapping=False*)

Apply the lsp 14-point semantic labels.

The semantic labels applied are as follows:

- left_leg
- right_leg
- left_arm
- right_arm
- head

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

pose_stickmen_12_to_pose_stickmen_12

menpo.landmark.**pose_stickmen_12_to_pose_stickmen_12**(*x*, *return_mapping=False*)

Apply the 'stickmen' 12-point semantic labels.

The semantic labels applied are as follows:

- torso
- right_upper_arm
- left_upper_arm
- right_lower_arm
- left_lower_arm
- head

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.9 Car Labels

car_streetscene_20_to_car_streetscene_view_0_8

menpo.landmark.**car_streetscene_20_to_car_streetscene_view_0_8**(*x*, *return_mapping=False*)

Apply the 8-point semantic labels of "view 0" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- front
- bonnet
- windshield

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_1_14

`menpo.landmark.`**`car_streetscene_20_to_car_streetscene_view_1_14`**(*x*, *return_mapping=False*)

Apply the 14-point semantic labels of "view 1" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- front
- bonnet
- windshield
- left_side

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity

information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_2_10

menpo.landmark.**car_streetscene_20_to_car_streetscene_view_2_10**(*x*, *return_mapping=False*)

Apply the 10-point semantic labels of "view 2" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:
- left_side

References

Parameters
- **x** (`LandmarkGroup` or `PointCloud` or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a `PointCloud` or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting `PointCloud` (which is then used to for building a `LandmarkGroup`. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns
- **x_labelled** (`LandmarkGroup` or `PointCloud`) – If a `LandmarkGroup` was passed, a `LandmarkGroup` is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a `PointCloud` was passed, a `PointCloud` is returned. Only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_3_14

menpo.landmark.**car_streetscene_20_to_car_streetscene_view_3_14**(*x*, *return_mapping=False*)

Apply the 14-point semantic labels of "view 3" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:
- left_side
- rear windshield
- trunk
- rear

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_4_14

`menpo.landmark.`**`car_streetscene_20_to_car_streetscene_view_4_14`**(*x*, *return_mapping=False*)

Apply the 14-point semantic labels of "view 4" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- front
- bonnet
- windshield
- right_side

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific

labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a `PointCloud` was passed, a `PointCloud` is returned. Only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).

- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_5_10

`menpo.landmark.`**`car_streetscene_20_to_car_streetscene_view_5_10`**(*x*, *return_mapping=False*)

Apply the 10-point semantic labels of "view 5" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- right_side

References

Parameters

- **x** (`LandmarkGroup` or `PointCloud` or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **`return_mapping`** (*bool*, optional) – Only applicable if a `PointCloud` or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting `PointCloud` (which is then used to for building a `LandmarkGroup`. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (`LandmarkGroup` or `PointCloud`) – If a `LandmarkGroup` was passed, a `LandmarkGroup` is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a `PointCloud` was passed, a `PointCloud` is returned. Only the connectivity information is propagated to the pointcloud (a subclass of `PointCloud` may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building `LandmarkGroup`.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_6_14

`menpo.landmark.`**`car_streetscene_20_to_car_streetscene_view_6_14`**(*x*, *return_mapping=False*)

Apply the 14-point semantic labels of "view 6" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- right_side
- rear_windshield
- trunk
- rear

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of PointCloud may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a PointCloud or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting PointCloud (which is then used to for building a LandmarkGroup. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a LandmarkGroup was passed, a LandmarkGroup is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a PointCloud was passed, a PointCloud is returned. Only the connectivity information is propagated to the pointcloud (a subclass of PointCloud may be returned).
- **mapping_dict** (*ordereddict* {*str* -> *int ndarray*}, optional) – Only returned if `return_mapping==True`. Used for building LandmarkGroup.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_7_8

`menpo.landmark.`**`car_streetscene_20_to_car_streetscene_view_7_8`**(*x*, *return_mapping=False*)

Apply the 8-point semantic labels of "view 7" from the MIT Street Scene Car dataset (originally a 20-point markup).

The semantic labels applied are as follows:

- rear_windshield
- trunk
- rear

References

Parameters

- **x** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of PointCloud may be returned).
- **return_mapping** (*bool*, optional) – Only applicable if a PointCloud or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting PointCloud (which is then used to for building a LandmarkGroup. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.5.10 Tongue Labels

tongue_ibug_19_to_tongue_ibug_19

`menpo.landmark.`**`tongue_ibug_19_to_tongue_ibug_19`**(*x*, *return_mapping=False*)

Apply the IBUG 19-point tongue semantic labels.

The semantic labels applied are as follows:

- outline
- bisector

Parameters

- **`x`** (*LandmarkGroup* or *PointCloud* or *ndarray*) – The input landmark group, pointcloud or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **`return_mapping`** (*bool*, optional) – Only applicable if a *PointCloud* or *ndarray* is passed. Returns the mapping dictionary which maps labels to indices into the resulting *PointCloud* (which is then used to for building a *LandmarkGroup*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LandmarkGroup* or *PointCloud*) – If a *LandmarkGroup* was passed, a *LandmarkGroup* is returned. This landmark group will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

  If a *PointCloud* was passed, a *PointCloud* is returned. Only the connectivity information is propagated to the pointcloud (a subclass of *PointCloud* may be returned).
- **mapping_dict** (*ordereddict {str -> int ndarray}*, optional) – Only returned if `return_mapping==True`. Used for building *LandmarkGroup*.

Raises: *:map:'LabellingError'* – If the given landmark group/pointcloud contains less than the expected number of points.

2.6 menpo.math

2.6.1 Decomposition

eigenvalue_decomposition

`menpo.math.`**`eigenvalue_decomposition`**(*C*, *is_inverse=False*, *eps=1e-10*)

Eigenvalue decomposition of a given covariance (or scatter) matrix.

Parameters

- **C** (`(N, N)` *ndarray* or *scipy.sparse*) – The Covariance/Scatter matrix. If it is a *numpy.array*, then *numpy.linalg.eigh* is used. If it is an instance of *scipy.sparse*, then *scipy.sparse.linalg.eigsh* is used. If it is a precision matrix (inverse covariance), then set *is_inverse=True*.
- **is_inverse** (*bool*, optional) – It `True`, then it is assumed that *C* is a precision matrix (inverse covariance). Thus, the eigenvalues will be inverted. If `False`, then it is assumed that *C* is a covariance matrix.
- **eps** (*float*, optional) – Tolerance value for positive eigenvalue. Those eigenvalues smaller than the specified eps value, together with their corresponding eigenvectors, will be automatically discarded. The final limit is computed as

```
limit = np.max(np.abs(eigenvalues)) * eps
```

Returns

- **pos_eigenvectors** (`(N, p)` *ndarray*) – The matrix with the eigenvectors corresponding to positive eigenvalues.
- **pos_eigenvalues** (`(p,)` *ndarray*) – The array of positive eigenvalues.

pca

`menpo.math.`**`pca`**(*X*, *centre=True*, *inplace=False*, *eps=1e-10*)

Apply Principal Component Analysis (PCA) on the data matrix *X*. In the case where the data matrix is very large, it is advisable to set `inplace = True`. However, note this destructively edits the data matrix by subtracting the mean inplace.

Parameters

- **X** (`(n_samples, n_dims)` *ndarray*) – Data matrix.
- **centre** (*bool*, optional) – Whether to centre the data matrix. If *False*, zero will be subtracted.
- **inplace** (*bool*, optional) – Whether to do the mean subtracting inplace or not. This is crucial if the data matrix is greater than half the available memory size.
- **eps** (*float*, optional) – Tolerance value for positive eigenvalue. Those eigenvalues smaller than the specified eps value, together with their corresponding eigenvectors, will be automatically discarded.

Returns

- **U (eigenvectors)** (`(``(n_components, n_dims)``` *ndarray*) – Eigenvectors of the data matrix.
- **l (eigenvalues)** (`(n_components,)` *ndarray*) – Positive eigenvalues of the data matrix.
- **m (mean vector)** (`(n_dimensions,)` *ndarray*) – Mean that was subtracted from the data matrix.

pcacov

menpo.math.**pcacov**(*C*, *is_inverse=False*, *eps=1e-05*)

Apply Principal Component Analysis (PCA) given a covariance/scatter matrix *C*. In the case where the data matrix is very large, it is advisable to set `inplace = True`. However, note this destructively edits the data matrix by subtracting the mean inplace.

Parameters

- **C** (`(N, N)` *ndarray* or *scipy.sparse*) – The Covariance/Scatter matrix. If it is a precision matrix (inverse covariance), then set *is_inverse=True*.
- **is_inverse** (*bool*, optional) – It `True`, then it is assumed that *C* is a precision matrix (inverse covariance). Thus, the eigenvalues will be inverted. If `False`, then it is assumed that *C* is a covariance matrix.
- **eps** (*float*, optional) – Tolerance value for positive eigenvalue. Those eigenvalues smaller than the specified eps value, together with their corresponding eigenvectors, will be automatically discarded.

Returns

- **U (eigenvectors)** (`(n_components, n_dims)` *ndarray*) – Eigenvectors of the data matrix.
- **l (eigenvalues)** (`(n_components,)` *ndarray*) – Positive eigenvalues of the data matrix.

ipca

menpo.math.**ipca**(*B*, *U_a*, *l_a*, *n_a*, *m_a=None*, *f=1.0*, *eps=1e-10*)

Perform Incremental PCA on the eigenvectors `U_a`, eigenvalues `l_a` and mean vector `m_a` (if present) given a new data matrix `B`.

Parameters

- **B** (`(n_samples, n_dims)` *ndarray*) – New data matrix.
- **U_a** (`(n_components, n_dims)` *ndarray*) – Eigenvectors to be updated.
- **l_a** ((n_components) *ndarray*) – Eigenvalues to be updated.
- **n_a** (*int*) – Total number of samples used to produce U_a, s_a and m_a.
- **m_a** (`(n_dims,)` *ndarray*, optional) – Mean to be updated. If `None` or `(n_dims,)` *ndarray* filled with 0s the data matrix will not be centred.
- **f** (`[0, 1]` *float*, optional) – Forgetting factor that weights the relative contribution of new samples vs old samples. If 1.0, all samples are weighted equally and, hence, the results is the exact same as performing batch PCA on the concatenated list of old and new simples. If <1.0, more emphasis is put on the new samples. See [1] for details.
- **eps** (*float*, optional) – Tolerance value for positive eigenvalue. Those eigenvalues smaller than the specified eps value, together with their corresponding eigenvectors, will be automatically discarded.

Returns

- **U (eigenvectors)** (`(n_components, n_dims)` *ndarray*) – Updated eigenvectors.
- **s (eigenvalues)** (`(n_components,)` *ndarray*) – Updated positive eigenvalues.
- **m (mean vector)** (`(n_dims,)` *ndarray*) – Updated mean.

References

2.6.2 Linear Algebra

dot_inplace_right

menpo.math.**dot_inplace_right**(*a*, *b*, *block_size=1000*)

Inplace dot product for memory efficiency. It computes `a * b = c` where `b` will be replaced inplace with `c`.

Parameters

- **a** (`(n_small, k)` *ndarray*, n_small <= k) – The first array to dot - assumed to be small. `n_small` must be smaller than `k` so the result can be stored within the memory space of `b`.
- **b** (`(k, n_big)` *ndarray*) – Second array to dot - assumed to be large. Will be damaged by this function call as it is used to store the output inplace.
- **block_size** (*int*, optional) – The size of the block of `b` that `a` will be dotted against in each iteration. larger block sizes increase the time performance of the dot product at the cost of a higher memory overhead for the operation.

Returnsc (`(n_small, n_big)` *ndarray*) – The output of the operation. Exactly the same as a memory view onto `b` (`b[:n_small]`) as `b` is modified inplace to store the result.

dot_inplace_left

menpo.math.**dot_inplace_left**(*a*, *b*, *block_size=1000*)

Inplace dot product for memory efficiency. It computes `a * b = c`, where `a` will be replaced inplace with `c`.

Parameters

- **a** (`(n_big, k)` *ndarray*) – First array to dot - assumed to be large. Will be damaged by this function call as it is used to store the output inplace.
- **b** (`(k, n_small)` *ndarray*, `n_small <= k`) – The second array to dot - assumed to be small. `n_small` must be smaller than `k` so the result can be stored within the memory space of `a`.
- **block_size** (*int*, optional) – The size of the block of `a` that will be dotted against `b` in each iteration. larger block sizes increase the time performance of the dot product at the cost of a higher memory overhead for the operation.

Returnsc (`(n_big, n_small)` *ndarray*) – The output of the operation. Exactly the same as a memory view onto `a` (`a[:, :n_small]`) as `a` is modified inplace to store the result.

as_matrix

menpo.math.**as_matrix**(*vectorizables*, *length=None*, *return_template=False*, *verbose=False*)

Create a matrix from a list/generator of `Vectorizable` objects. All the objects in the list **must** be the same size when vectorized.

Consider using a generator if the matrix you are creating is large and passing the length of the generator explicitly.

Parameters

- **vectorizables** (*list* or generator if `Vectorizable` objects) – A list or generator of objects that supports the vectorizable interface
- **length** (*int*, optional) – Length of the vectorizable list. Useful if you are passing a generator with a known length.
- **verbose** (*bool*, optional) – If `True`, will print the progress of building the matrix.
- **return_template** (*bool*, optional) – If `True`, will return the first element of the list/generator, which was used as the template. Useful if you need to map back from the matrix to a list of vectorizable objects.

Returns

- **M** ((length, n_features) *ndarray*) – Every row is an element of the list.

- **template** (*Vectorizable*, optional) – If `return_template == True`, will return the template used to build the matrix *M*.

Raises `ValueError` – `vectorizables` terminates in fewer than `length` iterations

from_matrix

`menpo.math.from_matrix(matrix, template)`

Create a generator from a matrix given a template *Vectorizable* objects as a template. The `from_vector` method will be used to reconstruct each object.

If you want a list, warp the returned value in `list()`.

Parameters

- **matrix** ((n_items, n_features) *ndarray*) – A matrix whereby every *row* represents the data of a vectorizable object.
- **template** (*Vectorizable*) – The template object to use to reconstruct each row of the matrix with.

Returns **vectorizables** (generator of *Vectorizable*) – Every row of the matrix becomes an element of the list.

2.6.3 Convolution

log_gabor

`menpo.math.log_gabor(image, **kwargs)`

Creates a log-gabor filter bank, including smoothing the images via a low-pass filter at the edges.

To create a 2D filter bank, simply specify the number of phi orientations (orientations in the xy-plane).

To create a 3D filter bank, you must specify both the number of phi (azimuth) and theta (elevation) orientations.

This algorithm is directly derived from work by Peter Kovesi.

Parameters

- **image** (`(M, N, ...)` *ndarray*) – Image to be convolved
- **num_scales** (*int*, optional) – Number of wavelet scales.

Default 2D	4
Default 3D	4

- **num_phi_orientations** (*int*, optional) – Number of filter orientations in the xy-plane

Default 2D	6
Default 3D	6

- **num_theta_orientations** (*int*, optional) – **Only required for 3D**. Number of filter orientations in the z-plane

Default 2D	N/A
Default 3D	4

- **min_wavelength** (*int*, optional) – Wavelength of smallest scale filter.

Default 2D	3
Default 3D	3

- **scaling_constant** (*int*, optional) – Scaling factor between successive filters.

Default 2D	2
Default 3D	2

- **center_sigma** (*float*, optional) – Ratio of the standard deviation of the Gaussian describing the Log Gabor filter's transfer function in the frequency domain to the filter centre frequency.

Default 2D	0.65
Default 3D	0.65

- **d_phi_sigma** (*float*, optional) – Angular bandwidth in xy-plane

Default 2D	1.3
Default 3D	1.5

- **d_theta_sigma** (*float*, optional) – **Only required for 3D**. Angular bandwidth in z-plane

Default 2D	N/A
Default 3D	1.5

Returns

- **complex_conv** (`(num_scales, num_orientations, image.shape)` *ndarray*) – Complex valued convolution results. The real part is the result of convolving with the even symmetric filter, the imaginary part is the result from convolution with the odd symmetric filter.
- **bandpass** (`(num_scales, image.shape)` *ndarray*) – Bandpass images corresponding to each scale *s*
- **S** (`(image.shape,)` *ndarray*) – Convolved image

Examples

Return the magnitude of the convolution over the image at scale *s* and orientation *o*

```
np.abs(complex_conv[s, o, :, :])
```

Return the phase angles

```
np.angle(complex_conv[s, o, :, :])
```

References

2.7 menpo.model

2.7.1 Abstract Classes

LinearVectorModel

class menpo.model.**LinearVectorModel**(*components*)

Bases: *Copyable*

A Linear Model contains a matrix of vector components, each component vector being made up of *features*.

Parameterscomponents (`(n_components, n_features)` *ndarray*) – The components array.

component(*index*)

A particular component of the model.

Parametersindex (*int*) – The component that is to be returned.

Returnscomponent_vector ((n_features,) *ndarray*) – The component vector.

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.
Returnstype(self) – A copy of this object

instance(*weights*)
Creates a new vector instance of the model by weighting together the components.
Parametersweights ((n_weights,) *ndarray* or *list*) – The weightings for the first *n_weights* components that should be used.

weights[j] is the linear contribution of the j'th principal component to the instance vector.
Returnsvector ((n_features,) *ndarray*) – The instance vector for the weighting provided.

instance_vectors(*weights*)
Creates new vectorized instances of the model using all the components of the linear model.
Parametersweights ((n_vectors, n_weights) *ndarray* or *list* of *lists*) – The weightings for all components of the linear model. All components will be used to produce the instance.

weights[i, j] is the linear contribution of the j'th principal component to the i'th instance vector produced.
RaisesValueError – If n_weights > n_available_components
Returnsvectors ((n_vectors, n_features) *ndarray*) – The instance vectors for the weighting provided.

orthonormalize_against_inplace(*linear_model*)
Enforces that the union of this model's components and another are both mutually orthonormal.

Both models keep its number of components unchanged or else a value error is raised.
Parameterslinear_model (*LinearVectorModel*) – A second linear model to orthonormalize this against.
RaisesValueError – The number of features must be greater or equal than the sum of the number of components in both linear models ({} < {})

orthonormalize_inplace()
Enforces that this model's components are orthonormalized, s.t. component_vector(i).dot(component_vector(j) = dirac_delta.

project(*vector*)
Projects the *vector* onto the model, retrieving the optimal linear reconstruction weights.
Parametersvector ((n_features,) *ndarray*) – A vectorized novel instance.
Returnsweights ((n_components,) *ndarray*) – A vector of optimal linear weights.

project_out(*vector*)
Returns a version of *vector* where all the basis of the model have been projected out.
Parametersvector ((n_features,) *ndarray*) – A novel vector.
Returnsprojected_out ((n_features,) *ndarray*) – A copy of *vector* with all basis of the model projected out.

project_out_vectors(*vectors*)
Returns a version of *vectors* where all the basis of the model have been projected out.

Parameters **vectors** ((n_vectors, n_features) *ndarray*) – A matrix of novel vectors.

Returns **projected_out** ((n_vectors, n_features) *ndarray*) – A copy of *vectors* with all basis of the model projected out.

project_vectors (*vectors*)

Projects each of the *vectors* onto the model, retrieving the optimal linear reconstruction weights for each instance.

Parameters **vectors** ((n_samples, n_features) *ndarray*) – Array of vectorized novel instances.

Returns **weights** ((n_samples, n_components) *ndarray*) – The matrix of optimal linear weights.

reconstruct (*vector*)

Project a *vector* onto the linear space and rebuild from the weights found.

Parameters **vector** ((n_features,) *ndarray*) – A vectorized novel instance to project.

Returns **reconstructed** ((n_features,) *ndarray*) – The reconstructed vector.

reconstruct_vectors (*vectors*)

Projects the *vectors* onto the linear space and rebuilds vectors from the weights found.

Parameters **vectors** ((n_vectors, n_features) *ndarray*) – A set of vectors to project.

Returns **reconstructed** ((n_vectors, n_features) *ndarray*) – The reconstructed vectors.

components

The components matrix of the linear model.

Type (n_available_components, n_features) *ndarray*

n_components

The number of bases of the model.

Type *int*

n_features

The number of elements in each linear component.

Type *int*

MeanLinearVectorModel

class menpo.model.**MeanLinearVectorModel** (*components*, *mean*)

Bases: LinearVectorModel

A Linear Model containing a matrix of vector components, each component vector being made up of *features*. The model additionally has a mean component which is handled accordingly when either:

1. A component of the model is selected
2. A projection operation is performed

Parameters

- **components** ((n_components, n_features) *ndarray*) – The components array.
- **mean** ((n_features,) *ndarray*) – The mean vector.

component (*index*, *with_mean=True*, *scale=1.0*)

A particular component of the model, in vectorized form.

Parameters

- **index** (*int*) – The component that is to be returned

- **with_mean** (*bool*, optional) – If `True`, the component will be blended with the mean vector before being returned. If not, the component is returned on it's own.
- **scale** (*float*, optional) – A scale factor that should be directly applied to the component. Only valid in the case where `with_mean == True`.

Returns **component_vector** (`(n_features,)` *ndarray*) – The component vector.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

instance(*weights*)

Creates a new vector instance of the model by weighting together the components.

Parameters **weights** (`(n_weights,)` *ndarray* or *list*) – The weightings for the first *n_weights* components that should be used.

`weights[j]` is the linear contribution of the j'th principal component to the instance vector.

Returns **vector** (`(n_features,)` *ndarray*) – The instance vector for the weighting provided.

instance_vectors(*weights*)

Creates new vectorized instances of the model using all the components of the linear model.

Parameters **weights** (`(n_vectors, n_weights)` *ndarray* or *list* of *lists*) – The weightings for all components of the linear model. All components will be used to produce the instance.

`weights[i, j]` is the linear contribution of the j'th principal component to the i'th instance vector produced.

Raises `ValueError` – If n_weights > n_available_components

Returns **vectors** (`(n_vectors, n_features)` *ndarray*) – The instance vectors for the weighting provided.

mean()

Return the mean of the model.

Type *ndarray*

orthonormalize_against_inplace(*linear_model*)

Enforces that the union of this model's components and another are both mutually orthonormal.

Both models keep its number of components unchanged or else a value error is raised.

Parameters **linear_model** (*LinearVectorModel*) – A second linear model to orthonormalize this against.

Raises `ValueError` – The number of features must be greater or equal than the sum of the number of components in both linear models ({} < {})

orthonormalize_inplace()

Enforces that this model's components are orthonormalized, s.t. `component_vector(i).dot(component_vector(j) = dirac_delta`.

project(*vector*)

Projects the *vector* onto the model, retrieving the optimal linear reconstruction weights.

Parameters **vector** (`(n_features,)` *ndarray*) – A vectorized novel instance.

Returns **weights** (`(n_components,)` *ndarray*) – A vector of optimal linear weights.

project_out (*vector*)
Returns a version of *vector* where all the basis of the model have been projected out.
Parameters **vector** ((n_features,) *ndarray*) – A novel vector.
Returns **projected_out** ((n_features,) *ndarray*) – A copy of *vector* with all basis of the model projected out.

project_out_vectors (*vectors*)
Returns a version of *vectors* where all the bases of the model have been projected out.
Parameters **vectors** ((n_vectors, n_features) *ndarray*) – A matrix of novel vectors.
Returns **projected_out** ((n_vectors, n_features) *ndarray*) – A copy of *vectors* with all bases of the model projected out.

project_vectors (*vectors*)
Projects each of the *vectors* onto the model, retrieving the optimal linear reconstruction weights for each instance.
Parameters **vectors** ((n_samples, n_features) *ndarray*) – Array of vectorized novel instances.
Returns **projected** ((n_samples, n_components) *ndarray*) – The matrix of optimal linear weights.

reconstruct (*vector*)
Project a *vector* onto the linear space and rebuild from the weights found.
Parameters **vector** ((n_features,) *ndarray*) – A vectorized novel instance to project.
Returns **reconstructed** ((n_features,) *ndarray*) – The reconstructed vector.

reconstruct_vectors (*vectors*)
Projects the *vectors* onto the linear space and rebuilds vectors from the weights found.
Parameters **vectors** ((n_vectors, n_features) *ndarray*) – A set of vectors to project.
Returns **reconstructed** ((n_vectors, n_features) *ndarray*) – The reconstructed vectors.

components
The components matrix of the linear model.
Type (n_available_components, n_features) *ndarray*

n_components
The number of bases of the model.
Type *int*

n_features
The number of elements in each linear component.
Type *int*

2.7.2 Principal Component Analysis

PCAModel

class menpo.model.**PCAModel** (*samples*, *centre=True*, *n_samples=None*, *max_n_components=None*, *inplace=True*, *verbose=False*)
Bases: VectorizableBackedModel, PCAVectorModel

A MeanLinearModel where components are Principal Components and the components are vectorized instances.

Principal Component Analysis (PCA) by eigenvalue decomposition of the data's scatter matrix. For details of the implementation of PCA, see `pca`.

Parameters

- **`samples`** (*list* or *iterable* of `Vectorizable`) – List or iterable of samples to build the model from.
- **`centre`** (*bool*, optional) – When `True` (default) PCA is performed after mean centering the data. If `False` the data is assumed to be centred, and the mean will be `0`.
- **`n_samples`** (*int*, optional) – If provided then `samples` must be an iterator that yields `n_samples`. If not provided then samples has to be a *list* (so we know how large the data matrix needs to be).
- **`max_n_components`** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.
- **`inplace`** (*bool*, optional) – If `True` the data matrix is modified in place. Otherwise, the data matrix is copied.
- **`verbose`** (*bool*, optional) – Whether to print building information or not.

`component`(*index*, *with_mean=True*, *scale=1.0*)

Return a particular component of the linear model.

Parameters

- **`index`** (*int*) – The component that is to be returned
- **`with_mean`** (*bool*, optional) – If `True`, the component will be blended with the mean vector before being returned. If not, the component is returned on it's own.
- **`scale`** (*float*, optional) – A scale factor that should be applied to the component. Only valid in the case where `with_mean == True`. See `component_vector()` for how this scale factor is interpreted.

Returns **component** (*type(self.template_instance)*) – The requested component instance.

`component_vector`(**args*, ***kwargs*)

A particular component of the model.

Parameters **`index`** (*int*) – The component that is to be returned.

Returns **component** (*type(self.template_instance)*) – The component instance.

`copy`()

Generate an efficient copy of this object.

Note that Numpy arrays and other `Copyable` objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

`eigenvalues_cumulative_ratio`()

Returns the cumulative ratio between the variance captured by the active components and the total amount of variance present on the original samples.

Returns **eigenvalues_cumulative_ratio** (`(n_active_components,)` *ndarray*) – Array of cumulative eigenvalues.

`eigenvalues_ratio`()

Returns the ratio between the variance captured by each active component and the total amount of variance present on the original samples.

Returns **eigenvalues_ratio** (`(n_active_components,)` *ndarray*) – The active eigenvalues array scaled by the original variance.

`increment`(*samples*, *n_samples=None*, *forgetting_factor=1.0*, *verbose=False*)

Update the eigenvectors, eigenvalues and mean vector of this model by performing incremental PCA on the given samples.

Parameters

- **samples** (*list* of *Vectorizable*) – List of new samples to update the model from.
- **n_samples** (*int*, optional) – If provided then `samples` must be an iterator that yields `n_samples`. If not provided then samples has to be a list (so we know how large the data matrix needs to be).
- **forgetting_factor** (`[0.0, 1.0]` *float*, optional) – Forgetting factor that weights the relative contribution of new samples vs old samples. If 1.0, all samples are weighted equally and, hence, the results is the exact same as performing batch PCA on the concatenated list of old and new simples. If <1.0, more emphasis is put on the new samples. See [1] for details.

References

classmethod `init_from_components`(*components*, *eigenvalues*, *mean*, *n_samples*, *centred*, *max_n_components=None*)

Build the Principal Component Analysis (PCA) using the provided components (eigenvectors) and eigenvalues.

Parameters

- **components** (`(n_components, n_features)` *ndarray*) – The eigenvectors to be used.
- **eigenvalues** (`(n_components, )` *ndarray*) – The corresponding eigenvalues.
- **mean** (*Vectorizable*) – The mean instance. It must be a *Vectorizable* and *not* an *ndarray*.
- **n_samples** (*int*) – The number of samples used to generate the eigenvectors.
- **centred** (*bool*, optional) – When `True` we assume that the data were centered before computing the eigenvectors.
- **max_n_components** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.

classmethod `init_from_covariance_matrix`(*C*, *mean*, *n_samples*, *centred=True*, *is_inverse=False*, *max_n_components=None*)

Build the Principal Component Analysis (PCA) by eigenvalue decomposition of the provided covariance/scatter matrix. For details of the implementation of PCA, see *pcacov*.

Parameters

- **C** (`(n_features, n_features)` *ndarray* or *scipy.sparse*) – The Covariance/Scatter matrix. If it is a precision matrix (inverse covariance), then set *is_inverse=True*.
- **mean** (*Vectorizable*) – The mean instance. It must be a *Vectorizable* and *not* an *ndarray*.
- **n_samples** (*int*) – The number of samples used to generate the covariance matrix.
- **centred** (*bool*, optional) – When `True` we assume that the data were centered before computing the covariance matrix.
- **is_inverse** (*bool*, optional) – It `True`, then it is assumed that *C* is a precision matrix (inverse covariance). Thus, the eigenvalues will be inverted. If `False`, then it is assumed that *C* is a covariance matrix.
- **max_n_components** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.

`instance`(*weights*, *normalized_weights=False*)

Creates a new instance of the model using the first `len(weights)` components.

Parameters

- **weights** ((n_weights,) *ndarray* or *list*) – weights[i] is the linear contribution of the i'th component to the instance vector.
- **normalized_weights** (*bool*, optional) – If True, the weights are assumed to be normalized w.r.t the eigenvalues. This can be easier to create unique instances by making the weights more interpretable.

RaisesValueError – If n_weights > n_components

Returns**instance** (*type(self.template_instance)*) – An instance of the model.

instance_vector(**args*, ***kwargs*)

Creates a new instance of the model using the first len(weights) components.

Parameters**weights** ((n_weights,) *ndarray* or *list*) – weights[i] is the linear contribution of the i'th component to the instance vector.

RaisesValueError – If n_weights > n_components

Returns**instance** (*type(self.template_instance)*) – An instance of the model.

instance_vectors(*weights*, *normalized_weights=False*)

Creates new vectorized instances of the model using the first components in a particular weighting.

Parameters

- **weights** ((n_vectors, n_weights) *ndarray* or *list* of *lists*) – The weightings for the first *n_weights* components that should be used per instance that is to be produced

  weights[i, j] is the linear contribution of the j'th principal component to the i'th instance vector produced. Note that if n_weights < n_components, only the first n_weight components are used in the reconstruction (i.e. unspecified weights are implicitly 0).
- **normalized_weights** (*bool*, optional) – If True, the weights are assumed to be normalized w.r.t the eigenvalues. This can be easier to create unique instances by making the weights more interpretable.

Returns**vectors** ((n_vectors, n_features) *ndarray*) – The instance vectors for the weighting provided.

RaisesValueError – If n_weights > n_components

inverse_noise_variance()

Returns the inverse of the noise variance.

Returns**inverse_noise_variance** (*float*) – Inverse of the noise variance.

RaisesValueError – If noise_variance() == 0

mean()

Return the mean of the model.

Type*Vectorizable*

noise_variance()

Returns the average variance captured by the inactive components, i.e. the sample noise assumed in a Probabilistic PCA formulation.

If all components are active, then noise_variance == 0.0.

Returns**noise_variance** (*float*) – The mean variance of the inactive components.

noise_variance_ratio()

Returns the ratio between the noise variance and the total amount of variance present on the original samples.

Returns**noise_variance_ratio** (*float*) – The ratio between the noise variance and the variance present in the original samples.

original_variance()

Returns the total amount of variance captured by the original model, i.e. the amount of variance present on the original samples.

Returnsoptional_variance (*float*) – The variance captured by the model.

orthonormalize_against_inplace(*linear_model*)

Enforces that the union of this model's components and another are both mutually orthonormal.

Note that the model passed in is guaranteed to not have it's number of available components changed. This model, however, may loose some dimensionality due to reaching a degenerate state.

The removed components will always be trimmed from the end of components (i.e. the components which capture the least variance). If trimming is performed, *n_components* and *n_available_components* would be altered - see `trim_components()` for details.

Parameterslinear_model (`LinearModel`) – A second linear model to orthonormalize this against.

orthonormalize_inplace()

Enforces that this model's components are orthonormalized, s.t. `component_vector(i).dot(component_vector(j) = dirac_delta`.

plot_eigenvalues(*figure_id=None, new_figure=False, render_lines=True, line_colour='b', line_style='-', line_width=2, render_markers=True, marker_style='o', marker_size=6, marker_face_colour='b', marker_edge_colour='k', marker_edge_width=1.0, render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', figure_size=(10, 6), render_grid=True, grid_line_style='–', grid_line_width=0.5*)

Plot of the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **render_lines** (*bool*, optional) – If `True`, the line will be rendered.
- **line_colour** (`See Below, optional`) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (`See Below, optional`) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (`See Below, optional`) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the grid lines.
- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returns **viewer** (*MatplotlibRenderer*) – The viewer object.

plot_eigenvalues_cumulative_ratio(*figure_id=None, new_figure=False, render_lines=True, line_colour='b', line_style='-', line_width=2, render_markers=True, marker_style='o', marker_size=6, marker_face_colour='b', marker_edge_colour='k', marker_edge_width=1.0, render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', figure_size=(10, 6), render_grid=True, grid_line_style='–', grid_line_width=0.5*)

Plot of the cumulative variance ratio captured by the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **render_lines** (*bool*, optional) – If `True`, the line will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
```

```
or
`list` of length ``3``
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or *None*, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({-, --, -., :}, optional) – The style of the grid lines.
- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returnsviewer (*MatplotlibRenderer*) – The viewer object.

plot_eigenvalues_cumulative_ratio_widget (*figure_size=(10, 6)*, *style='coloured'*)
Plot of the cumulative variance ratio captured by the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

plot_eigenvalues_ratio (*figure_id=None*, *new_figure=False*, *render_lines=True*, *line_colour='b'*, *line_style='-'*, *line_width=2*, *render_markers=True*, *marker_style='o'*, *marker_size=6*, *marker_face_colour='b'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *figure_size=(10, 6)*, *render_grid=True*, *grid_line_style='–'*, *grid_line_width=0.5*)
Plot of the variance ratio captured by the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **render_lines** (*bool*, optional) – If `True`, the line will be rendered.
- **line_colour** (`See Below, optional`) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (`See Below, optional`) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (`See Below, optional`) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_colour** (`See Below, optional`) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
```

```
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or *None*, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the grid lines.
- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returns **viewer** (*MatplotlibRenderer*) – The viewer object.

plot_eigenvalues_ratio_widget (*figure_size=(10, 6)*, *style='coloured'*)

Plot of the variance ratio captured by the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

plot_eigenvalues_widget (*figure_size=(10, 6)*, *style='coloured'*)

Plot of the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

project (*instance*)

Projects the *instance* onto the model, retrieving the optimal linear weightings.

Parameters **instance** (*Vectorizable*) – A novel instance.

Returns **projected** (`(n_components,)` *ndarray*) – A vector of optimal linear weightings.

project_out (*instance*)

Returns a version of *instance* where all the basis of the model have been projected out.

Parameters **instance** (`Vectorizable`) – A novel instance of `Vectorizable`.

Returns **projected_out** (*self.instance_class*) – A copy of *instance*, with all basis of the model projected out.

project_out_vector (**args*, ***kwargs*)
Returns a version of *instance* where all the basis of the model have been projected out.

Parameters **instance** (Vectorizable) – A novel instance of Vectorizable.

Returns **projected_out** (*self.instance_class*) – A copy of *instance*, with all basis of the model projected out.

project_out_vectors (*vectors*)
Returns a version of *vectors* where all the bases of the model have been projected out.

Parameters **vectors** ((n_vectors, n_features) *ndarray*) – A matrix of novel vectors.

Returns **projected_out** ((n_vectors, n_features) *ndarray*) – A copy of *vectors* with all bases of the model projected out.

project_vector (**args*, ***kwargs*)
Projects the *instance* onto the model, retrieving the optimal linear weightings.

Parameters **instance** (*Vectorizable*) – A novel instance.

Returns **projected** ((n_components,) *ndarray*) – A vector of optimal linear weightings.

project_vectors (*vectors*)
Projects each of the *vectors* onto the model, retrieving the optimal linear reconstruction weights for each instance.

Parameters **vectors** ((n_samples, n_features) *ndarray*) – Array of vectorized novel instances.

Returns **projected** ((n_samples, n_components) *ndarray*) – The matrix of optimal linear weights.

project_whitened (*instance*)
Projects the *instance* onto the whitened components, retrieving the whitened linear weightings.

Parameters **instance** (*Vectorizable*) – A novel instance.

Returns **projected** (*(n_components,)*) – A vector of whitened linear weightings

project_whitened_vector (**args*, ***kwargs*)
Projects the *vector_instance* onto the whitened components, retrieving the whitened linear weightings.

Parameters **vector_instance** ((n_features,) *ndarray*) – A novel vector.

Returns **projected** ((n_features,) *ndarray*) – A vector of whitened linear weightings

reconstruct (*instance*)
Projects a *instance* onto the linear space and rebuilds from the weights found.

Syntactic sugar for:

```
instance(project(instance))
```

but faster, as it avoids the conversion that takes place each time.

Parameters **instance** (Vectorizable) – A novel instance of Vectorizable.

Returns **reconstructed** (*self.instance_class*) – The reconstructed object.

reconstruct_vector (**args*, ***kwargs*)
Projects a *instance* onto the linear space and rebuilds from the weights found.

Syntactic sugar for:

```
instance(project(instance))
```

but faster, as it avoids the conversion that takes place each time.

Parameters **instance** (Vectorizable) – A novel instance of Vectorizable.

Returns **reconstructed** (*self.instance_class*) – The reconstructed object.

reconstruct_vectors(*vectors*)
Projects the *vectors* onto the linear space and rebuilds vectors from the weights found.
Parametersvectors ((n_vectors, n_features) *ndarray*) – A set of vectors to project.
Returnsreconstructed ((n_vectors, n_features) *ndarray*) – The reconstructed vectors.

trim_components(*n_components=None*)
Permanently trims the components down to a certain amount. The number of active components will be automatically reset to this particular value.

This will reduce *self.n_components* down to *n_components* (if None, *self.n_active_components* will be used), freeing up memory in the process.

Once the model is trimmed, the trimmed components cannot be recovered.
Parametersn_components (*int* >= 1 or *float* > 0.0 or None, optional) – The number of components that are kept or else the amount (ratio) of variance that is kept. If None, *self.n_active_components* is used.

Notes

In case *n_components* is greater than the total number of components or greater than the amount of variance currently kept, this method does not perform any action.

variance()
Returns the total amount of variance retained by the active components.
Returnsvariance (*float*) – Total variance captured by the active components.

variance_ratio()
Returns the ratio between the amount of variance retained by the active components and the total amount of variance present on the original samples.
Returnsvariance_ratio (*float*) – Ratio of active components variance and total variance present in original samples.

whitened_components()
Returns the active components of the model, whitened.
Returnswhitened_components ((n_active_components, n_features) *ndarray*) – The whitened components.

components
Returns the active components of the model.
Type(n_active_components, n_features) *ndarray*

eigenvalues
Returns the eigenvalues associated with the active components of the model, i.e. the amount of variance captured by each active component, sorted form largest to smallest.
Type(n_active_components,) *ndarray*

mean_vector
Return the mean of the model as a 1D vector.
Type*ndarray*

n_active_components
The number of components currently in use on this model.
Type*int*

n_components
The number of bases of the model.
Type*int*

n_features
The number of elements in each linear component.
Type*int*

PCAVectorModel

class menpo.model.**PCAVectorModel**(*samples*, *centre=True*, *n_samples=None*, *max_n_components=None*, *inplace=True*)
Bases: MeanLinearVectorModel

A MeanLinearModel where components are Principal Components.

Principal Component Analysis (PCA) by eigenvalue decomposition of the data's scatter matrix. For details of the implementation of PCA, see *pca*.

Parameters

- **samples** (*ndarray* or *list* or *iterable* of *ndarray*) – List or iterable of numpy arrays to build the model from, or an existing data matrix.
- **centre** (*bool*, optional) – When True (default) PCA is performed after mean centering the data. If False the data is assumed to be centred, and the mean will be 0.
- **n_samples** (*int*, optional) – If provided then samples must be an iterator that yields n_samples. If not provided then samples has to be a *list* (so we know how large the data matrix needs to be).
- **max_n_components** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.
- **inplace** (*bool*, optional) – If True the data matrix is modified in place. Otherwise, the data matrix is copied.

component(*index*, *with_mean=True*, *scale=1.0*)
A particular component of the model, in vectorized form.
Parameters

- **index** (*int*) – The component that is to be returned
- **with_mean** (*bool*, optional) – If True, the component will be blended with the mean vector before being returned. If not, the component is returned on it's own.
- **scale** (*float*, optional) – A scale factor that should be applied to the component. Only valid in the case where with_mean is True. The scale is applied in units of standard deviations (so a scale of 1.0 *with_mean* visualizes the mean plus 1 std. dev of the component in question).

Returns**component_vector** ((n_features,) *ndarray*) – The component vector of the given index.

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.
Returnstype(self) – A copy of this object

eigenvalues_cumulative_ratio()
Returns the cumulative ratio between the variance captured by the active components and the total amount of variance present on the original samples.
Returns**eigenvalues_cumulative_ratio** ((n_active_components,) *ndarray*) – Array of cumulative eigenvalues.

eigenvalues_ratio()

Returns the ratio between the variance captured by each active component and the total amount of variance present on the original samples.

Returns eigenvalues_ratio ((n_active_components,) *ndarray*) – The active eigenvalues array scaled by the original variance.

increment (*data*, *n_samples=None*, *forgetting_factor=1.0*, *verbose=False*)

Update the eigenvectors, eigenvalues and mean vector of this model by performing incremental PCA on the given samples.

Parameters

- **samples** (*list* of *Vectorizable*) – List of new samples to update the model from.
- **n_samples** (*int*, optional) – If provided then `samples` must be an iterator that yields `n_samples`. If not provided then samples has to be a list (so we know how large the data matrix needs to be).
- **forgetting_factor** ([0.0, 1.0] *float*, optional) – Forgetting factor that weights the relative contribution of new samples vs old samples. If 1.0, all samples are weighted equally and, hence, the results is the exact same as performing batch PCA on the concatenated list of old and new simples. If <1.0, more emphasis is put on the new samples. See [1] for details.

References

classmethod **init_from_components** (*components*, *eigenvalues*, *mean*, *n_samples*, *centred*, *max_n_components=None*)

Build the Principal Component Analysis (PCA) using the provided components (eigenvectors) and eigenvalues.

Parameters

- **components** ((n_components, n_features) *ndarray*) – The eigenvectors to be used.
- **eigenvalues** ((n_components,) *ndarray*) – The corresponding eigenvalues.
- **mean** ((n_features,) *ndarray*) – The mean vector.
- **n_samples** (*int*) – The number of samples used to generate the eigenvectors.
- **centred** (*bool*, optional) – When `True` we assume that the data were centered before computing the eigenvectors.
- **max_n_components** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.

classmethod **init_from_covariance_matrix** (*C*, *mean*, *n_samples*, *centred=True*, *is_inverse=False*, *max_n_components=None*)

Build the Principal Component Analysis (PCA) by eigenvalue decomposition of the provided covariance/scatter matrix. For details of the implementation of PCA, see *pcacov*.

Parameters

- **C** ((n_features, n_features) *ndarray* or *scipy.sparse*) – The Covariance/Scatter matrix. If it is a precision matrix (inverse covariance), then set *is_inverse=True*.
- **mean** ((n_features,) *ndarray*) – The mean vector.
- **n_samples** (*int*) – The number of samples used to generate the covariance matrix.
- **centred** (*bool*, optional) – When `True` we assume that the data were centered before computing the covariance matrix.
- **is_inverse** (*bool*, optional) – It `True`, then it is assumed that *C* is a precision matrix (inverse covariance). Thus, the eigenvalues will be inverted. If `False`, then it is assumed that *C* is a covariance matrix.

- **max_n_components** (*int*, optional) – The maximum number of components to keep in the model. Any components above and beyond this one are discarded.

instance(*weights*, *normalized_weights=False*)

Creates a new vector instance of the model by weighting together the components.

Parameters

- **weights** ((n_weights,) *ndarray* or *list*) – The weightings for the first *n_weights* components that should be used.

  weights[j] is the linear contribution of the j'th principal component to the instance vector.
- **normalized_weights** (*bool*, optional) – If True, the weights are assumed to be normalized w.r.t the eigenvalues. This can be easier to create unique instances by making the weights more interpretable.

Returns **vector** ((n_features,) *ndarray*) – The instance vector for the weighting provided.

instance_vectors(*weights*, *normalized_weights=False*)

Creates new vectorized instances of the model using the first components in a particular weighting.

Parameters

- **weights** ((n_vectors, n_weights) *ndarray* or *list* of *lists*) – The weightings for the first *n_weights* components that should be used per instance that is to be produced

  weights[i, j] is the linear contribution of the j'th principal component to the i'th instance vector produced. Note that if n_weights < n_components, only the first n_weight components are used in the reconstruction (i.e. unspecified weights are implicitly 0).
- **normalized_weights** (*bool*, optional) – If True, the weights are assumed to be normalized w.r.t the eigenvalues. This can be easier to create unique instances by making the weights more interpretable.

Returns **vectors** ((n_vectors, n_features) *ndarray*) – The instance vectors for the weighting provided.

Raises ValueError – If n_weights > n_components

inverse_noise_variance()

Returns the inverse of the noise variance.

Returns **inverse_noise_variance** (*float*) – Inverse of the noise variance.

Raises ValueError – If noise_variance() == 0

mean()

Return the mean of the model.

Type *ndarray*

noise_variance()

Returns the average variance captured by the inactive components, i.e. the sample noise assumed in a Probabilistic PCA formulation.

If all components are active, then noise_variance == 0.0.

Returns **noise_variance** (*float*) – The mean variance of the inactive components.

noise_variance_ratio()

Returns the ratio between the noise variance and the total amount of variance present on the original samples.

Returns **noise_variance_ratio** (*float*) – The ratio between the noise variance and the variance present in the original samples.

original_variance()

Returns the total amount of variance captured by the original model, i.e. the amount of variance present on the original samples.

Returns optional_variance (*float*) – The variance captured by the model.

orthonormalize_against_inplace(*linear_model*)

Enforces that the union of this model's components and another are both mutually orthonormal.

Note that the model passed in is guaranteed to not have it's number of available components changed. This model, however, may loose some dimensionality due to reaching a degenerate state.

The removed components will always be trimmed from the end of components (i.e. the components which capture the least variance). If trimming is performed, *n_components* and *n_available_components* would be altered - see `trim_components()` for details.

Parameters linear_model (`LinearModel`) – A second linear model to orthonormalize this against.

orthonormalize_inplace()

Enforces that this model's components are orthonormalized, s.t. `component_vector(i).dot(component_vector(j) = dirac_delta`.

plot_eigenvalues(*figure_id=None, new_figure=False, render_lines=True, line_colour='b', line_style='-', line_width=2, render_markers=True, marker_style='o', marker_size=6, marker_face_colour='b', marker_edge_colour='k', marker_edge_width=1.0, render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', figure_size=(10, 6), render_grid=True, grid_line_style='–', grid_line_width=0.5*)

Plot of the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **render_lines** (*bool*, optional) – If `True`, the line will be rendered.
- **line_colour** (`See Below, optional`) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (`See Below, optional`) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (`See Below, optional`) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
```

```
or
`list` of length ``3``
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the grid lines.
- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returns **viewer** (*MatplotlibRenderer*) – The viewer object.

plot_eigenvalues_cumulative_ratio(*figure_id=None, new_figure=False, render_lines=True, line_colour='b', line_style='-', line_width=2, render_markers=True, marker_style='o', marker_size=6, marker_face_colour='b', marker_edge_colour='k', marker_edge_width=1.0, render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', figure_size=(10, 6), render_grid=True, grid_line_style='–', grid_line_width=0.5*)

Plot of the cumulative variance ratio captured by the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **render_lines** (*bool*, optional) – If `True`, the line will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or *None*, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({-, --, -., :}, optional) – The style of the grid lines.

- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returnsviewer (*MatplotlibRenderer*) – The viewer object.

plot_eigenvalues_cumulative_ratio_widget (*figure_size=(10, 6)*, *style='coloured'*)

Plot of the cumulative variance ratio captured by the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or None, optional) – The size of the figure in inches.
- **style** ({'coloured', 'minimal'}, optional) – If 'coloured', then the style of the widget will be coloured. If minimal, then the style is simple using black and white colours.

plot_eigenvalues_ratio (*figure_id=None*, *new_figure=False*, *render_lines=True*, *line_colour='b'*, *line_style='-'*, *line_width=2*, *render_markers=True*, *marker_style='o'*, *marker_size=6*, *marker_face_colour='b'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *figure_size=(10, 6)*, *render_grid=True*, *grid_line_style='–'*, *grid_line_width=0.5*)

Plot of the variance ratio captured by the eigenvalues.

Parameters

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **render_lines** (*bool*, optional) – If True, the line will be rendered.
- **line_colour** (See Below, optional) – The colour of the lines. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (See Below, optional) – The style of the markers. Example options

```
{``.``, ``,``, ``o``, ``v``, ``^``, ``<``, ``>``, ``+``,
 ``x``, ``D``, ``d``, ``s``, ``p``, ``*``, ``h``, ``H``,
 ``1``, ``2``, ``3``, ``4``, ``8``}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (See Below, optional) – The face (filling) colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_colour** (See Below, optional) – The edge colour of the markers. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of length ``3``
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{``serif``, ``sans-serif``, ``cursive``, ``fantasy``,
 ``monospace``}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{``ultralight``, ``light``, ``normal``, ``regular``,
 ``book``, ``medium``, ``roman``, ``semibold``,
 ``demibold``, ``demi``, ``bold``, ``heavy``,
 ``extra bold``, ``black``}
```

- **figure_size** ((*float*, *float*) or *None*, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** ({`-`, `--`, `-.`, `:`}, optional) – The style of the grid lines.
- **grid_line_width** (*float*, optional) – The width of the grid lines.

Returnsviewer (`MatplotlibRenderer`) – The viewer object.

plot_eigenvalues_ratio_widget (*figure_size=(10, 6)*, *style='coloured'*)

Plot of the variance ratio captured by the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

plot_eigenvalues_widget (*figure_size=(10, 6)*, *style='coloured'*)

Plot of the eigenvalues using an interactive widget.

Parameters

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

project (*vector*)

Projects the *vector* onto the model, retrieving the optimal linear reconstruction weights.

Parametersvector (`(n_features,)` *ndarray*) – A vectorized novel instance.

Returnsweights (`(n_components,)` *ndarray*) – A vector of optimal linear weights.

project_out (*vector*)

Returns a version of *vector* where all the basis of the model have been projected out.

Parametersvector (`(n_features,)` *ndarray*) – A novel vector.

Returns projected_out (`(n_features,)` *ndarray*) – A copy of *vector* with all basis of the model projected out.

project_out_vectors(*vectors*)
Returns a version of *vectors* where all the bases of the model have been projected out.

Parameters vectors (`(n_vectors, n_features)` *ndarray*) – A matrix of novel vectors.

Returns projected_out (`(n_vectors, n_features)` *ndarray*) – A copy of *vectors* with all bases of the model projected out.

project_vectors(*vectors*)
Projects each of the *vectors* onto the model, retrieving the optimal linear reconstruction weights for each instance.

Parameters vectors (`(n_samples, n_features)` *ndarray*) – Array of vectorized novel instances.

Returns projected (`(n_samples, n_components)` *ndarray*) – The matrix of optimal linear weights.

project_whitened(*vector_instance*)
Projects the *vector_instance* onto the whitened components, retrieving the whitened linear weightings.

Parameters vector_instance (`(n_features,)` *ndarray*) – A novel vector.

Returns projected (`(n_features,)` *ndarray*) – A vector of whitened linear weightings

reconstruct(*vector*)
Project a *vector* onto the linear space and rebuild from the weights found.

Parameters vector (`(n_features, )` *ndarray*) – A vectorized novel instance to project.

Returns reconstructed (`(n_features,)` *ndarray*) – The reconstructed vector.

reconstruct_vectors(*vectors*)
Projects the *vectors* onto the linear space and rebuilds vectors from the weights found.

Parameters vectors (`(n_vectors, n_features)` *ndarray*) – A set of vectors to project.

Returns reconstructed (`(n_vectors, n_features)` *ndarray*) – The reconstructed vectors.

trim_components(*n_components=None*)
Permanently trims the components down to a certain amount. The number of active components will be automatically reset to this particular value.

This will reduce *self.n_components* down to *n_components* (if `None`, *self.n_active_components* will be used), freeing up memory in the process.

Once the model is trimmed, the trimmed components cannot be recovered.

Parameters n_components (*int* >= `1` or *float* > `0.0` or `None`, optional) – The number of components that are kept or else the amount (ratio) of variance that is kept. If `None`, *self.n_active_components* is used.

Notes

In case *n_components* is greater than the total number of components or greater than the amount of variance currently kept, this method does not perform any action.

variance()
Returns the total amount of variance retained by the active components.

Returns variance (*float*) – Total variance captured by the active components.

variance_ratio()

Returns the ratio between the amount of variance retained by the active components and the total amount of variance present on the original samples.

Returnsvariance_ratio (*float*) – Ratio of active components variance and total variance present in original samples.

whitened_components()

Returns the active components of the model, whitened.

Returnswhitened_components ((n_active_components, n_features) *ndarray*) – The whitened components.

components

Returns the active components of the model.

Type(n_active_components, n_features) *ndarray*

eigenvalues

Returns the eigenvalues associated with the active components of the model, i.e. the amount of variance captured by each active component, sorted form largest to smallest.

Type(n_active_components,) *ndarray*

n_active_components

The number of components currently in use on this model.

Type*int*

n_components

The number of bases of the model.

Type*int*

n_features

The number of elements in each linear component.

Type*int*

2.7.3 Gaussian Markov Random Field

GMRFModel

class menpo.model.**GMRFModel**(*samples*, *graph*, *mode='concatenation'*, *n_components=None*, *dtype=<type 'numpy.float64'>*, *sparse=True*, *n_samples=None*, *bias=0*, *incremental=False*, *verbose=False*)

Bases: GMRFVectorModel

Trains a Gaussian Markov Random Field (GMRF).

Parameters

- **samples** (*list* or *iterable* of *Vectorizable*) – List or iterable of samples to build the model from.
- **graph** (*UndirectedGraph* or *DirectedGraph* or *Tree*) – The graph that defines the relations between the features.
- **n_samples** (*int*, optional) – If provided then samples must be an iterator that yields n_samples. If not provided then samples has to be a *list* (so we know how large the data matrix needs to be).
- **mode** ({'concatenation', 'subtraction'}, optional) – Defines the feature vector of each edge. Assuming that $\mathbf{x}_i$ and $\mathbf{x}_j$ are the feature vectors of two adjacent vertices ($i, j : (v_i, v_j) \in E$), then the edge's feature vector in the case of 'concatenation' is

$$\left[{\mathbf{x}_i}^T, {\mathbf{x}_j}^T\right]^T$$

and in the case of 'subtraction'

$$\mathbf{x}_i - \mathbf{x}_j$$

- **n_components** (*int* or None, optional) – When None (default), the covariance matrix of each edge is inverted using *np.linalg.inv*. If *int*, it is inverted using truncated SVD using the specified number of compnents.
- **dtype** (*numpy.dtype*, optional) – The data type of the GMRF's precision matrix. For example, it can be set to *numpy.float32* for single precision or to *numpy.float64* for double precision. Depending on the size of the precision matrix, this option can you a lot of memory.
- **sparse** (*bool*, optional) – When True, the GMRF's precision matrix has type *scipy.sparse.bsr_matrix*, otherwise it is a *numpy.array*.
- **bias** (*int*, optional) – Default normalization is by (N - 1), where N is the number of observations given (unbiased estimate). If *bias* is 1, then normalization is by N. These values can be overridden by using the keyword ddof in numpy versions >= 1.5.
- **incremental** (*bool*, optional) – This argument must be set to True in case the user wants to incrementally update the GMRF. Note that if True, the model occupies 2x memory.
- **verbose** (*bool*, optional) – If True, the progress of the model's training is printed.

Notes

Let us denote a graph as $G = (V, E)$, where $V = \{v_i, v_2, \ldots, v_{|V|}\}$ is the set of $|V|$ vertices and there is an edge $(v_i, v_j) \in E$ for each pair of connected vertices. Let us also assume that we have a set of random variables $X = \{X_i\}, \forall i : v_i \in V$, which represent an abstract feature vector of length k extracted from each vertex v_i, i.e. $\mathbf{x}_i, i : v_i \in V$.

A GMRF is described by an undirected graph, where the vertexes stand for random variables and the edges impose statistical constraints on these random variables. Thus, the GMRF models the set of random variables with a multivariate normal distribution

$$p(X = \mathbf{x}|G) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

We denote by $\mathbf{Q}$ the block-sparse precision matrix that is the inverse of the covariance matrix $\boldsymbol{\Sigma}$, i.e. $\mathbf{Q} = \boldsymbol{\Sigma}^{-1}$. By applying the GMRF we make the assumption that the random variables satisfy the three Markov properties (pairwise, local and global) and that the blocks of the precision matrix that correspond to disjoint vertexes are zero, i.e.

$$\mathbf{Q}_{ij} = \mathbf{0}_{k \times k}, \forall i, j : (v_i, v_j) \notin E$$

References

increment(*samples*, *n_samples=None*, *verbose=False*)

Update the mean and precision matrix of the GMRF by updating the distributions of all the edges.

Parameters

- **samples** (*list* or *iterable* of Vectorizable) – List or iterable of samples to build the model from.
- **n_samples** (*int*, optional) – If provided then samples must be an iterator that yields n_samples. If not provided then samples has to be a list (so we know how large the data matrix needs to be).
- **verbose** (*bool*, optional) – If True, the progress of the model's incremental update is printed.

mahalanobis_distance(*samples*, *subtract_mean=True*, *square_root=False*)
Compute the mahalanobis distance given a sample $\mathbf{x}$ or an array of samples $\mathbf{X}$, i.e.

$$\sqrt{(\mathbf{x}-\boldsymbol{\mu})^T \mathbf{Q} (\mathbf{x}-\boldsymbol{\mu})} \text{ or } \sqrt{(\mathbf{X}-\boldsymbol{\mu})^T \mathbf{Q} (\mathbf{X}-\boldsymbol{\mu})}$$

Parameters

- **samples** (`Vectorizable` or *list* of `Vectorizable`) – The new data sample or a list of samples.
- **subtract_mean** (*bool*, optional) – When `True`, the mean vector is subtracted from the data vector.
- **square_root** (*bool*, optional) – If `False`, the mahalanobis distance gets squared.

mean()
Return the mean of the model.
Type `Vectorizable`

principal_components_analysis(*max_n_components=None*)
Returns a `PCAModel` with the Principal Components.

Note that the eigenvalue decomposition is applied directly on the precision matrix and then the eigenvalues are inverted.

Parameters **max_n_components** (*int* or `None`, optional) – The maximum number of principal components. If `None`, all the components are returned.
Returns **pca** (`PCAModel`) – The PCA model.

GMRFVectorModel

class `menpo.model.`**GMRFVectorModel**(*samples*, *graph*, *n_samples=None*, *mode='concatenation'*, *n_components=None*, *dtype=<type 'numpy.float64'>*, *sparse=True*, *bias=0*, *incremental=False*, *verbose=False*)
Bases: `object`

Trains a Gaussian Markov Random Field (GMRF).

Parameters

- **samples** (*ndarray* or *list* or *iterable* of *ndarray*) – List or iterable of numpy arrays to build the model from, or an existing data matrix.
- **graph** (`UndirectedGraph` or `DirectedGraph` or `Tree`) – The graph that defines the relations between the features.
- **n_samples** (*int*, optional) – If provided then `samples` must be an iterator that yields `n_samples`. If not provided then samples has to be a *list* (so we know how large the data matrix needs to be).
- **mode** (`{'concatenation', 'subtraction'}`, optional) – Defines the feature vector of each edge. Assuming that $\mathbf{x}_i$ and $\mathbf{x}_j$ are the feature vectors of two adjacent vertices ($i,j : (v_i, v_j) \in E$), then the edge's feature vector in the case of `'concatenation'` is

$$\left[{\mathbf{x}_i}^T, {\mathbf{x}_j}^T\right]^T$$

and in the case of `'subtraction'`

$$\mathbf{x}_i - \mathbf{x}_j$$

- **n_components** (*int* or `None`, optional) – When `None` (default), the covariance matrix of each edge is inverted using *np.linalg.inv*. If *int*, it is inverted using truncated SVD using the specified number of compnents.

- **dtype** (*numpy.dtype*, optional) – The data type of the GMRF's precision matrix. For example, it can be set to *numpy.float32* for single precision or to *numpy.float64* for double precision. Depending on the size of the precision matrix, this option can you a lot of memory.
- **sparse** (*bool*, optional) – When `True`, the GMRF's precision matrix has type *scipy.sparse.bsr_matrix*, otherwise it is a *numpy.array*.
- **bias** (*int*, optional) – Default normalization is by `(N - 1)`, where `N` is the number of observations given (unbiased estimate). If *bias* is 1, then normalization is by `N`. These values can be overridden by using the keyword `ddof` in numpy versions >= 1.5.
- **incremental** (*bool*, optional) – This argument must be set to `True` in case the user wants to incrementally update the GMRF. Note that if `True`, the model occupies 2x memory.
- **verbose** (*bool*, optional) – If `True`, the progress of the model's training is printed.

Notes

Let us denote a graph as $G = (V, E)$, where $V = \{v_i, v_2, \ldots, v_{|V|}\}$ is the set of $|V|$ vertices and there is an edge $(v_i, v_j) \in E$ for each pair of connected vertices. Let us also assume that we have a set of random variables $X = \{X_i\}, \forall i : v_i \in V$, which represent an abstract feature vector of length k extracted from each vertex v_i, i.e. $\mathbf{x}_i, i : v_i \in V$.

A GMRF is described by an undirected graph, where the vertexes stand for random variables and the edges impose statistical constraints on these random variables. Thus, the GMRF models the set of random variables with a multivariate normal distribution

$$p(X = \mathbf{x}|G) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

We denote by $\mathbf{Q}$ the block-sparse precision matrix that is the inverse of the covariance matrix $\boldsymbol{\Sigma}$, i.e. $\mathbf{Q} = \boldsymbol{\Sigma}^{-1}$. By applying the GMRF we make the assumption that the random variables satisfy the three Markov properties (pairwise, local and global) and that the blocks of the precision matrix that correspond to disjoint vertexes are zero, i.e.

$$\mathbf{Q}_{ij} = \mathbf{0}_{k \times k}, \forall i, j : (v_i, v_j) \notin E$$

References

increment(*samples*, *n_samples=None*, *verbose=False*)

Update the mean and precision matrix of the GMRF by updating the distributions of all the edges.

Parameters

- **samples** (*ndarray* or *list* or *iterable* of *ndarray*) – List or iterable of numpy arrays to build the model from, or an existing data matrix.
- **n_samples** (*int*, optional) – If provided then `samples` must be an iterator that yields `n_samples`. If not provided then samples has to be a list (so we know how large the data matrix needs to be).
- **verbose** (*bool*, optional) – If `True`, the progress of the model's incremental update is printed.

mahalanobis_distance(*samples*, *subtract_mean=True*, *square_root=False*)

Compute the mahalanobis distance given a sample $\mathbf{x}$ or an array of samples $\mathbf{X}$, i.e.

$$\sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \mathbf{Q} (\mathbf{x} - \boldsymbol{\mu})} \text{ or } \sqrt{(\mathbf{X} - \boldsymbol{\mu})^T \mathbf{Q} (\mathbf{X} - \boldsymbol{\mu})}$$

Parameters

- **samples** (*ndarray*) – A single data vector or an array of multiple data vectors.
- **subtract_mean** (*bool*, optional) – When True, the mean vector is subtracted from the data vector.
- **square_root** (*bool*, optional) – If False, the mahalanobis distance gets squared.

mean()

Return the mean of the model. For this model, returns the same result as mean_vector.

Type*ndarray*

principal_components_analysis(*max_n_components=None*)

Returns a PCAVectorModel with the Principal Components.

Note that the eigenvalue decomposition is applied directly on the precision matrix and then the eigenvalues are inverted.

Parameters **max_n_components** (*int* or None, optional) – The maximum number of principal components. If None, all the components are returned.

Returns **pca** (PCAVectorModel) – The PCA model.

2.8 menpo.shape

2.8.1 Base Classes

Shape

class menpo.shape.base.**Shape**

Bases: *Vectorizable*, *Transformable*, Landmarkable, LandmarkableViewable, Viewable

Abstract representation of shape. Shapes are Transformable, *Vectorizable*, *Landmarkable*, *LandmarkableViewable* and *Viewable*. This base class handles transforming landmarks when the shape is transformed. Therefore, implementations of Shape have to implement the abstract _transform_self_inplace() method that handles transforming the Shape itself.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

from_vector(*vector*)

Build a new instance of the object from it's vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **object** (type(self)) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters vector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()
Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns has_nan_values (*bool*) – If the vectorized object contains nan values.

n_dims()
The total number of dimensions.

Type *int*

has_landmarks
Whether the object has landmarks.

Type *bool*

landmarks
The landmarks object.

Type *LandmarkManager*

n_landmark_groups
The number of landmark groups on this object.

Type *int*

n_parameters
The length of the vector that this object produces.

Type *int*

2.8.2 PointCloud

PointCloud

class menpo.shape.**PointCloud**(*points*, *copy=True*)
Bases: *Shape*

An N-dimensional point cloud. This is internally represented as an *ndarray* of shape (n_points, n_dims). This class is important for dealing with complex functionality such as viewing and representing metadata such as landmarks.

Currently only 2D and 3D pointclouds are viewable.

Parameters

- **points** ((n_points, n_dims) *ndarray*) – The array representing the points.
- **copy** (*bool*, optional) – If False, the points will not be copied on assignment. Note that this will miss out on additional checks. Further note that we still demand that the array is C-contiguous - if it isn't, a copy will be generated anyway. In general this should only be used if you know what you are doing.

_view_2d(*figure_id=None*, *new_figure=False*, *image_view=True*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour='r'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*, *label=None*, ***kwargs*)

Visualization of the PointCloud in 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the PointCloud will be viewed as if it is in the image coordinate system.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (`PointGraphViewer2d`) – The viewer object.

`_view_landmarks_2d`(*group=None, with_labels=None, without_labels=None, figure_id=None, new_figure=False, image_view=True, render_lines=True, line_colour=None, line_style='-', line_width=1, render_markers=True, marker_style='o', marker_size=5, marker_face_colour=None, marker_edge_colour=None, marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_legend=False, legend_title='', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', legend_marker_scale=None, legend_location=2, legend_bbox_to_anchor=(1.05, 1.0), legend_border_axes_pad=None, legend_n_columns=1, legend_horizontal_spacing=None, legend_vertical_spacing=None, legend_border=True, legend_border_padding=None, legend_shadow=False, legend_rounded_corners=False, render_axes=False, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **group** (*str* or``None`` optional) – The landmark group to be visualized. If None and there are more than one landmark groups, an error is raised.
- **with_labels** (None or *str* or *list* of *str*, optional) – If not None, only show the given label(s). Should **not** be used with the without_labels kwarg.
- **without_labels** (None or *str* or *list* of *str*, optional) – If not None, show all except the given label(s). Should **not** be used with the with_labels kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **image_view** (*bool*, optional) – If True the PointCloud will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If True, the landmarks will be numbered.
- **numbers_horizontal_align** ({center, right, left}, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** ({center, top, bottom, baseline}, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.

- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See Below, optional*) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.

- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None` optional) – The size of the figure in inches.

Raises

- `ValueError` – If both `with_labels` and `without_labels` are passed.
- `ValueError` – If the landmark manager doesn't contain the provided group label.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Returns **bounding_box** (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds (*boundary=0*)

The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters **boundary** (*float*) – A optional padding distance that is added to the bounds. Default is `0`, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** (`(n_dims,)` *ndarray*) – The minimum extent of the *PointCloud* and boundary along each dimension
- **max_b** (`(n_dims,)` *ndarray*) – The maximum extent of the *PointCloud* and boundary along each dimension

centre ()

The mean of all the points in this PointCloud (centre of mass).

Returns **centre** (`(n_dims)` *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds ()

The centre of the absolute bounds of this PointCloud. Contrast with `centre()`, which is the mean point position.

Returns **centre** (`n_dims` *ndarray*) – The centre of the bounds of this PointCloud.

constrain_to_bounds (*bounds*)

Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters **bounds** (`(n_dims, n_dims)` tuple of scalars) – The bounds to constrain this pointcloud within.

Returns **constrained** (*PointCloud*) – The constrained pointcloud.

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

distance_to (*pointcloud*, ***kwargs*)

Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters **pointcloud** (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returns **distance_matrix** (`(n_points, n_points)` *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. `distance_matrix[i, j]` is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

from_mask (*mask*)

A 1D boolean array with the same number of elements as the number of points in the PointCloud. This is then broadcast across the dimensions of the PointCloud and returns a new PointCloud containing only those points that were `True` in the mask.

Parameters **mask** (`(n_points,)` *ndarray*) – 1D array of booleans

Returns **pointcloud** (*PointCloud*) – A new pointcloud that has been masked.

Raises `ValueError` – Mask must have same number of points as pointcloud.

from_vector(*vector*)

Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **object** (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

h_points()

Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type `type(self)`

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

classmethod init_2d_grid(*shape*, *spacing=None*)

Create a pointcloud that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.

Returns **shape_cls** (*type(cls)*) – A PointCloud or subclass arranged in a grid.

classmethod init_from_depth_image(*depth_image*)

Return a 3D point cloud from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

Parameters **depth_image** (*Image* or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.

Returns **depth_cloud** (`type(cls)`) – A new 3D PointCloud with unit XY coordinates and the given depth values as Z coordinates.

norm(***kwargs*)

Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returns **norm** (*float*) – The norm of this `PointCloud`

range(*boundary=0*)

The range of the extent of the PointCloud.

Parameters **boundary** (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returns **range** (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

tojson()

Convert this `PointCloud` to a dictionary representation suitable for inclusion in the LJSON landmark format.

Returns **json** (*dict*) – Dictionary with `points` keys.

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualization of the PointCloud using an interactive widget.

Parameters

- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*), optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

has_landmarks

Whether the object has landmarks.

Type *bool*

landmarks

The landmarks object.

Type `LandmarkManager`

n_dims

The number of dimensions in the pointcloud.

Type *int*

n_landmark_groups

The number of landmark groups on this object.

Type *int*

n_parameters

The length of the vector that this object produces.

Type *int*

n_points

The number of points in the pointcloud.

Type *int*

2.8.3 Graphs

UndirectedGraph

class `menpo.shape.`**UndirectedGraph**(*adjacency_matrix*, *copy=True*, *skip_checks=False*)

Bases: `Graph`

Class for Undirected Graph definition and manipulation.

Parameters

- **adjacency_matrix** (`(n_vertices, n_vertices, )` *ndarray* or *csr_matrix*) – The adjacency matrix of the graph. The non-edges must be represented with zeros and the edges can have a weight value.

  Note `adjacency_matrix` must be symmetric.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Raises

- `ValueError` – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- `ValueError` – Graph must have at least two vertices.
- `ValueError` – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.
- `ValueError` – The adjacency matrix of an undirected graph must be symmetric.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0],
                             [1, 0, 1, 1, 0, 0],
                             [1, 1, 0, 0, 1, 0],
                             [0, 1, 0, 0, 1, 1],
                             [0, 0, 1, 1, 0, 0],
                             [0, 0, 0, 1, 0, 0]])
graph = UndirectedGraph(adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(
                    ([1] * 14,
                     ([0, 1, 0, 2, 1, 2, 1, 3, 2, 4, 3, 4, 3, 5],
                      [1, 0, 2, 0, 2, 1, 3, 1, 4, 2, 4, 3, 5, 3])),
                    shape=(6, 6))
graph = UndirectedGraph(adjacency_matrix)
```

The adjacency matrix of the following graph with isolated vertices

```
        0---|
            |
            |
    1       2
            |
            |
    3-------4

5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 1, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0],
                             [1, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 0],
                             [0, 0, 1, 1, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
graph = UndirectedGraph(adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 6, ([0, 2, 2, 4, 3, 4],
                                          [2, 0, 4, 2, 4, 3])),
                              shape=(6, 6))
graph = UndirectedGraph(adjacency_matrix)
```

find_all_paths (*start*, *end*, *path=[]*)

Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.

Parameters

- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.
- **path** (*list*, optional) – An existing path to append to.

Returnspaths (*list* of *list*) – The list containing all the paths from start to end.

find_all_shortest_paths (*algorithm='auto'*, *unweighted=False*)

Returns the distances and predecessors arrays of the graph's shortest paths.

Parameters

- **algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.

- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

find_path (*start*, *end*, *method='bfs'*, *skip_checks=False*)
Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **method** ({`bfs`, `dfs`}, optional) – The method to be used.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returnspath (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

find_shortest_path (*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)
Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **algorithm** (`'str', see below, optional`) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find the path such that the number of edges is minimized.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

- **path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.
- **distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

get_adjacency_list ()
Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returnsadjacency_list (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

has_cycles ()
Checks if the graph has at least one cycle.

Returnshas_cycles (*bool*) – `True` if the graph has cycles.

has_isolated_vertices ()
Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returnshas_isolated_vertices (*bool*) – `True` if the graph has at least one isolated vertex.

classmethod **init_from_edges** (*edges*, *n_vertices*, *skip_checks=False*)
Initialize graph from edges array.

Parameters

- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **n_vertices** (*int*) – The total number of vertices, assuming that the numbering of vertices starts from `0`. `edges` and `n_vertices` can be defined in a way to set isolated vertices.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
from menpo.shape import UndirectedGraph
import numpy as np
edges = np.array([[0, 1], [1, 0], [0, 2], [2, 0], [1, 2], [2, 1],
                  [1, 3], [3, 1], [2, 4], [4, 2], [3, 4], [4, 3],
                  [3, 5], [5, 3]])
graph = UndirectedGraph.init_from_edges(edges, n_vertices=6)
```

Finally, the following graph with isolated vertices

```
        0---|
            |
            |
    1       2
            |
            |
    3-------4

5
```

can be defined as

```
from menpo.shape import UndirectedGraph
import numpy as np
edges = np.array([[0, 2], [2, 0], [2, 4], [4, 2], [3, 4], [4, 3]])
graph = UndirectedGraph.init_from_edges(edges, n_vertices=6)
```

is_edge(*vertex_1*, *vertex_2*, *skip_checks=False*)

Whether there is an edge between the provided vertices.

Parameters

- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.

- **skip_checks** (*bool*, optional) – If False, the given vertices will be checked.

Returns is_edge (*bool*) – True if there is an edge connecting vertex_1 and vertex_2.

Raises ValueError – The vertex must be between 0 and {n_vertices-1}.

is_tree()

Checks if the graph is tree.

Returns is_true (*bool*) – If the graph is a tree.

isolated_vertices()

Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.

Returns isolated_vertices (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

minimum_spanning_tree(*root_vertex*)

Returns the minimum spanning tree of the graph using Kruskal's algorithm.

Parameters root_vertex (*int*) – The vertex that will be set as root in the output MST.

Returns mst (*Tree*) – The computed minimum spanning tree.

Raises ValueError – Cannot compute minimum spanning tree of a graph with isolated vertices

n_neighbours(*vertex*, *skip_checks=False*)

Returns the number of neighbours of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If False, the given vertex will be checked.

Returns n_neighbours (*int*) – The number of neighbours.

Raises ValueError – The vertex must be between 0 and {n_vertices-1}.

n_paths(*start*, *end*)

Returns the number of all the paths (without cycles) existing from start vertex to end vertex.

Parameters

- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returns paths (*int*) – The paths' numbers.

neighbours(*vertex*, *skip_checks=False*)

Returns the neighbours of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If False, the given vertex will be checked.

Returns neighbours (*list*) – The list of neighbours.

Raises ValueError – The vertex must be between 0 and {n_vertices-1}.

n_edges

Returns the number of edges.

Type *int*

n_vertices

Returns the number of vertices.

Type *int*

vertices

Returns the *list* of vertices.

Type *list*

DirectedGraph

class `menpo.shape.`**`DirectedGraph`**`(`*adjacency_matrix*, *copy=True*, *skip_checks=False*`)`

Bases: `Graph`

Class for Directed Graph definition and manipulation.

Parameters

- **`adjacency_matrix`** (`(n_vertices, n_vertices, )` *ndarray* or *csr_matrix*) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.
- **`copy`** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **`skip_checks`** (*bool*, optional) – If `True`, no checks will be performed.

Raises

- `ValueError` – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- `ValueError` – Graph must have at least two vertices.
- `ValueError` – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.

Examples

The following directed graph

```
|-->0<--|
|       |
|       |
1<----->2
|       |
v       v
3------>4
|
v
5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 0, 0, 0, 0],
                             [1, 0, 1, 1, 0, 0],
                             [1, 1, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 1],
                             [0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
graph = DirectedGraph(adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 8, ([1, 2, 1, 2, 1, 2, 3, 3],
                                          [0, 0, 2, 1, 3, 4, 4, 5])),
                              shape=(6, 6))
graph = DirectedGraph(adjacency_matrix)
```

The following graph with isolated vertices

```
        0<--|
            |
            |
    1       2
            |
            v
    3------>4

5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0],
                             [1, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
graph = DirectedGraph(adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 3, ([2, 2, 3], [0, 4, 4])),
                              shape=(6, 6))
graph = DirectedGraph(adjacency_matrix)
```

children (*vertex*, *skip_checks=False*)

Returns the children of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns children (*list*) – The list of children.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

find_all_paths (*start*, *end*, *path=[]*)

Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.

Parameters

- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.
- **path** (*list*, optional) – An existing path to append to.

Returns paths (*list* of *list*) – The list containing all the paths from start to end.

find_all_shortest_paths (*algorithm='auto'*, *unweighted=False*)

Returns the distances and predecessors arrays of the graph's shortest paths.

Parameters

- **algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.
- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

find_path(*start*, *end*, *method='bfs'*, *skip_checks=False*)

Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **method** ({`bfs`, `dfs`}, optional) – The method to be used.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returnspath (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

find_shortest_path(*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)

Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **algorithm** (`'str', see below, optional`) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find the path such that the number of edges is minimized.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

- **path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.
- **distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

get_adjacency_list()

Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returnsadjacency_list (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

has_cycles()

Checks if the graph has at least one cycle.

Returns **has_cycles** (*bool*) – `True` if the graph has cycles.

has_isolated_vertices()

Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returns **has_isolated_vertices** (*bool*) – `True` if the graph has at least one isolated vertex.

init_from_edges(*edges, n_vertices, skip_checks=False*)

Initialize graph from edges array.

Parameters

- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **n_vertices** (*int*) – The total number of vertices, assuming that the numbering of vertices starts from `0`. `edges` and `n_vertices` can be defined in a way to set isolated vertices.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
from menpo.shape import UndirectedGraph
import numpy as np
edges = np.array([[0, 1], [1, 0], [0, 2], [2, 0], [1, 2], [2, 1],
                  [1, 3], [3, 1], [2, 4], [4, 2], [3, 4], [4, 3],
                  [3, 5], [5, 3]])
graph = UndirectedGraph.init_from_edges(edges, n_vertices=6)
```

The following directed graph

```
|-->0<--|
|       |
|       |
1<----->2
|       |
v       v
3------>4
|
v
5
```

can be represented as

```
from menpo.shape import DirectedGraph
import numpy as np
edges = np.array([[1, 0], [2, 0], [1, 2], [2, 1], [1, 3], [2, 4],
                  [3, 4], [3, 5]])
graph = DirectedGraph.init_from_edges(edges, n_vertices=6)
```

Finally, the following graph with isolated vertices

```
    0---|
        |
        |
1       2
        |
        |
3-------4


5
```

can be defined as

```
from menpo.shape import UndirectedGraph
import numpy as np
edges = np.array([[0, 2], [2, 0], [2, 4], [4, 2], [3, 4], [4, 3]])
graph = UndirectedGraph.init_from_edges(edges, n_vertices=6)
```

is_edge (*vertex_1*, *vertex_2*, *skip_checks=False*)

Whether there is an edge between the provided vertices.

Parameters

- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.
- **skip_checks** (*bool*, optional) – If `False`, the given vertices will be checked.

Returns **is_edge** (*bool*) – `True` if there is an edge connecting `vertex_1` and `vertex_2`.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

is_tree ()

Checks if the graph is tree.

Returns **is_true** (*bool*) – If the graph is a tree.

isolated_vertices ()

Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.

Returns **isolated_vertices** (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

n_children (*vertex*, *skip_checks=False*)

Returns the number of children of the selected vertex.

Parameters **vertex** (*int*) – The selected vertex.

Returns

- **n_children** (*int*) – The number of children.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_parents (*vertex*, *skip_checks=False*)

Returns the number of parents of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns n_parents (*int*) – The number of parents.
Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_paths (*start*, *end*)
Returns the number of all the paths (without cycles) existing from start vertex to end vertex.

Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returns paths (*int*) – The paths' numbers.

parents (*vertex*, *skip_checks=False*)
Returns the parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns parents (*list*) – The list of parents.
Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_edges
Returns the number of edges.
Type *int*

n_vertices
Returns the number of vertices.
Type *int*

vertices
Returns the *list* of vertices.
Type *list*

Tree

class `menpo.shape.`**Tree** (*adjacency_matrix*, *root_vertex*, *copy=True*, *skip_checks=False*)
Bases: `DirectedGraph`

Class for Tree definitions and manipulation.

Parameters
- **adjacency_matrix** (`(n_vertices, n_vertices, )` *ndarray* or *csr_matrix*) – The adjacency matrix of the tree in which the rows represent parents and columns represent children. The non-edges must be represented with zeros and the edges can have a weight value.

  Note A tree must not have isolated vertices.
- **root_vertex** (*int*) – The vertex to be set as root.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Raises
- `ValueError` – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- `ValueError` – Graph must have at least two vertices.
- `ValueError` – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.
- `ValueError` – The provided edges do not represent a tree.
- `ValueError` – The root_vertex must be in the range `[0, n_vertices - 1]`.
- `ValueError` – The combination of adjacency matrix and root vertex is not valid. BFS returns a different tree.

Examples

The following tree

```
         0
         |
      ___|___
     1       2
     |       |
    _|_      |
   3   4     5
   |   |     |
   |   |     |
   6   7     8
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 1, 1, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 1, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0, 1, 0, 0],
                             [0, 0, 0, 0, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 0, 0, 0, 0, 1],
                             [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]])
tree = Tree(adjacency_matrix, root_vertex=0)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 8, ([0, 0, 1, 1, 2, 3, 4, 5],
                                          [1, 2, 3, 4, 5, 6, 7, 8])),
                              shape=(9, 9))
tree = Tree(adjacency_matrix, root_vertex=0)
```

children(*vertex*, *skip_checks=False*)

Returns the children of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **children** (*list*) – The list of children.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

depth_of_vertex(*vertex*, *skip_checks=False*)

Returns the depth of the specified vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **depth** (*int*) – The depth of the selected vertex.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

find_all_paths(*start*, *end*, *path=[]*)

Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.

Parameters

- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.
- **path** (*list*, optional) – An existing path to append to.

Returnspaths (*list* of *list*) – The list containing all the paths from start to end.

find_all_shortest_paths (*algorithm='auto'*, *unweighted=False*)

Returns the distances and predecessors arrays of the graph's shortest paths.

Parameters

- **algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.
- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

find_path (*start*, *end*, *method='bfs'*, *skip_checks=False*)

Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **method** ({`bfs`, `dfs`}, optional) – The method to be used.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returnspath (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

find_shortest_path (*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)

Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find

the path such that the number of edges is minimized.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

- **path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.
- **distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

get_adjacency_list()

Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returnsadjacency_list (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

has_cycles()

Checks if the graph has at least one cycle.

Returnshas_cycles (*bool*) – `True` if the graph has cycles.

has_isolated_vertices()

Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returnshas_isolated_vertices (*bool*) – `True` if the graph has at least one isolated vertex.

classmethod init_from_edges(*edges*, *n_vertices*, *root_vertex*, *copy=True*, *skip_checks=False*)

Construct a *Tree* from edges array.

Parameters

- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **n_vertices** (*int*) – The total number of vertices, assuming that the numbering of vertices starts from `0`. `edges` and `n_vertices` can be defined in a way to set isolated vertices.
- **root_vertex** (*int*) – That vertex that will be set as root.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following tree

```
        0
        |
     ___|___
    1       2
    |       |
   _|_      |
  3   4     5
  |   |     |
  |   |     |
  6   7     8
```

can be defined as

```
from menpo.shape import PointTree
import numpy as np
points = np.array([[30, 30], [10, 20], [50, 20], [0, 10], [20, 10],
                   [50, 10], [0, 0], [20, 0], [50, 0]])
edges = np.array([[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [3, 6],
```

```
                          [4, 7], [5, 8]])
    tree = PointTree.init_from_edges(points, edges, root_vertex=0)
```

is_edge(*vertex_1*, *vertex_2*, *skip_checks=False*)
Whether there is an edge between the provided vertices.

Parameters
- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.
- **skip_checks** (*bool*, optional) – If `False`, the given vertices will be checked.

Returns **is_edge** (*bool*) – `True` if there is an edge connecting `vertex_1` and `vertex_2`.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

is_leaf(*vertex*, *skip_checks=False*)
Whether the vertex is a leaf.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **is_leaf** (*bool*) – If `True`, then selected vertex is a leaf.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

is_tree()
Checks if the graph is tree.

Returns **is_true** (*bool*) – If the graph is a tree.

isolated_vertices()
Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.

Returns **isolated_vertices** (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

n_children(*vertex*, *skip_checks=False*)
Returns the number of children of the selected vertex.

Parameters **vertex** (*int*) – The selected vertex.

Returns
- **n_children** (*int*) – The number of children.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_parents(*vertex*, *skip_checks=False*)
Returns the number of parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **n_parents** (*int*) – The number of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_paths(*start*, *end*)
Returns the number of all the paths (without cycles) existing from start vertex to end vertex.

Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returns **paths** (*int*) – The paths' numbers.

n_vertices_at_depth(*depth*)
Returns the number of vertices at the specified depth.

Parameters **depth** (*int*) – The selected depth.

Returns **n_vertices** (*int*) – The number of vertices that lie in the specified depth.

parent (*vertex*, *skip_checks=False*)
Returns the parent of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **parent** (*int*) – The parent vertex.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

parents (*vertex*, *skip_checks=False*)
Returns the parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **parents** (*list*) – The list of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

vertices_at_depth (*depth*)
Returns a list of vertices at the specified depth.

Parameters **depth** (*int*) – The selected depth.

Returns **vertices** (*list*) – The vertices that lie in the specified depth.

leaves
Returns a *list* with the all leaves of the tree.

Type *list*

maximum_depth
Returns the maximum depth of the tree.

Type *int*

n_edges
Returns the number of edges.

Type *int*

n_leaves
Returns the number of leaves of the tree.

Type *int*

n_vertices
Returns the number of vertices.

Type *int*

vertices
Returns the *list* of vertices.

Type *list*

2.8.4 PointGraphs

Mix-ins of Graphs and `PointCloud` for graphs with geometry.

PointUndirectedGraph

class `menpo.shape.`**PointUndirectedGraph** (*points*, *adjacency_matrix*, *copy=True*, *skip_checks=False*)
Bases: `PointGraph`, `UndirectedGraph`

Class for defining an Undirected Graph with geometry.

Parameters

- **points** ((n_vertices, n_dims,) *ndarray*) – The array of point locations.
- **adjacency_matrix** ((n_vertices, n_vertices,) *ndarray* or *csr_matrix*) – The adjacency matrix of the graph. The non-edges must be represented with zeros and the edges can have a weight value.

  Note adjacency_matrix must be symmetric.
- **copy** (*bool*, optional) – If False, the adjacency_matrix will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If True, no checks will be performed.

Raises

- ValueError – A point for each graph vertex needs to be passed. Got n_points points instead of n_vertices.
- ValueError – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- ValueError – Graph must have at least two vertices.
- ValueError – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.
- ValueError – The adjacency matrix of an undirected graph must be symmetric.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0],
                             [1, 0, 1, 1, 0, 0],
                             [1, 1, 0, 0, 1, 0],
                             [0, 1, 0, 0, 1, 1],
                             [0, 0, 1, 1, 0, 0],
                             [0, 0, 0, 1, 0, 0]])
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointUndirectedGraph(points, adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(
                    ([1] * 14,
                     ([0, 1, 0, 2, 1, 2, 1, 3, 2, 4, 3, 4, 3, 5],
                      [1, 0, 2, 0, 2, 1, 3, 1, 4, 2, 4, 3, 5, 3])),
                    shape=(6, 6))
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointUndirectedGraph(points, adjacency_matrix)
```

The adjacency matrix of the following graph with isolated vertices

```
        0---|
            |
            |
    1       2
            |
            |
    3-------4

5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 1, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0],
                             [1, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 0],
                             [0, 0, 1, 1, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointUndirectedGraph(points, adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 6, ([0, 2, 2, 4, 3, 4],
                                         [2, 0, 4, 2, 4, 3])),
                              shape=(6, 6))
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointUndirectedGraph(points, adjacency_matrix)
```

_view_2d(*figure_id=None, new_figure=False, image_view=True, render_lines=True, line_colour='r', line_style='-', line_width=1.0, render_markers=True, marker_style='o', marker_size=5, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8), label=None*)

Visualization of the PointGraph in 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the PointGraph will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({'-', '--', '-.', ':'}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** ({center, right, left}, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** ({center, top, bottom, baseline}, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.

- **axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointGraph as a percentage of the PointGraph's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointGraph as a percentage of the PointGraph's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (PointGraphViewer2d) – The viewer object.

_view_landmarks_2d(*group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05, 1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as view_landmarks if the Image is 2D.

Parameters

- **group** (*str* or``None`` optional) – The landmark group to be visualized. If None and there are more than one landmark groups, an error is raised.
- **with_labels** (None or *str* or *list* of *str*, optional) – If not None, only show the given label(s). Should **not** be used with the without_labels kwarg.
- **without_labels** (None or *str* or *list* of *str*, optional) – If not None, show all except the given label(s). Should **not** be used with the with_labels kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.

- **image_view** (*bool*, optional) – If `True` the PointCloud will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If True, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** ({normal, italic, oblique}, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See Below, optional*) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If True, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If True, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If True, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If True, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Raises

- ValueError – If both with_labels and without_labels are passed.
- ValueError – If the landmark manager doesn't contain the provided group label.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Returns **bounding_box** (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds(*boundary=0*)

The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters **boundary** (*float*) – A optional padding distance that is added to the bounds. Default is `0`, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** (`(n_dims,)` *ndarray*) – The minimum extent of the `PointCloud` and boundary along each dimension
- **max_b** (`(n_dims,)` *ndarray*) – The maximum extent of the `PointCloud` and boundary along each dimension

centre()

The mean of all the points in this PointCloud (centre of mass).

Returns **centre** (`(n_dims)` *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds()

The centre of the absolute bounds of this PointCloud. Contrast with `centre()`, which is the mean point position.

Returns **centre** (`n_dims` *ndarray*) – The centre of the bounds of this PointCloud.

constrain_to_bounds(*bounds*)

Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters **bounds** (`(n_dims, n_dims)` tuple of scalars) – The bounds to constrain this pointcloud within.

Returns **constrained** (`PointCloud`) – The constrained pointcloud.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other `Copyable` objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

distance_to(*pointcloud*, ***kwargs*)

Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters **pointcloud** (`PointCloud`) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returns **distance_matrix** (`(n_points, n_points)` *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. `distance_matrix[i, j]` is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

find_all_paths(*start*, *end*, *path=[]*)

Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.

Parameters

- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.
- **path** (*list*, optional) – An existing path to append to.

Returns **paths** (*list* of *list*) – The list containing all the paths from start to end.

find_all_shortest_paths(*algorithm='auto'*, *unweighted=False*)

Returns the distances and predecessors arrays of the graph's shortest paths.

Parameters

- **algorithm** (`'str', see below, optional`) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.
- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

find_path (*start*, *end*, *method='bfs'*, *skip_checks=False*)

Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **method** ({`bfs`, `dfs`}, optional) – The method to be used.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returnspath (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

find_shortest_path (*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)

Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **algorithm** (`'str', see below, optional`) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find the path such that the number of edges is minimized.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

- **path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.
- **distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

from_mask (*mask*)

A 1D boolean array with the same number of elements as the number of points in the *PointUndirectedGraph*. This is then broadcast across the dimensions of the *PointUndirectedGraph* and returns a new

PointUndirectedGraph containing only those points that were `True` in the mask.

Parameters **mask** (`(n_vertices,)` *ndarray*) – 1D array of booleans

Returns **pointgraph** (*PointUndirectedGraph*) – A new pointgraph that has been masked.

Raises `ValueError` – Mask must be a 1D boolean array of the same number of entries as points in this PointUndirectedGraph.

from_vector(*vector*)

Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **object** (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

get_adjacency_list()

Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returns **adjacency_list** (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

h_points()

Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type `type(self)`

has_cycles()

Checks if the graph has at least one cycle.

Returns **has_cycles** (*bool*) – `True` if the graph has cycles.

has_isolated_vertices()

Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returns **has_isolated_vertices** (*bool*) – `True` if the graph has at least one isolated vertex.

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

init_2d_grid(*shape*, *spacing=None*, *adjacency_matrix=None*, *skip_checks=False*)

Create a PointGraph that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

If no adjacency matrix is provided, the default connectivity will be a 4-connected lattice.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.

- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.

  The adjacency matrix of an undirected graph must be symmetric.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if no adjacency matrix is provided.

Returns **pgraph** (*PointGraph*) – A pointgraph arranged in a grid.

init_from_depth_image(*depth_image*, *spacing=None*, *adjacency_matrix=None*, *skip_checks=False*)

Return a 3D point graph from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

If no adjacency matrix is provided, the default connectivity will be a 4-connected lattice.

Parameters

- **depth_image** (`Image` or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.

  The adjacency matrix of an undirected graph must be symmetric.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if no adjacency matrix is provided.

Returns **depth_cloud** (`type(cls)`) – A new 3D PointGraph with unit XY coordinates and the given depth values as Z coordinates.

classmethod init_from_edges(*points*, *edges*, *copy=True*, *skip_checks=False*)

Construct a `PointUndirectedGraph` from edges array.

Parameters

- **points** (`(n_vertices, n_dims, )` *ndarray*) – The array of point locations.
- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
from menpo.shape import PointUndirectedGraph
import numpy as np
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
edges = np.array([[0, 1], [1, 0], [0, 2], [2, 0], [1, 2], [2, 1],
                  [1, 3], [3, 1], [2, 4], [4, 2], [3, 4], [4, 3],
                  [3, 5], [5, 3]])
graph = PointUndirectedGraph.init_from_edges(points, edges)
```

Finally, the following graph with isolated vertices

```
    0---|
        |
        |
1       2
        |
        |
3-------4


5
```

can be defined as

```
from menpo.shape import PointUndirectedGraph
import numpy as np
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
edges = np.array([[0, 2], [2, 0], [2, 4], [4, 2], [3, 4], [4, 3]])
graph = PointUndirectedGraph.init_from_edges(points, edges)
```

is_edge(*vertex_1*, *vertex_2*, *skip_checks=False*)

Whether there is an edge between the provided vertices.

Parameters

- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.
- **skip_checks** (*bool*, optional) – If `False`, the given vertices will be checked.

Returns **is_edge** (*bool*) – `True` if there is an edge connecting `vertex_1` and `vertex_2`.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

is_tree()

Checks if the graph is tree.

Returns **is_true** (*bool*) – If the graph is a tree.

isolated_vertices()
Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.
Returnsisolated_vertices (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

minimum_spanning_tree(*root_vertex*)
Returns the minimum spanning tree of the graph using Kruskal's algorithm.
Parametersroot_vertex (*int*) – The vertex that will be set as root in the output MST.
Returnsmst (`PointTree`) – The computed minimum spanning tree with the *points* of *self*.
Raises`ValueError` – Cannot compute minimum spanning tree of a graph with isolated vertices

n_neighbours(*vertex*, *skip_checks=False*)
Returns the number of neighbours of the selected vertex.
Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returnsn_neighbours (*int*) – The number of neighbours.
Raises`ValueError` – The vertex must be between 0 and {n_vertices-1}.

n_paths(*start*, *end*)
Returns the number of all the paths (without cycles) existing from start vertex to end vertex.
Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returnspaths (*int*) – The paths' numbers.

neighbours(*vertex*, *skip_checks=False*)
Returns the neighbours of the selected vertex.
Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returnsneighbours (*list*) – The list of neighbours.
Raises`ValueError` – The vertex must be between 0 and {n_vertices-1}.

norm(***kwargs*)
Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.
Returnsnorm (*float*) – The norm of this `PointCloud`

range(*boundary=0*)
The range of the extent of the PointCloud.
Parametersboundary (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.
Returnsrange (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

tojson()
Convert this PointGraph to a dictionary representation suitable for inclusion in the LJSON landmark format.
Returnsjson (*dict*) – Dictionary with `points` and `connectivity` keys.

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)
Visualization of the PointGraph using an interactive widget.
Parameters

- **browser_style** ({'buttons', 'slider'}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **style** ({'coloured', 'minimal'}, optional) – If 'coloured', then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

has_landmarks
Whether the object has landmarks.
Type*bool*

landmarks
The landmarks object.
Type*LandmarkManager*

n_dims
The number of dimensions in the pointcloud.
Type*int*

n_edges
Returns the number of edges.
Type*int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_points
The number of points in the pointcloud.
Type*int*

n_vertices
Returns the number of vertices.
Type*int*

vertices
Returns the *list* of vertices.
Type*list*

PointDirectedGraph

class `menpo.shape.`**PointDirectedGraph**(*points*, *adjacency_matrix*, *copy=True*, *skip_checks=False*)
Bases: `PointGraph`, `DirectedGraph`

Class for defining a directed graph with geometry.

Parameters

- **points** (`(n_vertices, n_dims)` *ndarray*) – The array representing the points.
- **adjacency_matrix** (`(n_vertices, n_vertices, )` *ndarray* or *csr_matrix*) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.

- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Raises

- `ValueError` – A point for each graph vertex needs to be passed. Got {n_points} points instead of {n_vertices}.
- `ValueError` – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- `ValueError` – Graph must have at least two vertices.
- `ValueError` – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.

Examples

The following directed graph

```
|-->0<--|
|       |
|       |
1<----->2
|       |
v       v
3------>4
|
v
5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 0, 0, 0, 0],
                             [1, 0, 1, 1, 0, 0],
                             [1, 1, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 1],
                             [0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointDirectedGraph(points, adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 8, ([1, 2, 1, 2, 1, 2, 3, 3],
                                         [0, 0, 2, 1, 3, 4, 4, 5])),
                              shape=(6, 6))
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointDirectedGraph(points, adjacency_matrix)
```

The following graph with isolated vertices

```
       0<--|
           |
           |
   1       2
           |
           v
   3------>4
```

```
5
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0],
                             [1, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0]])
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointDirectedGraph(points, adjacency_matrix)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 3, ([2, 2, 3], [0, 4, 4])),
                              shape=(6, 6))
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
graph = PointDirectedGraph(points, adjacency_matrix)
```

_view_2d(*figure_id=None, new_figure=False, image_view=True, render_lines=True, line_colour='r', line_style='-', line_width=1.0, render_markers=True, marker_style='o', marker_size=5, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8), label=None*)

Visualization of the PointGraph in 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the PointGraph will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{'-', '--', '-.', ':'}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.

- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointGraph as a percentage of the PointGraph's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.

- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointGraph as a percentage of the PointGraph's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (`PointGraphViewer2d`) – The viewer object.

`_view_landmarks_2d`(*group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05*, *1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **`group`** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **`with_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.
- **`without_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, show all except the given label(s). Should **not** be used with the `with_labels` kwarg.
- **`figure_id`** (*object*, optional) – The id of the figure to be used.
- **`new_figure`** (*bool*, optional) – If `True`, a new figure is created.
- **`image_view`** (*bool*, optional) – If `True` the PointCloud will be viewed as if it is in the image coordinate system.
- **`render_lines`** (*bool*, optional) – If `True`, the edges will be rendered.
- **`line_colour`** (*`See Below, optional`*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **`line_style`** (`{-, --, -., :}`, optional) – The style of the lines.
- **`line_width`** (*float*, optional) – The width of the lines.

- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (`See Below, optional`) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (`See Below, optional`) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (`See Below, optional`) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (`See Below, optional`) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (`See Below, optional`) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (`See Below, optional`) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (`See below, optional`) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.

- **legend_font_weight** (`See Below, optional`) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.

- **axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Raises

- ValueError – If both with_labels and without_labels are passed.
- ValueError – If the landmark manager doesn't contain the provided group label.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns vector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Returns bounding_box (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds(*boundary=0*)

The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters boundary (*float*) – A optional padding distance that is added to the bounds. Default is 0, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** ((n_dims,) *ndarray*) – The minimum extent of the *PointCloud* and boundary along each dimension
- **max_b** ((n_dims,) *ndarray*) – The maximum extent of the *PointCloud* and boundary along each dimension

centre()

The mean of all the points in this PointCloud (centre of mass).

Returns centre ((n_dims) *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds()
The centre of the absolute bounds of this PointCloud. Contrast with *centre()*, which is the mean point position.
Returns centre (n_dims *ndarray*) – The centre of the bounds of this PointCloud.

children(*vertex*, *skip_checks=False*)
Returns the children of the selected vertex.
Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If False, the given vertex will be checked.

Returns children (*list*) – The list of children.
Raises ValueError – The vertex must be between 0 and {n_vertices-1}.

constrain_to_bounds(*bounds*)
Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.
Parameters bounds ((n_dims, n_dims) tuple of scalars) – The bounds to constrain this pointcloud within.
Returns constrained (*PointCloud*) – The constrained pointcloud.

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.
Returns type(self) – A copy of this object

distance_to(*pointcloud*, ***kwargs*)
Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.
Parameters pointcloud (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.
Returns distance_matrix ((n_points, n_points) *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. distance_matrix[i, j] is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

find_all_paths(*start*, *end*, *path=[]*)
Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.
Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.
- **path** (*list*, optional) – An existing path to append to.

Returns paths (*list* of *list*) – The list containing all the paths from start to end.

find_all_shortest_paths(*algorithm='auto'*, *unweighted=False*)
Returns the distances and predecessors arrays of the graph's shortest paths.
Parameters
- **algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.
- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

find_path (*start*, *end*, *method='bfs'*, *skip_checks=False*)

Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **method** ({`bfs`, `dfs`}, optional) – The method to be used.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns**path** (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

find_shortest_path (*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)

Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **start** (*int*) – The vertex from which the path starts.
- **end** (*int*) – The vertex to which the path ends.
- **algorithm** (`'str', see below, optional`) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find the path such that the number of edges is minimized.
- **skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

- **path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.
- **distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

from_mask (*mask*)

A 1D boolean array with the same number of elements as the number of points in the *PointDirectedGraph*. This is then broadcast across the dimensions of the *PointDirectedGraph* and returns a new *PointDirected-*

Graph containing only those points that were `True` in the mask.

Parameters **mask** (`(n_points,)` *ndarray*) – 1D array of booleans

Returns **pointgraph** (*PointDirectedGraph*) – A new pointgraph that has been masked.

Raises `ValueError` – Mask must be a 1D boolean array of the same number of entries as points in this PointDirectedGraph.

from_vector(*vector*)

Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **object** (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

get_adjacency_list()

Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returns **adjacency_list** (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

h_points()

Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type `type(self)`

has_cycles()

Checks if the graph has at least one cycle.

Returns **has_cycles** (*bool*) – `True` if the graph has cycles.

has_isolated_vertices()

Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returns **has_isolated_vertices** (*bool*) – `True` if the graph has at least one isolated vertex.

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

init_2d_grid(*shape*, *spacing=None*, *adjacency_matrix=None*, *skip_checks=False*)

Create a PointGraph that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

If no adjacency matrix is provided, the default connectivity will be a 4-connected lattice.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple*

is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.

- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.

  The adjacency matrix of an undirected graph must be symmetric.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if no adjacency matrix is provided.

Returns **pgraph** (*PointGraph*) – A pointgraph arranged in a grid.

init_from_depth_image(*depth_image*, *spacing=None*, *adjacency_matrix=None*, *skip_checks=False*)

Return a 3D point graph from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

If no adjacency matrix is provided, the default connectivity will be a 4-connected lattice.

Parameters

- **depth_image** (*Image* or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the graph in which the rows represent source vertices and columns represent destination vertices. The non-edges must be represented with zeros and the edges can have a weight value.

  The adjacency matrix of an undirected graph must be symmetric.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if no adjacency matrix is provided.

Returns **depth_cloud** (`type(cls)`) – A new 3D PointGraph with unit XY coordinates and the given depth values as Z coordinates.

init_from_edges(*points*, *edges*, *copy=True*, *skip_checks=False*)

Construct a PointGraph from edges array.

Parameters

- **points** (`(n_vertices, n_dims, )` *ndarray*) – The array of point locations.
- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following undirected graph

```
|---0---|
|       |
|       |
```

```
1-------2
|       |
|       |
3-------4
|
|
5
```

can be defined as

```
from menpo.shape import PointUndirectedGraph
import numpy as np
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
edges = np.array([[0, 1], [1, 0], [0, 2], [2, 0], [1, 2], [2, 1],
                  [1, 3], [3, 1], [2, 4], [4, 2], [3, 4], [4, 3],
                  [3, 5], [5, 3]])
graph = PointUndirectedGraph.init_from_edges(points, edges)
```

The following directed graph

```
|-->0<--|
|       |
|       |
1<----->2
|       |
v       v
3------>4
|
v
5
```

can be represented as

```
from menpo.shape import PointDirectedGraph
import numpy as np
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
edges = np.array([[1, 0], [2, 0], [1, 2], [2, 1], [1, 3], [2, 4],
                  [3, 4], [3, 5]])
graph = PointDirectedGraph.init_from_edges(points, edges)
```

Finally, the following graph with isolated vertices

```
    0---|
        |
        |
1       2
        |
        |
3-------4


5
```

can be defined as

```
from menpo.shape import PointUndirectedGraph
import numpy as np
points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                   [0, 0]])
edges = np.array([[0, 2], [2, 0], [2, 4], [4, 2], [3, 4], [4, 3]])
graph = PointUndirectedGraph.init_from_edges(points, edges)
```

is_edge(*vertex_1*, *vertex_2*, *skip_checks=False*)
Whether there is an edge between the provided vertices.

Parameters
- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.
- **skip_checks** (*bool*, optional) – If `False`, the given vertices will be checked.

Returns **is_edge** (*bool*) – `True` if there is an edge connecting `vertex_1` and `vertex_2`.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

is_tree()
Checks if the graph is tree.

Returns **is_true** (*bool*) – If the graph is a tree.

isolated_vertices()
Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.

Returns **isolated_vertices** (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

n_children(*vertex*, *skip_checks=False*)
Returns the number of children of the selected vertex.

Parameters **vertex** (*int*) – The selected vertex.

Returns
- **n_children** (*int*) – The number of children.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_parents(*vertex*, *skip_checks=False*)
Returns the number of parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **n_parents** (*int*) – The number of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_paths(*start*, *end*)
Returns the number of all the paths (without cycles) existing from start vertex to end vertex.

Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returns **paths** (*int*) – The paths' numbers.

norm(***kwargs*)
Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returns **norm** (*float*) – The norm of this `PointCloud`

parents(*vertex*, *skip_checks=False*)
Returns the parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns parents (*list*) – The list of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

range(*boundary=0*)

The range of the extent of the PointCloud.

Parameters **boundary** (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returns range (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

relative_location_edge(*parent*, *child*)

Returns the relative location between the provided vertices. That is if vertex j is the parent and vertex i is its child and vector l denotes the coordinates of a vertex, then

```
l_i - l_j = [[x_i], [y_i]] - [[x_j], [y_j]] =
          = [[x_i - x_j], [y_i - y_j]]
```

Parameters
- **parent** (*int*) – The first selected vertex which is considered as the parent.
- **child** (*int*) – The second selected vertex which is considered as the child.

Returns relative_location (`(2,)` *ndarray*) – The relative location vector.

Raises `ValueError` – Vertices `parent` and `child` are not connected with an edge.

relative_locations()

Returns the relative location between the vertices of each edge. If vertex j is the parent and vertex i is its child and vector l denotes the coordinates of a vertex, then:

```
l_i - l_j = [[x_i], [y_i]] - [[x_j], [y_j]] =
          = [[x_i - x_j], [y_i - y_j]]
```

Returns relative_locations (`(n_vertexes, 2)` *ndarray*) – The relative locations vector.

tojson()

Convert this PointGraph to a dictionary representation suitable for inclusion in the LJSON landmark format.

Returns json (*dict*) – Dictionary with `points` and `connectivity` keys.

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualization of the PointGraph using an interactive widget.

Parameters
- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

has_landmarks

Whether the object has landmarks.

Type *bool*

landmarks

The landmarks object.

Type `LandmarkManager`

n_dims
The number of dimensions in the pointcloud.
Type*int*

n_edges
Returns the number of edges.
Type*int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_points
The number of points in the pointcloud.
Type*int*

n_vertices
Returns the number of vertices.
Type*int*

vertices
Returns the *list* of vertices.
Type*list*

PointTree

class menpo.shape.**PointTree** (*points*, *adjacency_matrix*, *root_vertex*, *copy=True*, *skip_checks=False*)
Bases: `PointDirectedGraph`, `Tree`

Class for defining a Tree with geometry.

Parameters

- **points** ((n_vertices, n_dims) *ndarray*) – The array representing the points.
- **adjacency_matrix** ((n_vertices, n_vertices) *ndarray* or *csr_matrix*) – The adjacency matrix of the tree in which the rows represent parents and columns represent children. The non-edges must be represented with zeros and the edges can have a weight value.
  NoteA tree must not have isolated vertices.
- **root_vertex** (*int*) – The vertex to be set as root.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Raises

- `ValueError` – A point for each graph vertex needs to be passed. Got {n_points} points instead of {n_vertices}.
- `ValueError` – adjacency_matrix must be either a numpy.ndarray or a scipy.sparse.csr_matrix.
- `ValueError` – Graph must have at least two vertices.
- `ValueError` – adjacency_matrix must be square (n_vertices, n_vertices,), ({adjacency_matrix.shape[0]}, {adjacency_matrix.shape[1]}) given instead.
- `ValueError` – The provided edges do not represent a tree.
- `ValueError` – The root_vertex must be in the range `[0, n_vertices - 1]`.
- `ValueError` – The combination of adjacency matrix and root vertex is not valid. BFS returns a different tree.

Examples

The following tree

```
       0
       |
    ___|___
   1       2
   |       |
  _|_      |
 3   4     5
 |   |     |
 |   |     |
 6   7     8
```

can be defined as

```
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0, 0, 0, 0],
                             [0, 0, 0, 1, 1, 0, 0, 0, 0],
                             [0, 0, 0, 0, 0, 1, 0, 0, 0],
                             [0, 0, 0, 0, 0, 0, 1, 0, 0],
                             [0, 0, 0, 0, 0, 0, 0, 1, 0],
                             [0, 0, 0, 0, 0, 0, 0, 0, 1],
                             [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]])
points = np.array([[30, 30], [10, 20], [50, 20], [0, 10], [20, 10],
                   [50, 10], [0, 0], [20, 0], [50, 0]])
tree = PointTree(points, adjacency_matrix, root_vertex=0)
```

or

```
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 8, ([0, 0, 1, 1, 2, 3, 4, 5],
                                         [1, 2, 3, 4, 5, 6, 7, 8])),
                              shape=(9, 9))
points = np.array([[30, 30], [10, 20], [50, 20], [0, 10], [20, 10],
                   [50, 10], [0, 0], [20, 0], [50, 0]])
tree = PointTree(points, adjacency_matrix, root_vertex=0)
```

_view_2d(*figure_id=None, new_figure=False, image_view=True, render_lines=True, line_colour='r', line_style='-', line_width=1.0, render_markers=True, marker_style='o', marker_size=5, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_axes=True, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8), label=None*)

Visualization of the PointGraph in 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.

- **image_view** (*bool*, optional) – If `True` the PointGraph will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{'-', '--', '-.', ':'}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.

- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointGraph as a percentage of the PointGraph's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*,*float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointGraph as a percentage of the PointGraph's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (`PointGraphViewer2d`) – The viewer object.

`_view_landmarks_2d`(*group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05*, *1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10*, *8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **`group`** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **`with_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.

- **without_labels** (None or *str* or *list* of *str*, optional) – If not None, show all except the given label(s). Should **not** be used with the with_labels kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **image_view** (*bool*, optional) – If True the PointCloud will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (See Below, optional) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (See Below, optional) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (See Below, optional) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (See Below, optional) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If True, the landmarks will be numbered.
- **numbers_horizontal_align** ({center, right, left}, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** ({center, top, bottom, baseline}, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (See Below, optional) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
- **numbers_font_weight** (See Below, optional) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (`See Below, optional`) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (`See below, optional`) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (`See Below, optional`) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).

- **render_axes** (*bool*, optional) – If True, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Raises

- ValueError – If both with_labels and without_labels are passed.
- ValueError – If the landmark manager doesn't contain the provided group label.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Returnsbounding_box (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds (*boundary=0*)
The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters **boundary** (*float*) – A optional padding distance that is added to the bounds. Default is `0`, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** (`(n_dims,)` *ndarray*) – The minimum extent of the `PointCloud` and boundary along each dimension
- **max_b** (`(n_dims,)` *ndarray*) – The maximum extent of the `PointCloud` and boundary along each dimension

centre ()
The mean of all the points in this PointCloud (centre of mass).

Returns **centre** (`(n_dims)` *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds ()
The centre of the absolute bounds of this PointCloud. Contrast with `centre()`, which is the mean point position.

Returns **centre** (`n_dims` *ndarray*) – The centre of the bounds of this PointCloud.

children (*vertex*, *skip_checks=False*)
Returns the children of the selected vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **children** (*list*) – The list of children.

Raises `ValueError` – The vertex must be between 0 and {n_vertices-1}.

constrain_to_bounds (*bounds*)
Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters **bounds** (`(n_dims, n_dims)` tuple of scalars) – The bounds to constrain this pointcloud within.

Returns **constrained** (`PointCloud`) – The constrained pointcloud.

copy ()
Generate an efficient copy of this object.

Note that Numpy arrays and other `Copyable` objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

depth_of_vertex (*vertex*, *skip_checks=False*)
Returns the depth of the specified vertex.

Parameters

- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **depth** (*int*) – The depth of the selected vertex.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

distance_to (*pointcloud*, ***kwargs*)
Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters`pointcloud` (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returnsdistance_matrix (`(n_points, n_points)` *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. `distance_matrix[i, j]` is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

`find_all_paths` (*start*, *end*, *path=[]*)

Returns a list of lists with all the paths (without cycles) found from start vertex to end vertex.

Parameters

- **`start`** (*int*) – The vertex from which the paths start.
- **`end`** (*int*) – The vertex from which the paths end.
- **`path`** (*list*, optional) – An existing path to append to.

Returnspaths (*list* of *list*) – The list containing all the paths from start to end.

`find_all_shortest_paths` (*algorithm='auto'*, *unweighted=False*)

Returns the distances and predecessors arrays of the graph's shortest paths.

Parameters

- **`algorithm`** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

- **`unweighted`** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path between each vertex such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

- **distances** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of distances between all graph vertices. `distances[i,j]` gives the shortest distance from vertex `i` to vertex `j` along the graph.
- **predecessors** (`(n_vertices, n_vertices,)` *ndarray*) – The matrix of predecessors, which can be used to reconstruct the shortest paths. Each entry `predecessors[i, j]` gives the index of the previous vertex in the path from vertex `i` to vertex `j`. If no path exists between vertices `i` and `j`, then `predecessors[i, j] = -9999`.

`find_path` (*start*, *end*, *method='bfs'*, *skip_checks=False*)

Returns a *list* with the first path (without cycles) found from the `start` vertex to the `end` vertex. It can employ either depth-first search or breadth-first search.

Parameters

- **`start`** (*int*) – The vertex from which the path starts.
- **`end`** (*int*) – The vertex to which the path ends.
- **`method`** ({`bfs`, `dfs`}, optional) – The method to be used.
- **`skip_checks`** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returnspath (*list*) – The path's vertices.

Raises`ValueError` – Method must be either bfs or dfs.

`find_shortest_path` (*start*, *end*, *algorithm='auto'*, *unweighted=False*, *skip_checks=False*)

Returns a *list* with the shortest path (without cycles) found from `start` vertex to `end` vertex.

Parameters

- **`start`** (*int*) – The vertex from which the path starts.
- **`end`** (*int*) – The vertex to which the path ends.

•**algorithm** (*'str', see below, optional*) – The algorithm to be used. Possible options are:

'dijkstra'	Dijkstra's algorithm with Fibonacci heaps
'bellman-ford'	Bellman-Ford algorithm
'johnson'	Johnson's algorithm
'floyd-warshall'	Floyd-Warshall algorithm
'auto'	Select the best among the above

•**unweighted** (*bool*, optional) – If `True`, then find unweighted distances. That is, rather than finding the path such that the sum of weights is minimized, find the path such that the number of edges is minimized.

•**skip_checks** (*bool*, optional) – If `True`, then input arguments won't pass through checks. Useful for efficiency.

Returns

•**path** (*list*) – The shortest path's vertices, including `start` and `end`. If there was not path connecting the vertices, then an empty *list* is returned.

•**distance** (*int* or *float*) – The distance (cost) of the path from `start` to `end`.

from_mask(*mask*)

A 1D boolean array with the same number of elements as the number of points in the *PointTree*. This is then broadcast across the dimensions of the *PointTree* and returns a new *PointTree* containing only those points that were `True` in the mask.

Parametersmask (`(n_points,)` *ndarray*) – 1D array of booleans

Returnspointtree (*PointTree*) – A new pointtree that has been masked.

Raises

•`ValueError` – Mask must be a 1D boolean array of the same number of entries as points in this PointTree.

•`ValueError` – Cannot remove root vertex.

from_vector(*vector*)

Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returnsobject (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of this object

get_adjacency_list()

Returns the adjacency list of the graph, i.e. a *list* of length `n_vertices` that for each vertex has a *list* of the vertex neighbours. If the graph is directed, the neighbours are children.

Returnsadjacency_list (*list* of *list* of length `n_vertices`) – The adjacency list of the graph.

h_points()

Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type`type(self)`

has_cycles()

Checks if the graph has at least one cycle.

Returns has_cycles (*bool*) – `True` if the graph has cycles.

has_isolated_vertices()

Whether the graph has any isolated vertices, i.e. vertices with no edge connections.

Returns has_isolated_vertices (*bool*) – `True` if the graph has at least one isolated vertex.

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns has_nan_values (*bool*) – If the vectorized object contains `nan` values.

classmethod init_2d_grid(*shape*, *spacing=None*, *adjacency_matrix=None*, *root_vertex=None*, *skip_checks=False*)

Create a pointtree that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

The default connectivity is the minimum spanning tree formed from a triangulation of the grid. The default root will be the centre of the grid.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the tree in which the rows represent parents and columns represent children. The non-edges must be represented with zeros and the edges can have a weight value.

  Note A tree must not have isolated vertices.
- **root_vertex** (*int*) – The vertex to be set as root.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if an adjacency matrix is provided.

Returns shape_cls (*type(cls)*) – A PointCloud or subclass arranged in a grid.

classmethod init_from_depth_image(*depth_image*, *spacing=None*, *adjacency_matrix=None*, *root_vertex=None*, *skip_checks=False*)

Return a 3D point cloud from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

The default connectivity is the minimum spanning tree formed from a triangulation of the grid. The default root will be the centre of the grid (for an unmasked image), otherwise it will be the first pixel in the masked are of the image.

Parameters

- **depth_image** (*Image* or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **adjacency_matrix** (`(n_vertices, n_vertices)` *ndarray* or *csr_matrix*, optional) – The adjacency matrix of the tree in which the rows represent parents and columns represent children. The non-edges must be

represented with zeros and the edges can have a weight value.

NoteA tree must not have isolated vertices.

- **root_vertex** (*int*) – The vertex to be set as root.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed. Only considered if an adjacency matrix is provided.

Returnsdepth_cloud (`type(cls)`) – A new 3D PointCloud with unit XY coordinates and the given depth values as Z coordinates.

classmethod init_from_edges(*points*, *edges*, *root_vertex*, *copy=True*, *skip_checks=False*)

Construct a `PointTree` from edges array.

Parameters

- **points** (`(n_vertices, n_dims, )` *ndarray*) – The array of point locations.
- **edges** (`(n_edges, 2, )` *ndarray*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.
- **root_vertex** (*int*) – That vertex that will be set as root.
- **copy** (*bool*, optional) – If `False`, the `adjacency_matrix` will not be copied on assignment.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Examples

The following tree

```
       0
       |
    ___|___
   1       2
   |       |
  _|_      |
 3   4     5
 |   |     |
 |   |     |
 6   7     8
```

can be defined as

```
from menpo.shape import PointTree
import numpy as np
points = np.array([[30, 30], [10, 20], [50, 20], [0, 10], [20, 10],
                   [50, 10], [0, 0], [20, 0], [50, 0]])
edges = np.array([[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [3, 6],
                  [4, 7], [5, 8]])
tree = PointTree.init_from_edges(points, edges, root_vertex=0)
```

is_edge(*vertex_1*, *vertex_2*, *skip_checks=False*)

Whether there is an edge between the provided vertices.

Parameters

- **vertex_1** (*int*) – The first selected vertex. Parent if the graph is directed.
- **vertex_2** (*int*) – The second selected vertex. Child if the graph is directed.
- **skip_checks** (*bool*, optional) – If `False`, the given vertices will be checked.

Returnsis_edge (*bool*) – `True` if there is an edge connecting `vertex_1` and `vertex_2`.

Raises`ValueError` – The vertex must be between 0 and {n_vertices-1}.

is_leaf(*vertex*, *skip_checks=False*)

Whether the vertex is a leaf.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **is_leaf** (*bool*) – If `True`, then selected vertex is a leaf.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

is_tree()

Checks if the graph is tree.

Returns **is_true** (*bool*) – If the graph is a tree.

isolated_vertices()

Returns the isolated vertices of the graph (if any), i.e. the vertices that have no edge connections.

Returns **isolated_vertices** (*list*) – A *list* of the isolated vertices. If there aren't any, it returns an empty *list*.

n_children(*vertex*, *skip_checks=False*)

Returns the number of children of the selected vertex.

Parameters **vertex** (*int*) – The selected vertex.

Returns
- **n_children** (*int*) – The number of children.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_parents(*vertex*, *skip_checks=False*)

Returns the number of parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **n_parents** (*int*) – The number of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

n_paths(*start*, *end*)

Returns the number of all the paths (without cycles) existing from start vertex to end vertex.

Parameters
- **start** (*int*) – The vertex from which the paths start.
- **end** (*int*) – The vertex from which the paths end.

Returns **paths** (*int*) – The paths' numbers.

n_vertices_at_depth(*depth*)

Returns the number of vertices at the specified depth.

Parameters **depth** (*int*) – The selected depth.

Returns **n_vertices** (*int*) – The number of vertices that lie in the specified depth.

norm(***kwargs*)

Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returns **norm** (*float*) – The norm of this `PointCloud`

parent(*vertex*, *skip_checks=False*)

Returns the parent of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **parent** (*int*) – The parent vertex.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

parents (*vertex*, *skip_checks=False*)
Returns the parents of the selected vertex.

Parameters
- **vertex** (*int*) – The selected vertex.
- **skip_checks** (*bool*, optional) – If `False`, the given vertex will be checked.

Returns **parents** (*list*) – The list of parents.

Raises `ValueError` – The vertex must be in the range `[0, n_vertices - 1]`.

range (*boundary=0*)
The range of the extent of the PointCloud.

Parameters **boundary** (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returns **range** (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

relative_location_edge (*parent*, *child*)
Returns the relative location between the provided vertices. That is if vertex j is the parent and vertex i is its child and vector l denotes the coordinates of a vertex, then

```
l_i - l_j = [[x_i], [y_i]] - [[x_j], [y_j]] =
          = [[x_i - x_j], [y_i - y_j]]
```

Parameters
- **parent** (*int*) – The first selected vertex which is considered as the parent.
- **child** (*int*) – The second selected vertex which is considered as the child.

Returns **relative_location** (`(2,)` *ndarray*) – The relative location vector.

Raises `ValueError` – Vertices `parent` and `child` are not connected with an edge.

relative_locations ()
Returns the relative location between the vertices of each edge. If vertex j is the parent and vertex i is its child and vector l denotes the coordinates of a vertex, then:

```
l_i - l_j = [[x_i], [y_i]] - [[x_j], [y_j]] =
          = [[x_i - x_j], [y_i - y_j]]
```

Returns **relative_locations** (`(n_vertexes, 2)` *ndarray*) – The relative locations vector.

tojson ()
Convert this PointGraph to a dictionary representation suitable for inclusion in the LJSON landmark format.

Returns **json** (*dict*) – Dictionary with `points` and `connectivity` keys.

vertices_at_depth (*depth*)
Returns a list of vertices at the specified depth.

Parameters **depth** (*int*) – The selected depth.

Returns **vertices** (*list*) – The vertices that lie in the specified depth.

view_widget (*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)
Visualization of the PointGraph using an interactive widget.

Parameters
- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

has_landmarks
Whether the object has landmarks.
Type*bool*

landmarks
The landmarks object.
Type*LandmarkManager*

leaves
Returns a *list* with the all leaves of the tree.
Type*list*

maximum_depth
Returns the maximum depth of the tree.
Type*int*

n_dims
The number of dimensions in the pointcloud.
Type*int*

n_edges
Returns the number of edges.
Type*int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_leaves
Returns the number of leaves of the tree.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_points
The number of points in the pointcloud.
Type*int*

n_vertices
Returns the number of vertices.
Type*int*

vertices
Returns the *list* of vertices.
Type*list*

2.8.5 Predefined Graphs

empty_graph

menpo.shape.**empty_graph**(*shape*, *return_pointgraph=True*)
Returns an empty graph given the landmarks configuration of a shape instance.
Parameters
- **shape** (*PointCloud* or *LandmarkGroup* or subclass) – The shape instance that defines the landmarks configuration based on which the graph will be created.

- **return_pointgraph** (*bool*, optional) – If `True`, then a *PointUndirectedGraph* instance will be returned. If `False`, then an *UndirectedGraph* instance will be returned.

Returns graph (*UndirectedGraph* or *PointUndirectedGraph*) – The generated graph.

star_graph

menpo.shape.**star_graph**(*shape*, *root_vertex*, *graph_cls=<class 'menpo.shape.graph.PointTree'>*)

Returns a star graph given the landmarks configuration of a shape instance.

Parameters
- **shape** (*PointCloud* or *LandmarkGroup* or subclass) – The shape instance that defines the landmarks configuration based on which the graph will be created.
- **root_vertex** (*int*) – The root of the star tree.
- **graph_cls** (*Graph* or *PointGraph* subclass) – The output graph type. Possible options are

```
{:map:`UndirectedGraph`, :map:`DirectedGraph`, :map:`Tree`,
 :map:`PointUndirectedGraph`, :map:`PointDirectedGraph`,
 :map:`PointTree`}
```

Returns graph (*Graph* or *PointGraph* subclass) – The generated graph.

Raises `ValueError` – graph_cls must be UndirectedGraph, DirectedGraph, Tree, PointUndirectedGraph, PointDirectedGraph or PointTree.

complete_graph

menpo.shape.**complete_graph**(*shape*, *graph_cls=<class 'menpo.shape.graph.PointUndirectedGraph'>*)

Returns a complete graph given the landmarks configuration of a shape instance.

Parameters
- **shape** (*PointCloud* or *LandmarkGroup* or subclass) – The shape instance that defines the landmarks configuration based on which the graph will be created.
- **graph_cls** (*Graph* or *PointGraph* subclass) – The output graph type. Possible options are

```
{:map:`UndirectedGraph`, :map:`DirectedGraph`,
 :map:`PointUndirectedGraph`, :map:`PointDirectedGraph`}
```

Returns graph (*Graph* or *PointGraph* subclass) – The generated graph.

Raises `ValueError` – graph_cls must be UndirectedGraph, DirectedGraph, PointUndirectedGraph or PointDirectedGraph.

chain_graph

menpo.shape.**chain_graph**(*shape*, *graph_cls=<class 'menpo.shape.graph.PointDirectedGraph'>*, *closed=False*)

Returns a chain graph given the landmarks configuration of a shape instance.

Parameters
- **shape** (*PointCloud* or *LandmarkGroup* or subclass) – The shape instance that defines the landmarks configuration based on which the graph will be created.
- **graph_cls** (*Graph* or *PointGraph* subclass) – The output graph type. Possible options are

```
{:map:`UndirectedGraph`, :map:`DirectedGraph`, :map:`Tree`,
 :map:`PointUndirectedGraph`, :map:`PointDirectedGraph`,
 :map:`PointTree`}
```

- **closed** (*bool*, optional) – If `True`, then the chain will be closed (i.e. edge between the first and last vertices).

Returns graph (*Graph* or *PointGraph* subclass) – The generated graph.

Raises

- `ValueError` – A closed chain graph cannot be a Tree or PointTree instance.
- `ValueError` – graph_cls must be UndirectedGraph, DirectedGraph, Tree, PointUndirectedGraph, PointDirectedGraph or PointTree.

delaunay_graph

`menpo.shape.`**`delaunay_graph`**(*shape*, *return_pointgraph=True*)

Returns a graph with the edges being generated by Delaunay triangulation.

Parameters

- **shape** (`PointCloud` or `LandmarkGroup` or subclass) – The shape instance that defines the landmarks configuration based on which the graph will be created.
- **return_pointgraph** (*bool*, optional) – If `True`, then a `PointUndirectedGraph` instance will be returned. If `False`, then an `UndirectedGraph` instance will be returned.

Returns graph (`UndirectedGraph` or `PointUndirectedGraph`) – The generated graph.

2.8.6 Triangular Meshes

TriMesh

class `menpo.shape.`**`TriMesh`**(*points*, *trilist=None*, *copy=True*)

Bases: `PointCloud`

A `PointCloud` with a connectivity defined by a triangle list. These are designed to be explicitly 2D or 3D.

Parameters

- **points** (`(n_points, n_dims)` *ndarray*) – The array representing the points.
- **trilist** (`(M, 3)` *ndarray* or `None`, optional) – The triangle list. If *None*, a Delaunay triangulation of the points will be used instead.
- **copy** (*bool*, optional) – If `False`, the points will not be copied on assignment. Any trilist will also not be copied. In general this should only be used if you know what you are doing.

`_view_2d`(*figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour='r'*, *line_style='-'*, *line_width=1.0*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour='k'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*, *label=None*)

Visualization of the TriMesh in 2D.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.

- **image_view** (*bool*, optional) – If `True` the TriMesh will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.

- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({`normal`, `italic`, `oblique`}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the TriMesh as a percentage of the TriMesh's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the TriMesh as a percentage of the TriMesh's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (`PointGraphViewer2d`) – The viewer object.

`_view_landmarks_2d`(*group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05, 1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **`group`** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **`with_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.

- **without_labels** (None or *str* or *list* of *str*, optional) – If not None, show all except the given label(s). Should **not** be used with the with_labels kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **image_view** (*bool*, optional) – If True the PointCloud will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If True, the landmarks will be numbered.
- **numbers_horizontal_align** ({center, right, left}, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** ({center, top, bottom, baseline}, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (`See Below, optional`) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (`See below, optional`) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (`See Below, optional`) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None` optional) – The size of the figure in inches.

Raises

- `ValueError` – If both `with_labels` and `without_labels` are passed.
- `ValueError` – If the landmark manager doesn't contain the provided group label.

as_pointgraph(*copy=True*, *skip_checks=False*)

Converts the TriMesh to a `PointUndirectedGraph`.

Parameters

- **copy** (*bool*, optional) – If `True`, the graph will be a copy.
- **skip_checks** (*bool*, optional) – If `True`, no checks will be performed.

Returnspointgraph (`PointUndirectedGraph`) – The point graph.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

boundary_tri_index()

Boolean index into triangles that are at the edge of the TriMesh

Returnsboundary_tri_index (`(n_tris,)` *ndarray*) – For each triangle (ABC), returns whether any of it's edges is not also an edge of another triangle (and so this triangle exists on the boundary of the TriMesh)

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
```

```
v    |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|    ^
|    |
v    |
0-->1
```

Returns **bounding_box** (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds (*boundary=0*)

The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters **boundary** (*float*) – A optional padding distance that is added to the bounds. Default is `0`, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** (`(n_dims,)` *ndarray*) – The minimum extent of the *PointCloud* and boundary along each dimension
- **max_b** (`(n_dims,)` *ndarray*) – The maximum extent of the *PointCloud* and boundary along each dimension

centre ()

The mean of all the points in this PointCloud (centre of mass).

Returns **centre** (`(n_dims)` *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds ()

The centre of the absolute bounds of this PointCloud. Contrast with `centre()`, which is the mean point position.

Returns **centre** (`n_dims` *ndarray*) – The centre of the bounds of this PointCloud.

constrain_to_bounds (*bounds*)

Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters **bounds** (`(n_dims, n_dims)` tuple of scalars) – The bounds to constrain this pointcloud within.

Returns **constrained** (*PointCloud*) – The constrained pointcloud.

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

distance_to (*pointcloud*, ***kwargs*)

Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters **pointcloud** (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returnsdistance_matrix (`(n_points, n_points)` *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. `distance_matrix[i, j]` is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

`edge_indices`()

An unordered index into points that rebuilds the edges of this `TriMesh`.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_indices()` for a single index for each physical edge on the `TriMesh`.

Returnsedge_indices (`(n_tris * 3, 2)` *ndarray*) – For each triangle (ABC), returns the pair of point indices that rebuild AB, AC, BC. All edge indices are concatenated for a total of `n_tris * 3` edge_indices. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

`edge_lengths`()

The length of each edge in this `TriMesh`.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_indices()` for a single index for each physical edge on the `TriMesh`. The ordering matches the case for edges and edge_indices.

Returnsedge_lengths (`(n_tris * 3, )` *ndarray*) – Scalar euclidean lengths for each edge in this `TriMesh`.

`edge_vectors`()

A vector of edges of each triangle face.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_vectors()` for a single vector for each physical edge on the `TriMesh`.

Returnsedges (`(n_tris * 3, n_dims)` *ndarray*) – For each triangle (ABC), returns the edge vectors AB, BC, CA. All edges are concatenated for a total of `n_tris * 3` edges. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

`from_mask`(*mask*)

A 1D boolean array with the same number of elements as the number of points in the TriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those points that were `True` in the mask.

Parametersmask (`(n_points,)` *ndarray*) – 1D array of booleans

Returnsmesh (`TriMesh`) – A new mesh that has been masked.

`from_tri_mask`(*tri_mask*)

A 1D boolean array with the same number of elements as the number of triangles in the TriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those triangles that were `True` in the mask.

Parametersmask (`(n_tris,)` *ndarray*) – 1D array of booleans

Returnsmesh (`TriMesh`) – A new mesh that has been masked by triangles.

`from_vector`(*vector*)

Build a new instance of the object from it's vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a `deepcopy` of the object followed by a call to `from_vector_inplace()`. This method can be overridden for a performance benefit if desired.

Parametersvector (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returnsobject (`type(self)`) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, `from_vector`.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

h_points()
Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type `type(self)`

has_nan_values()
Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

classmethod init_2d_grid(*shape*, *spacing=None*)
Create a TriMesh that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

The triangulation will be right-handed and the diagonal will go from the top left to the bottom right of a square on the grid.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.

Returns **trimesh** (`TriMesh`) – A TriMesh arranged in a grid.

classmethod init_from_depth_image(*depth_image*)
Return a 3D triangular mesh from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

Parameters **depth_image** (`Image` or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.

Returns **depth_cloud** (`type(cls)`) – A new 3D TriMesh with unit XY coordinates and the given depth values as Z coordinates. The trilist is constructed as in `init_2d_grid()`.

mean_edge_length(*unique=True*)
The mean length of each edge in this `TriMesh`.

Parameters **unique** (*bool*, optional) – If `True`, each shared edge will only be counted once towards the average. If false, shared edges will be counted twice.

Returns **mean_edge_length** (`float`) – The mean length of each edge in this `TriMesh`

mean_tri_area()
The mean area of each triangle face in this `TriMesh`.

Returns **mean_tri_area** (`float`) – The mean area of each triangle face in this `TriMesh`

Raises `ValueError` – If mesh is not 3D

norm(***kwargs*)
Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returnsnorm (*float*) – The norm of this `PointCloud`

range(*boundary=0*)

The range of the extent of the PointCloud.

Parametersboundary (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returnsrange (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

tojson()

Convert this `TriMesh` to a dictionary representation suitable for inclusion in the LJSON landmark format. Note that this enforces a simpler representation, and as such is not suitable for a permanent serialization of a `TriMesh` (to be clear, `TriMesh`'s serialized as part of a landmark set will be rebuilt as a `PointUndirectedGraph`).

Returnsjson (*dict*) – Dictionary with `points` and `connectivity` keys.

tri_areas()

The area of each triangle face.

Returnsareas (`(n_tris,)` *ndarray*) – Area of each triangle, ordered as the trilist is

Raises`ValueError` – If mesh is not 2D or 3D

tri_normals()

Compute the triangle face normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returnsnormals (`(n_tris, 3)` *ndarray*) – Normal at each triangle face.

Raises`ValueError` – If mesh is not 3D

unique_edge_indices()

An unordered index into points that rebuilds the unique edges of this `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returnsunique_edge_indices (`(n_unique_edges, 2)` *ndarray*) – Return a point index that rebuilds all edges present in this `TriMesh` only once.

unique_edge_lengths()

The length of each edge in this `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returnsedge_lengths (`(n_tris * 3, )` *ndarray*) – Scalar euclidean lengths for each edge in this `TriMesh`.

unique_edge_vectors()

An unordered vector of unique edges for the whole `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returnsunique_edge_vectors (`(n_unique_edges, n_dims)` *ndarray*) – Vectors for each unique edge in this `TriMesh`.

vertex_normals()

Compute the per-vertex normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returnsnormals (`(n_points, 3)` *ndarray*) – Normal at each point.

Raises`ValueError` – If mesh is not 3D

view_widget (*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)
Visualization of the TriMesh using an interactive widget.

Parameters

- **browser_style** ({'buttons', 'slider'}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **style** ({'coloured', 'minimal'}, optional) – If 'coloured', then the style of the widget will be coloured. If minimal, then the style is simple using black and white colours.

has_landmarks
Whether the object has landmarks.
Type*bool*

landmarks
The landmarks object.
Type*LandmarkManager*

n_dims
The number of dimensions in the pointcloud.
Type*int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_points
The number of points in the pointcloud.
Type*int*

n_tris
The number of triangles in the triangle list.
Type*int*

ColouredTriMesh

class menpo.shape.**ColouredTriMesh** (*points*, *trilist=None*, *colours=None*, *copy=True*)
Bases: TriMesh

Combines a *TriMesh* with a colour per vertex.

Parameters

- **points** ((n_points, n_dims) *ndarray*) – The array representing the points.
- **trilist** ((M, 3) *ndarray* or None, optional) – The triangle list. If *None*, a Delaunay triangulation of the points will be used instead.
- **colours** ((N, 3) *ndarray*, optional) – The floating point RGB colour per vertex. If not given, grey will be assigned to each vertex.
- **copy** (*bool*, optional) – If False, the points, trilist and colours will not be copied on assignment. In general this should only be used if you know what you are doing.

RaisesValueError – If the number of colour values does not match the number of vertices.

_view_2d(*figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour='r'*, *line_style='-'*, *line_width=1.0*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour='k'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*, *label=None*)

Visualization of the TriMesh in 2D. Currently, explicit coloured TriMesh viewing is not supported, and therefore viewing falls back to uncoloured 2D TriMesh viewing.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the ColouredTriMesh will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.

•**numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
•**numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

•**numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

•**render_axes** (*bool*, optional) – If True, the axes will be rendered.
•**axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

•**axes_font_size** (*int*, optional) – The font size of the axes.
•**axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
•**axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

•**axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the TriMesh as a percentage of the TriMesh's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
•**axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the TriMesh as a percentage of the TriMesh's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
•**axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
•**axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
•**figure_size** ((*float*, *float*) *tuple* or None, optional) – The size of the figure in inches.
•**label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (PointGraphViewer2d) – The viewer object.

Raiseswarning – 2D Viewing of Coloured TriMeshes is not supported, automatically falls back to 2D *TriMesh* viewing.

_view_landmarks_2d(*group=None, with_labels=None, without_labels=None, figure_id=None, new_figure=False, image_view=True, render_lines=True, line_colour=None, line_style='-', line_width=1, render_markers=True, marker_style='o', marker_size=5, marker_face_colour=None, marker_edge_colour=None, marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_style='normal', numbers_font_weight='normal', numbers_font_colour='k', render_legend=False, legend_title='', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', legend_marker_scale=None, legend_location=2, legend_bbox_to_anchor=(1.05, 1.0), legend_border_axes_pad=None, legend_n_columns=1, legend_horizontal_spacing=None, legend_vertical_spacing=None, legend_border=True, legend_border_padding=None, legend_shadow=False, legend_rounded_corners=False, render_axes=False, axes_font_name='sans-serif', axes_font_size=10, axes_font_style='normal', axes_font_weight='normal', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **group** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **with_labels** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.
- **without_labels** (`None` or *str* or *list* of *str*, optional) – If not `None`, show all except the given label(s). Should **not** be used with the `with_labels` kwarg.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the PointCloud will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (`See Below, optional`) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (`See Below, optional`) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (`See Below, optional`) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See Below, optional*) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original

- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See Below, optional`) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (`See Below, optional`) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.

- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Raises

- ValueError – If both with_labels and without_labels are passed.
- ValueError – If the landmark manager doesn't contain the provided group label.

as_pointgraph(*copy=True*, *skip_checks=False*)

Converts the TriMesh to a *PointUndirectedGraph*.

Parameters

- **copy** (*bool*, optional) – If True, the graph will be a copy.
- **skip_checks** (*bool*, optional) – If True, no checks will be performed.

Returns pointgraph (*PointUndirectedGraph*) – The point graph.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns vector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

boundary_tri_index()

Boolean index into triangles that are at the edge of the TriMesh

Returns boundary_tri_index ((n_tris,) *ndarray*) – For each triangle (ABC), returns whether any of it's edges is not also an edge of another triangle (and so this triangle exists on the boundary of the TriMesh)

bounding_box()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Returns bounding_box (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds(*boundary=0*)

The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters boundary (*float*) – A optional padding distance that is added to the bounds. Default is 0, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** ((n_dims,) *ndarray*) – The minimum extent of the *PointCloud* and boundary along each dimension
- **max_b** ((n_dims,) *ndarray*) – The maximum extent of the *PointCloud* and boundary along each dimension

centre()

The mean of all the points in this PointCloud (centre of mass).

Returns centre ((n_dims) *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds()

The centre of the absolute bounds of this PointCloud. Contrast with *centre()*, which is the mean point position.

Returns centre (n_dims *ndarray*) – The centre of the bounds of this PointCloud.

constrain_to_bounds(*bounds*)

Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters bounds ((n_dims, n_dims) tuple of scalars) – The bounds to constrain this pointcloud within.

Returns constrained (*PointCloud*) – The constrained pointcloud.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

distance_to(*pointcloud*, ***kwargs*)

Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters pointcloud (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returns distance_matrix ((n_points, n_points) *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. distance_matrix[i, j] is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

edge_indices()

An unordered index into points that rebuilds the edges of this *TriMesh*.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider *unique_edge_indices()* for a single index for each physical edge on the *TriMesh*.

Returns edge_indices ((n_tris * 3, 2) *ndarray*) – For each triangle (ABC), returns the pair of point indices that rebuild AB, AC, BC. All edge indices are concatenated for a total of n_tris * 3 edge_indices. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

edge_lengths()

The length of each edge in this *TriMesh*.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider *unique_edge_indices()* for a single index for each physical edge on the *TriMesh*. The ordering matches the case for edges and edge_indices.

Returns edge_lengths ((n_tris * 3,) *ndarray*) – Scalar euclidean lengths for each edge in this *TriMesh*.

edge_vectors()

A vector of edges of each triangle face.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider *unique_edge_vectors()* for a single vector for each physical edge on the *TriMesh*.

Returnsedges ((n_tris * 3, n_dims) *ndarray*) – For each triangle (ABC), returns the edge vectors AB, BC, CA. All edges are concatenated for a total of n_tris * 3 edges. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

from_mask(*mask*)
A 1D boolean array with the same number of elements as the number of points in the ColouredTriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those points that were True in the mask.

Parametersmask ((n_points,) *ndarray*) – 1D array of booleans
Returnsmesh (*ColouredTriMesh*) – A new mesh that has been masked.

from_tri_mask(*tri_mask*)
A 1D boolean array with the same number of elements as the number of triangles in the TriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those triangles that were True in the mask.

Parametersmask ((n_tris,) *ndarray*) – 1D array of booleans
Returnsmesh (*TriMesh*) – A new mesh that has been masked by triangles.

from_vector(*vector*)
Build a new instance of the object from it's vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of the object.
Returnsobject (type(self)) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of this object

h_points()
Convert poincloud to a homogeneous array: (n_dims + 1, n_points)

Typetype(self)

has_nan_values()
Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

classmethod init_2d_grid(*shape*, *spacing=None*, *colours=None*)
Create a ColouredTriMesh that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. spacing optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

The triangulation will be right-handed and the diagonal will go from the top left to the bottom right of a square on the grid.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple*

is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.

- **colours** (`(N, 3)` *ndarray*, optional) – The floating point RGB colour per vertex. If not given, grey will be assigned to each vertex.

Returnstrimesh (`TriMesh`) – A TriMesh arranged in a grid.

classmethod init_from_depth_image(*depth_image*, *colours=None*)

Return a 3D textured triangular mesh from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

The optionally passed texture will be textured mapped onto the planar surface using the correct texture coordinates for an image of the same shape as `depth_image`.

Parameters

- **depth_image** (`Image` or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.
- **colours** (`(N, 3)` *ndarray*, optional) – The floating point RGB colour per vertex. If not given, grey will be assigned to each vertex.

Returnsdepth_cloud (`type(cls)`) – A new 3D TriMesh with unit XY coordinates and the given depth values as Z coordinates. The trilist is constructed as in `init_2d_grid()`.

mean_edge_length(*unique=True*)

The mean length of each edge in this `TriMesh`.

Parametersunique (*bool*, optional) – If `True`, each shared edge will only be counted once towards the average. If false, shared edges will be counted twice.

Returnsmean_edge_length (`float`) – The mean length of each edge in this `TriMesh`

mean_tri_area()

The mean area of each triangle face in this `TriMesh`.

Returnsmean_tri_area (`float`) – The mean area of each triangle face in this `TriMesh`

Raises`ValueError` – If mesh is not 3D

norm(***kwargs*)

Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returnsnorm (*float*) – The norm of this `PointCloud`

range(*boundary=0*)

The range of the extent of the PointCloud.

Parametersboundary (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returnsrange (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

tojson()

Convert this `TriMesh` to a dictionary representation suitable for inclusion in the LJSON landmark format. Note that this enforces a simpler representation, and as such is not suitable for a permanent serialization of a `TriMesh` (to be clear, `TriMesh`'s serialized as part of a landmark set will be rebuilt as a `PointUndirectedGraph`).

Returnsjson (*dict*) – Dictionary with `points` and `connectivity` keys.

tri_areas()

The area of each triangle face.

Returnsareas (`(n_tris,)` *ndarray*) – Area of each triangle, ordered as the trilist is

Raises `ValueError` – If mesh is not 2D or 3D

tri_normals()

Compute the triangle face normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returns normals (`(n_tris, 3)` *ndarray*) – Normal at each triangle face.

Raises `ValueError` – If mesh is not 3D

unique_edge_indices()

An unordered index into points that rebuilds the unique edges of this *TriMesh*.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns unique_edge_indices (`(n_unique_edges, 2)` *ndarray*) – Return a point index that rebuilds all edges present in this *TriMesh* only once.

unique_edge_lengths()

The length of each edge in this *TriMesh*.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns edge_lengths (`(n_tris * 3, )` *ndarray*) – Scalar euclidean lengths for each edge in this *TriMesh*.

unique_edge_vectors()

An unordered vector of unique edges for the whole *TriMesh*.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns unique_edge_vectors (`(n_unique_edges, n_dims)` *ndarray*) – Vectors for each unique edge in this *TriMesh*.

vertex_normals()

Compute the per-vertex normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returns normals (`(n_points, 3)` *ndarray*) – Normal at each point.

Raises `ValueError` – If mesh is not 3D

view_widget(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualization of the TriMesh using an interactive widget.

Parameters

- **browser_style** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **figure_size** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **style** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

has_landmarks

Whether the object has landmarks.

Type *bool*

landmarks

The landmarks object.

Type *LandmarkManager*

n_dims

The number of dimensions in the pointcloud.

Type *int*

n_landmark_groups
The number of landmark groups on this object.
Type*int*

n_parameters
The length of the vector that this object produces.
Type*int*

n_points
The number of points in the pointcloud.
Type*int*

n_tris
The number of triangles in the triangle list.
Type*int*

TexturedTriMesh

class menpo.shape.**TexturedTriMesh**(*points*, *tcoords*, *texture*, *trilist=None*, *copy=True*)
Bases: TriMesh

Combines a *TriMesh* with a texture. Also encapsulates the texture coordinates required to render the texture on the mesh.

Parameters

- **points** ((n_points, n_dims) *ndarray*) – The array representing the points.
- **tcoords** ((N, 2) *ndarray*) – The texture coordinates for the mesh.
- **texture** (*Image*) – The texture for the mesh.
- **trilist** ((M, 3) *ndarray* or None, optional) – The triangle list. If None, a Delaunay triangulation of the points will be used instead.
- **copy** (*bool*, optional) – If False, the points, trilist and texture will not be copied on assignment. In general this should only be used if you know what you are doing.

_view_2d(*figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour='r'*, *line_style='-'*, *line_width=1.0*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour='k'*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*, *label=None*)
Visualization of the TriMesh in 2D. Currently, explicit textured TriMesh viewing is not supported, and therefore viewing falls back to untextured 2D TriMesh viewing.

Returns

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **image_view** (*bool*, optional) – If True the TexturedTriMesh will be viewed as if it is in the image coordinate system.
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.

- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*,*float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the TriMesh as a percentage of the TriMesh's width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the TriMesh as a percentage of the TriMesh's height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or `None`, optional) – The size of the figure in inches.
- **label** (*str*, optional) – The name entry in case of a legend.

Returnsviewer (`PointGraphViewer2d`) – The viewer object.

Raises`warning` – 2D Viewing of Coloured TriMeshes is not supported, automatically falls back to 2D *`TriMesh`* viewing.

`_view_landmarks_2d`(*group=None*, *with_labels=None*, *without_labels=None*, *figure_id=None*, *new_figure=False*, *image_view=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_legend=False*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05*, *1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10*, *8)*)

Visualize the landmarks. This method will appear on the Image as `view_landmarks` if the Image is 2D.

Parameters

- **`group`** (*str* or``None`` optional) – The landmark group to be visualized. If `None` and there are more than one landmark groups, an error is raised.
- **`with_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, only show the given label(s). Should **not** be used with the `without_labels` kwarg.
- **`without_labels`** (`None` or *str* or *list* of *str*, optional) – If not `None`, show all except the given label(s). Should **not** be used with the `with_labels` kwarg.
- **`figure_id`** (*object*, optional) – The id of the figure to be used.
- **`new_figure`** (*bool*, optional) – If `True`, a new figure is created.
- **`image_view`** (*bool*, optional) – If `True` the PointCloud will be viewed as if it is in the image coordinate system.

- **render_lines** (*bool*, optional) – If `True`, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If `True`, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If `True`, the landmarks will be numbered.
- **numbers_horizontal_align** (`{center, right, left}`, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** (`{center, top, bottom, baseline}`, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** (`{normal, italic, oblique}`, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **legend_font_style** (`{normal, italic, oblique}`, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See Below, optional*) – The font weight of the legend. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*) *tuple*, optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.

- **axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the PointCloud as a percentage of the PointCloud's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_y_limits** ((*float*, *float*) *tuple* or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the PointCloud as a percentage of the PointCloud's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Raises

- ValueError – If both with_labels and without_labels are passed.
- ValueError – If the landmark manager doesn't contain the provided group label.

as_pointgraph (*copy=True*, *skip_checks=False*)

Converts the TriMesh to a *PointUndirectedGraph*.

Parameters

- **copy** (*bool*, optional) – If True, the graph will be a copy.
- **skip_checks** (*bool*, optional) – If True, no checks will be performed.

Returnspointgraph (*PointUndirectedGraph*) – The point graph.

as_vector (***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

boundary_tri_index ()

Boolean index into triangles that are at the edge of the TriMesh

Returnsboundary_tri_index ((n_tris,) *ndarray*) – For each triangle (ABC), returns whether any of it's edges is not also an edge of another triangle (and so this triangle exists on the boundary of the TriMesh)

bounding_box ()

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
```

```
|    |
v    |
0-->1
```

Returns **bounding_box** (*PointDirectedGraph*) – The axis aligned bounding box of the PointCloud.

bounds (*boundary=0*)
The minimum to maximum extent of the PointCloud. An optional boundary argument can be provided to expand the bounds by a constant margin.

Parameters **boundary** (*float*) – A optional padding distance that is added to the bounds. Default is `0`, meaning the max/min of tightest possible containing square/cube/hypercube is returned.

Returns

- **min_b** (`(n_dims,)` *ndarray*) – The minimum extent of the *PointCloud* and boundary along each dimension
- **max_b** (`(n_dims,)` *ndarray*) – The maximum extent of the *PointCloud* and boundary along each dimension

centre ()
The mean of all the points in this PointCloud (centre of mass).

Returns **centre** (`(n_dims)` *ndarray*) – The mean of this PointCloud's points.

centre_of_bounds ()
The centre of the absolute bounds of this PointCloud. Contrast with *centre()*, which is the mean point position.

Returns **centre** (`n_dims` *ndarray*) – The centre of the bounds of this PointCloud.

constrain_to_bounds (*bounds*)
Returns a copy of this PointCloud, constrained to lie exactly within the given bounds. Any points outside the bounds will be 'snapped' to lie *exactly* on the boundary.

Parameters **bounds** (`(n_dims, n_dims)` tuple of scalars) – The bounds to constrain this pointcloud within.

Returns **constrained** (*PointCloud*) – The constrained pointcloud.

copy ()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

distance_to (*pointcloud*, ***kwargs*)
Returns a distance matrix between this PointCloud and another. By default the Euclidean distance is calculated - see *scipy.spatial.distance.cdist* for valid kwargs to change the metric and other properties.

Parameters **pointcloud** (*PointCloud*) – The second pointcloud to compute distances between. This must be of the same dimension as this PointCloud.

Returns **distance_matrix** (`(n_points, n_points)` *ndarray*) – The symmetric pairwise distance matrix between the two PointClouds s.t. `distance_matrix[i, j]` is the distance between the i'th point of this PointCloud and the j'th point of the input PointCloud.

edge_indices ()
An unordered index into points that rebuilds the edges of this *TriMesh*.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_indices()` for a single index for each physical edge on the `TriMesh`.

Returns **edge_indices** (`(n_tris * 3, 2)` *ndarray*) – For each triangle (ABC), returns the pair of point indices that rebuild AB, AC, BC. All edge indices are concatenated for a total of `n_tris * 3` edge_indices. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

`edge_lengths`()

The length of each edge in this `TriMesh`.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_indices()` for a single index for each physical edge on the `TriMesh`. The ordering matches the case for edges and edge_indices.

Returns **edge_lengths** (`(n_tris * 3, )` *ndarray*) – Scalar euclidean lengths for each edge in this `TriMesh`.

`edge_vectors`()

A vector of edges of each triangle face.

Note that there will be two edges present in cases where two triangles 'share' an edge. Consider `unique_edge_vectors()` for a single vector for each physical edge on the `TriMesh`.

Returns **edges** (`(n_tris * 3, n_dims)` *ndarray*) – For each triangle (ABC), returns the edge vectors AB, BC, CA. All edges are concatenated for a total of `n_tris * 3` edges. The ordering is done so that all AB vectors are first in the returned list, followed by BC, then CA.

`from_mask`(*mask*)

A 1D boolean array with the same number of elements as the number of points in the TexturedTriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those points that were `True` in the mask.

Parameters **mask** (`(n_points,)` *ndarray*) – 1D array of booleans

Returns **mesh** (`TexturedTriMesh`) – A new mesh that has been masked.

`from_tri_mask`(*tri_mask*)

A 1D boolean array with the same number of elements as the number of triangles in the TriMesh. This is then broadcast across the dimensions of the mesh and returns a new mesh containing only those triangles that were `True` in the mask.

Parameters **mask** (`(n_tris,)` *ndarray*) – 1D array of booleans

Returns **mesh** (`TriMesh`) – A new mesh that has been masked by triangles.

`from_vector`(*flattened*)

Builds a new `TexturedTriMesh` given the *flattened* 1D vector. Note that the trilist, texture, and tcoords will be drawn from self.

Parameters **flattened** (`(N,)` *ndarray*) – Vector representing a set of points.

Returns **trimesh** (`TriMesh`) – A new trimesh created from the vector with `self` trilist.

`from_vector_inplace`(*vector*)

Deprecated. Use the non-mutating API, `from_vector`.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

`h_points`()

Convert poincloud to a homogeneous array: `(n_dims + 1, n_points)`

Type `type(self)`

`has_nan_values`()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns has_nan_values (*bool*) – If the vectorized object contains `nan` values.

classmethod `init_2d_grid`(*shape*, *spacing=None*, *tcoords=None*, *texture=None*)

Create a TexturedTriMesh that exists on a regular 2D grid. The first dimension is the number of rows in the grid and the second dimension of the shape is the number of columns. `spacing` optionally allows the definition of the distance between points (uniform over points). The spacing may be different for rows and columns.

The triangulation will be right-handed and the diagonal will go from the top left to the bottom right of a square on the grid.

If no texture is passed a blank (black) texture is attached with correct texture coordinates for texture mapping an image of the same size as `shape`.

Parameters

- **shape** (*tuple* of 2 *int*) – The size of the grid to create, this defines the number of points across each dimension in the grid. The first element is the number of rows and the second is the number of columns.
- **spacing** (*int* or *tuple* of 2 *int*, optional) – The spacing between points. If a single *int* is provided, this is applied uniformly across each dimension. If a *tuple* is provided, the spacing is applied non-uniformly as defined e.g. `(2, 3)` gives a spacing of 2 for the rows and 3 for the columns.
- **tcoords** (`(N, 2)` *ndarray*, optional) – The texture coordinates for the mesh.
- **texture** (*Image*, optional) – The texture for the mesh.

Returns trimesh (*TriMesh*) – A TriMesh arranged in a grid.

classmethod `init_from_depth_image`(*depth_image*, *tcoords=None*, *texture=None*)

Return a 3D textured triangular mesh from the given depth image. The depth image is assumed to represent height/depth values and the XY coordinates are assumed to unit spaced and represent image coordinates. This is particularly useful for visualising depth values that have been recovered from images.

The optionally passed texture will be textured mapped onto the planar surface using the correct texture coordinates for an image of the same shape as `depth_image`.

Parameters

- **depth_image** (*Image* or subclass) – A single channel image that contains depth values - as commonly returned by RGBD cameras, for example.
- **tcoords** (`(N, 2)` *ndarray*, optional) – The texture coordinates for the mesh.
- **texture** (*Image*, optional) – The texture for the mesh.

Returns depth_cloud (`type(cls)`) – A new 3D TriMesh with unit XY coordinates and the given depth values as Z coordinates. The trilist is constructed as in `init_2d_grid()`.

`mean_edge_length`(*unique=True*)

The mean length of each edge in this *TriMesh*.

Parameters unique (*bool*, optional) – If `True`, each shared edge will only be counted once towards the average. If false, shared edges will be counted twice.

Returns mean_edge_length (`float`) – The mean length of each edge in this *TriMesh*

`mean_tri_area`()

The mean area of each triangle face in this *TriMesh*.

Returns mean_tri_area (`float`) – The mean area of each triangle face in this *TriMesh*

Raises `ValueError` – If mesh is not 3D

`norm`(***kwargs*)

Returns the norm of this PointCloud. This is a translation and rotation invariant measure of the point cloud's intrinsic size - in other words, it is always taken around the point cloud's centre.

By default, the Frobenius norm is taken, but this can be changed by setting kwargs - see `numpy.linalg.norm` for valid options.

Returns norm (*float*) – The norm of this `PointCloud`

range(*boundary=0*)

The range of the extent of the PointCloud.

Parameters boundary (*float*) – A optional padding distance that is used to extend the bounds from which the range is computed. Default is `0`, no extension is performed.

Returns range (`(n_dims,)` *ndarray*) – The range of the `PointCloud` extent in each dimension.

tcoords_pixel_scaled()

Returns a `PointCloud` that is modified to be suitable for directly indexing into the pixels of the texture (e.g. for manual mapping operations). The resulting tcoords behave just like image landmarks do.

The operations that are performed are:

- Flipping the origin from bottom-left to top-left
- Scaling the tcoords by the image shape (denormalising them)
- Permuting the axis so that

Returns tcoords_scaled (`PointCloud`) – A copy of the tcoords that behave like `Image` landmarks

Examples

Recovering pixel values for every texture coordinate:

```
>>> texture = texturedtrimesh.texture
>>> tc_ps = texturedtrimesh.tcoords_pixel_scaled()
>>> pixel_values_at_tcs = texture.sample(tc_ps)
```

tojson()

Convert this `TriMesh` to a dictionary representation suitable for inclusion in the LJSON landmark format. Note that this enforces a simpler representation, and as such is not suitable for a permanent serialization of a `TriMesh` (to be clear, `TriMesh`'s serialized as part of a landmark set will be rebuilt as a `PointUndirectedGraph`).

Returns json (*dict*) – Dictionary with `points` and `connectivity` keys.

tri_areas()

The area of each triangle face.

Returns areas (`(n_tris,)` *ndarray*) – Area of each triangle, ordered as the trilist is

Raises `ValueError` – If mesh is not 2D or 3D

tri_normals()

Compute the triangle face normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returns normals (`(n_tris, 3)` *ndarray*) – Normal at each triangle face.

Raises `ValueError` – If mesh is not 3D

unique_edge_indices()

An unordered index into points that rebuilds the unique edges of this `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns unique_edge_indices (`(n_unique_edges, 2)` *ndarray*) – Return a point index that rebuilds all edges present in this `TriMesh` only once.

unique_edge_lengths()

The length of each edge in this `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns edge_lengths (`(n_tris * 3, )` *ndarray*) – Scalar euclidean lengths for each edge in this `TriMesh`.

`unique_edge_vectors`()

An unordered vector of unique edges for the whole `TriMesh`.

Note that each physical edge will only be counted once in this method (i.e. edges shared between neighbouring triangles are only counted once not twice). The ordering should be considered random.

Returns unique_edge_vectors (`(n_unique_edges, n_dims)` *ndarray*) – Vectors for each unique edge in this `TriMesh`.

`vertex_normals`()

Compute the per-vertex normals from the current set of points and triangle list. Only valid for 3D dimensional meshes.

Returns normals (`(n_points, 3)` *ndarray*) – Normal at each point.

Raises `ValueError` – If mesh is not 3D

`view_widget`(*browser_style='buttons'*, *figure_size=(10, 8)*, *style='coloured'*)

Visualization of the TriMesh using an interactive widget.

Parameters

- **`browser_style`** ({`'buttons'`, `'slider'`}, optional) – It defines whether the selector of the objects will have the form of plus/minus buttons or a slider.
- **`figure_size`** ((*int*, *int*) *tuple*, optional) – The initial size of the rendered figure.
- **`style`** ({`'coloured'`, `'minimal'`}, optional) – If `'coloured'`, then the style of the widget will be coloured. If `minimal`, then the style is simple using black and white colours.

`has_landmarks`

Whether the object has landmarks.

Type *bool*

`landmarks`

The landmarks object.

Type `LandmarkManager`

`n_dims`

The number of dimensions in the pointcloud.

Type *int*

`n_landmark_groups`

The number of landmark groups on this object.

Type *int*

`n_parameters`

The length of the vector that this object produces.

Type *int*

`n_points`

The number of points in the pointcloud.

Type *int*

`n_tris`

The number of triangles in the triangle list.

Type *int*

2.8.7 Group Operations

mean_pointcloud

menpo.shape.**mean_pointcloud**(*pointclouds*)

Compute the mean of a *list* of `PointCloud` or subclass objects. The list is assumed to be homogeneous i.e all elements of the list are assumed to belong to the same point cloud subclass just as all elements are also assumed to have the same number of points and represent semantically equivalent point clouds.

Parameters **pointclouds** (*list* of `PointCloud` or subclass) – List of point cloud or subclass objects from which we want to compute the mean.

Returns **mean_pointcloud** (`PointCloud` or subclass) – The mean point cloud or subclass.

2.8.8 Shape Building

bounding_box

menpo.shape.**bounding_box**(*closest_to_origin*, *opposite_corner*)

Return a bounding box from two corner points as a directed graph. The the first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
v   |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v   |
0-->1
```

Parameters

- **closest_to_origin** ((*float*, *float*)) – Two floats representing the coordinates closest to the origin. Represented by (0) in the graph above. For an image, this will be the top left. For a pointcloud, this will be the bottom left.
- **opposite_corner** ((*float*, *float*)) – Two floats representing the coordinates opposite the corner closest to the origin. Represented by (2) in the graph above. For an image, this will be the bottom right. For a pointcloud, this will be the top right.

Returns **bounding_box** (`PointDirectedGraph`) – The axis aligned bounding box from the two given corners.

2.9 `menpo.transform`

2.9.1 Composite Transforms

rotate_ccw_about_centre

menpo.transform.**rotate_ccw_about_centre**(*obj*, *theta*, *degrees=True*)

Return a Homogeneous Transform that implements rotating an object counter-clockwise about its centre. The given object must be transformable and must implement a method to provide the object centre.

Parameters
- **obj** (Transformable) – A transformable object that has the centre method.
- **theta** (*float*) – The angle of rotation clockwise about the origin.
- **degrees** (*bool*, optional) – If True theta is interpreted as degrees. If False, theta is interpreted as radians.

Returnstransform (*Homogeneous*) – A homogeneous transform that implements the rotation.

scale_about_centre

menpo.transform.**scale_about_centre**(*obj*, *scale*)

Return a Homogeneous Transform that implements scaling an object about its centre. The given object must be transformable and must implement a method to provide the object centre.

Parameters
- **obj** (Transformable) – A transformable object that has the centre method.
- **scale** (*float* or (n_dims,) *ndarray*) – The scale factor as defined in the Scale documentation.

Returnstransform (*Homogeneous*) – A homogeneous transform that implements the scaling.

2.9.2 Homogeneous Transforms

Homogeneous

class menpo.transform.**Homogeneous**(*h_matrix*, *copy=True*, *skip_checks=False*)

Bases: *ComposableTransform*, *Vectorizable*, *VComposable*, *VInvertible*

A simple n-dimensional homogeneous transformation.

Adds a unit homogeneous coordinate to points, performs the dot product, re-normalizes by division by the homogeneous coordinate, and returns the result.

Can be composed with another *Homogeneous*, so long as the dimensionality matches.

Parameters
- **h_matrix** ((n_dims + 1, n_dims + 1) *ndarray*) – The homogeneous matrix defining this transform.
- **copy** (*bool*, optional) – If False, avoid copying h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, avoid sanity checks on the h_matrix. Useful for performance.

apply(*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters
- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.

- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns **transformed** (`type(x)`) – The transformed object or array

apply_inplace (**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector (***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after (*transform*)

A `Transform` that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **before** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace (*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before (*transform*)

A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **after** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace (*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains nan values.

classmethod init_identity(*n_dims*)

Creates an identity matrix Homogeneous transform.

Parameters **n_dims** (*int*) – The number of dimensions.

Returns **identity** (*Homogeneous*) – The identity matrix transform.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type *Homogeneous*

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters **vector** ((n_parameters,) *ndarray*) – A vectorized version of self

Returns **pseudoinverse_vector** (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters
- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

composes_inplace_with
Homogeneous can swallow composition with any other *Homogeneous*, subclasses will have to override and be more specific.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.
Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable
Deprecated `True` iff *set_h_matrix()* is permitted on this type of transform.

If this returns `False` calls to *set_h_matrix()* will raise a `NotImplementedError`.
Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.
Type `True`

n_dims
The dimensionality of the data the transform operates on.
Type *int*

n_dims_output
The output of the data from the transform.
Type *int*

n_parameters
The length of the vector that this object produces.
Type *int*

Affine

class `menpo.transform.`**Affine**(*h_matrix*, *copy=True*, *skip_checks=False*)
Bases: `Homogeneous`

Base class for all `n`-dimensional affine transformations. Provides methods to break the transform down into its constituent scale/rotation/translation, to view the homogeneous matrix equivalent, and to chain this transform with other affine transformations.

Parameters
- **h_matrix** (`(n_dims + 1, n_dims + 1)` *ndarray*) – The homogeneous matrix of the affine transformation.
- **copy** (*bool*, optional) – If `False` avoid copying `h_matrix` for performance.
- **skip_checks** (*bool*, optional) – If `True` avoid sanity checks on `h_matrix` for performance.

apply (*x*, *batch_size=None*, ***kwargs*)

Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns **transformed** (`type(x)`) – The transformed object or array

apply_inplace (**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector (***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after (*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **before** `self`

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace (*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (composes_inplace_with) – Transform to be applied **before** self

Raises ValueError – If transform isn't an instance of composes_inplace_with

compose_before (*transform*)

A Transform that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a TransformChain as a last resort. See composes_with for a description of how the mode of composition is decided.

Parameters **transform** (Transform) – Transform to be applied **after** self

Returns **transform** (Transform or TransformChain) – If the composition was native, a single new Transform will be returned. If not, a TransformChain is returned instead.

compose_before_inplace (*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (composes_inplace_with) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of composes_inplace_with

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other Copyable objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

decompose ()

Decompose this transform into discrete Affine Transforms.

Useful for understanding the effect of a complex composite transform.

Returns

transforms (*list* of DiscreteAffine) – Equivalent to this affine transform, such that

```
reduce(lambda x, y: x.chain(y), self.decompose()) == self
```

from_vector (*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to from_vector_inplace(). This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace (*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

has_nan_values ()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

classmethod init_identity (*n_dims*)

Creates an identity matrix Affine transform.

Parameters **n_dims** (*int*) – The number of dimensions.

Returns **identity** (*Affine*) – The identity matrix transform.

pseudoinverse ()

The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type *Homogeneous*

pseudoinverse_vector (*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters **vector** (`(n_parameters,)` *ndarray*) – A vectorized version of `self`

Returns **pseudoinverse_vector** (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix (*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

composes_inplace_with

Affine can swallow composition with any other *Affine*.

composes_with

Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.
Type (n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable
Deprecated True iff *set_h_matrix()* is permitted on this type of transform.

If this returns False calls to *set_h_matrix()* will raise a NotImplementedError.
Type*bool*

has_true_inverse
The pseudoinverse is an exact inverse.
TypeTrue

linear_component
The linear component of this affine transform.
Type (n_dims, n_dims) *ndarray*

n_dims
The dimensionality of the data the transform operates on.
Type*int*

n_dims_output
The output of the data from the transform.
Type*int*

n_parameters
n_dims * (n_dims + 1) parameters - every element of the matrix but the homogeneous part.
Typeint

Examples

2D Affine: 6 parameters:

```
[p1, p3, p5]
[p2, p4, p6]
```

3D Affine: 12 parameters:

```
[p1, p4, p7, p10]
[p2, p5, p8, p11]
[p3, p6, p9, p12]
```

translation_component
The translation component of this affine transform.
Type (n_dims,) *ndarray*

Similarity

class menpo.transform.**Similarity**(*h_matrix*, *copy=True*, *skip_checks=False*)
Bases: Affine

Specialist version of an *Affine* that is guaranteed to be a Similarity transform.

Parameters
- **h_matrix** ((n_dims + 1, n_dims + 1) *ndarray*) – The homogeneous matrix of the affine transformation.
- **copy** (*bool*, optional) – If False avoid copying h_matrix for performance.

- **skip_checks** (*bool*, optional) – If True avoid sanity checks on h_matrix for performance.

apply (*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace (**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector (***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after (*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameterstransform (*Transform*) – Transform to be applied **before** self

Returnstransform (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace (*transform*)

Update self so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameterstransform (*composes_inplace_with*) – Transform to be applied **before** self

RaisesValueError – If transform isn't an instance of *composes_inplace_with*

compose_before(*transform*)

A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace(*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

decompose()

Decompose this transform into discrete Affine Transforms.

Useful for understanding the effect of a complex composite transform.

Returns

transforms (*list* of DiscreteAffine) – Equivalent to this affine transform, such that

```
reduce(lambda x, y: x.chain(y), self.decompose()) == self
```

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters vector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()
Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns has_nan_values (*bool*) – If the vectorized object contains nan values.

classmethod init_identity(*n_dims*)
Creates an identity transform.

Parameters n_dims (*int*) – The number of dimensions.

Returns identity (*Similarity*) – The identity matrix transform.

pseudoinverse()
The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type *Homogeneous*

pseudoinverse_vector(*vector*)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parameters vector ((n_parameters,) *ndarray*) – A vectorized version of self

Returns pseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the h_matrix through this method, specifically if changing the h_matrix could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the h_matrix is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If False, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, skip checking. Useful for performance.

Raises NotImplementedError – If *h_matrix_is_mutable* returns False.

composes_inplace_with
Affine can swallow composition with any other *Affine*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.

Type (n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable
Deprecated `True` iff `set_h_matrix()` is permitted on this type of transform.

If this returns `False` calls to `set_h_matrix()` will raise a `NotImplementedError`.
Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.
Type `True`

linear_component
The linear component of this affine transform.
Type `(n_dims, n_dims)` *ndarray*

n_dims
The dimensionality of the data the transform operates on.
Type *int*

n_dims_output
The output of the data from the transform.
Type *int*

n_parameters
Number of parameters of Similarity

2D Similarity - 4 parameters

```
[(1 + a), -b,      tx]
[b,       (1 + a), ty]
```

3D Similarity: Currently not supported
Returns **n_parameters** (*int*) – The transform parameters
Raises *DimensionalityError, NotImplementedError* – Only 2D transforms are supported.

translation_component
The translation component of this affine transform.
Type `(n_dims,)` *ndarray*

Rotation

class `menpo.transform.`**`Rotation`**(*rotation_matrix*, *skip_checks=False*)
Bases: `DiscreteAffine`, `Similarity`

Abstract *n_dims* rotation transform.
Parameters
- **rotation_matrix** (`(n_dims, n_dims)` *ndarray*) – A valid, square rotation matrix
- **skip_checks** (*bool*, optional) – If `True` avoid sanity checks on `rotation_matrix` for performance.

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returns **transformed** (type(x)) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

axis_and_angle_of_rotation()
Abstract method for computing the axis and angle of rotation.

Returns
- **axis** ((n_dims,) *ndarray*) – The unit vector representing the axis of rotation
- **angle_of_rotation** (*float*) – The angle in radians of the rotation about the axis. The angle is signed in a right handed sense.

compose_after(*transform*)
A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **before** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace(*transform*)
Update self so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **before** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

compose_before(*transform*)
A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace(*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

decompose()

A DiscreteAffine is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (DiscreteAffine) – Deep copy of *self*.

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains nan values.

classmethod init_from_2d_ccw_angle(*theta*, *degrees=True*)
Convenience constructor for 2D CCW rotations about the origin.

Parameters

- **theta** (*float*) – The angle of rotation about the origin
- **degrees** (*bool*, optional) – If `True` theta is interpreted as a degree. If `False`, theta is interpreted as radians.

Returns rotation (`Rotation`) – A 2D rotation transform.

classmethod init_identity(*n_dims*)
Creates an identity transform.

Parameters n_dims (*int*) – The number of dimensions.

Returns identity (`Rotation`) – The identity matrix transform.

pseudoinverse()
The inverse rotation matrix.

Type `Rotation`

pseudoinverse_vector(*vector*)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters vector (`(n_parameters,)` *ndarray*) – A vectorized version of `self`

Returns pseudoinverse_vector (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See `h_matrix_is_mutable` for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If `h_matrix_is_mutable` returns `False`.

set_rotation_matrix(*value*, *skip_checks=False*)
Sets the rotation matrix.

Parameters

- **value** (`(n_dims, n_dims)` *ndarray*) – The new rotation matrix.
- **skip_checks** (*bool*, optional) – If `True` avoid sanity checks on `value` for performance.

composes_inplace_with
`Rotation` can swallow composition with any other `Rotation`.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.

Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable
Deprecated `True` iff *set_h_matrix()* is permitted on this type of transform.

If this returns `False` calls to *set_h_matrix()* will raise a `NotImplementedError`.
Typebool

has_true_inverse
The pseudoinverse is an exact inverse.
Type`True`

linear_component
The linear component of this affine transform.
Type `(n_dims, n_dims)` *ndarray*

n_dims
The dimensionality of the data the transform operates on.
Type*int*

n_dims_output
The output of the data from the transform.
Type*int*

rotation_matrix
The rotation matrix.
Type `(n_dims, n_dims)` *ndarray*

translation_component
The translation component of this affine transform.
Type `(n_dims,)` *ndarray*

Translation

class `menpo.transform.`**Translation**(*translation*, *skip_checks=False*)
Bases: *DiscreteAffine*, `Similarity`

An `n_dims`-dimensional translation transform.

Parameters
- **translation** (`(n_dims,)` *ndarray*) – The translation in each axis.
- **skip_checks** (*bool*, optional) – If `True` avoid sanity checks on `h_matrix` for performance.

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returnstransformed (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)
A `Transform` that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **before** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace(*transform*)
Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before(*transform*)
A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **after** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace(*transform*)
Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** `self`

Raises `ValueError` – If `transform` isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

decompose()

A `DiscreteAffine` is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (`DiscreteAffine`) – Deep copy of *self*.

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

classmethod init_identity(*n_dims*)

Creates an identity transform.

Parameters **n_dims** (*int*) – The number of dimensions.

Returns **identity** (*Translation*) – The identity matrix transform.

pseudoinverse()

The inverse translation (negated).

Type *Translation*

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters vector (`(n_parameters,)` *ndarray*) – A vectorized version of `self`

Returns pseudoinverse_vector (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

composes_inplace_with

Affine can swallow composition with any other *Affine*.

composes_with

Any Homogeneous can compose with any other Homogeneous.

h_matrix

The homogeneous matrix defining this transform.

Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable

Deprecated `True` iff *set_h_matrix()* is permitted on this type of transform.

If this returns `False` calls to *set_h_matrix()* will raise a `NotImplementedError`.

Type *bool*

has_true_inverse

The pseudoinverse is an exact inverse.

Type `True`

linear_component

The linear component of this affine transform.

Type `(n_dims, n_dims)` *ndarray*

n_dims

The dimensionality of the data the transform operates on.

Type *int*

n_dims_output

The output of the data from the transform.

Type *int*

n_parameters

The number of parameters – `n_dims`

Type *int*

translation_component

The translation component of this affine transform.

Type `(n_dims,)` *ndarray*

Scale

menpo.transform.**Scale**(*scale_factor*, *n_dims=None*)
Factory function for producing Scale transforms. Zero scale factors are not permitted.

A *UniformScale* will be produced if:
- A *float* scale_factor and a n_dims *kwarg* are provided
- A *ndarray* scale_factor with shape (n_dims,) is provided with all elements being the same

A *NonUniformScale* will be provided if:
- A *ndarray* scale_factor with shape (n_dims,) is provided with at least two differing scale factors.

Parameters
- **scale_factor** (*float* or (n_dims,) *ndarray*) – Scale for each axis.
- **n_dims** (*int*, optional) – The dimensionality of the output transform.

Returnsscale (*UniformScale* or *NonUniformScale*) – The correct type of scale

RaisesValueError – If any of the scale factors is zero

UniformScale

class menpo.transform.**UniformScale**(*scale*, *n_dims*, *skip_checks=False*)
Bases: *DiscreteAffine*, Similarity

An abstract similarity scale transform, with a single scale component applied to all dimensions. This is abstracted out to remove unnecessary code duplication.

Parameters
- **scale** ((n_dims,) *ndarray*) – A scale for each axis.
- **n_dims** (*int*) – The number of dimensions
- **skip_checks** (*bool*, optional) – If True avoid sanity checks on h_matrix for performance.

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters
- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after (*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **before** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace (*transform*)

Update self so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **before** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

compose_before (*transform*)

A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace (*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returnstype(self) – A copy of this object

decompose()
A DiscreteAffine is already maximally decomposed - return a copy of self in a *list*.

Returnstransform (DiscreteAffine) – Deep copy of *self*.

from_vector(*vector*)
Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returnstransform (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()
Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

classmethod init_identity(*n_dims*)
Creates an identity transform.

Parametersn_dims (*int*) – The number of dimensions.

Returnsidentity (*UniformScale*) – The identity matrix transform.

pseudoinverse()
The inverse scale.

Type*UniformScale*

pseudoinverse_vector(*vector*)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parametersvector ((n_parameters,) *ndarray*) – A vectorized version of self

Returnspseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

composes_inplace_with
UniformScale can swallow composition with any other *UniformScale*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.

Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable
Deprecated `True` iff *set_h_matrix()* is permitted on this type of transform.

If this returns `False` calls to *set_h_matrix()* will raise a `NotImplementedError`.

Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.

Type `True`

linear_component
The linear component of this affine transform.

Type `(n_dims, n_dims)` *ndarray*

n_dims
The dimensionality of the data the transform operates on.

Type *int*

n_dims_output
The output of the data from the transform.

Type *int*

n_parameters
The number of parameters – 1

Type *int*

scale
The single scale value.

Type *float*

translation_component
The translation component of this affine transform.

Type `(n_dims,)` *ndarray*

NonUniformScale

class `menpo.transform.`**NonUniformScale**(*scale*, *skip_checks=False*)
Bases: *DiscreteAffine*, `Affine`

An `n_dims` scale transform, with a scale component for each dimension.

Parameters

- **scale** (`(n_dims,)` *ndarray*) – A scale for each axis.
- **skip_checks** (*bool*, optional) – If `True` avoid sanity checks on `h_matrix` for performance.

apply(*x*, *batch_size=None*, ***kwargs*)

Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters

- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns**transformed** (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns**vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters**transform** (*Transform*) – Transform to be applied **before** `self`

Returns**transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace(*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **before** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

compose_before(*transform*)

A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace(*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

decompose()

A DiscreteAffine is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (DiscreteAffine) – Deep copy of *self*.

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters vector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returns has_nan_values (*bool*) – If the vectorized object contains nan values.

classmethod init_identity(*n_dims*)

Creates an identity transform.

Parameters n_dims (*int*) – The number of dimensions.

Returns identity (*NonUniformScale*) – The identity matrix transform.

pseudoinverse()

The inverse scale matrix.

Type *NonUniformScale*

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parameters vector ((n_parameters,) *ndarray*) – A vectorized version of self

Returns pseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the h_matrix through this method, specifically if changing the h_matrix could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the h_matrix is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If False, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, skip checking. Useful for performance.

Raises NotImplementedError – If *h_matrix_is_mutable* returns False.

composes_inplace_with

NonUniformScale can swallow composition with any other *NonUniformScale* and *UniformScale*.

composes_with

Any Homogeneous can compose with any other Homogeneous.

h_matrix

The homogeneous matrix defining this transform.

Type (n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable

Deprecated True iff *set_h_matrix()* is permitted on this type of transform.

If this returns False calls to *set_h_matrix()* will raise a NotImplementedError.

Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.
TypeTrue

linear_component
The linear component of this affine transform.
Type (n_dims, n_dims) *ndarray*

n_dims
The dimensionality of the data the transform operates on.
Type*int*

n_dims_output
The output of the data from the transform.
Type*int*

n_parameters
The number of parameters – n_dims. They have the form [scale_x, scale_y, ...] representing the scale across each axis.
Type*list* of *int*

scale
The scale vector.
Type (n_dims,) *ndarray*

translation_component
The translation component of this affine transform.
Type (n_dims,) *ndarray*

2.9.3 Alignments

ThinPlateSplines

class menpo.transform.**ThinPlateSplines**(*source*, *target*, *kernel=None*, *min_singular_val=0.0001*)
Bases: *Alignment*, Transform, *Invertible*

The thin plate splines (TPS) alignment between 2D *source* and *target* landmarks.

kernel can be used to specify an alternative kernel function. If None is supplied, the *R2LogR2RBF* kernel will be used.

Parameters
- **source** ((N, 2) *ndarray*) – The source points to apply the tps from
- **target** ((N, 2) *ndarray*) – The target points to apply the tps to
- **kernel** (RadialBasisFunction, optional) – The kernel to apply.
- **min_singular_val** (*float*, optional) – If the target has points that are nearly coincident, the coefficients matrix is rank deficient, and therefore not invertible. Therefore, we only take the inverse on the full-rank matrix and drop any singular values that are less than this value (close to zero).

RaisesValueError – TPS is only with on 2-dimensional data

aligned_source()
The result of applying self to *source*
Type*PointCloud*

alignment_error()
The Frobenius Norm of the difference between the target and the aligned source.
Type*float*

apply (*x*, *batch_size=None*, ***kwargs*)
Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returns**transformed** (type(x)) – The transformed object or array

apply_inplace (**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after (*transform*)
Returns a *TransformChain* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, *o*.

Parameters**transform** (*Transform*) – Transform to be applied **before** self

Returns**transform** (*TransformChain*) – The resulting transform chain.

compose_before (*transform*)
Returns a *TransformChain* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

Parameters**transform** (*Transform*) – Transform to be applied **after** self

Returns**transform** (*TransformChain*) – The resulting transform chain.

copy ()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returnstype(self) – A copy of this object

pseudoinverse ()
The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type`type(self)`

set_target(*new_target*)
Update this object so that it attempts to recreate the `new_target`.
Parameters **new_target** (*PointCloud*) – The new target that this object should try and regenerate.

has_true_inverse
type – `False`

n_dims
The number of dimensions of the *target*.
Type*int*

n_dims_output
The output of the data from the transform.

`None` if the output of the transform is not dimension specific.
Type*int* or `None`

n_points
The number of points on the *target*.
Type*int*

source
The source *PointCloud* that is used in the alignment.

The source is not mutable.
Type*PointCloud*

target
The current *PointCloud* that this object produces.

To change the target, use *set_target()*.
Type*PointCloud*

PiecewiseAffine

`menpo.transform.`**PiecewiseAffine**
alias of `CachedPWA`

AlignmentAffine

class `menpo.transform.`**AlignmentAffine**(*source*, *target*)
Bases: `HomogFamilyAlignment`, `Affine`

Constructs an *Affine* by finding the optimal affine transform to align *source* to *target*.
Parameters
- **source** (*PointCloud*) – The source pointcloud instance used in the alignment
- **target** (*PointCloud*) – The target pointcloud instance used in the alignment

Notes

We want to find the optimal transform M which satisfies $Ma = b$ where a and b are the *source* and *target* homogeneous vectors respectively.

```
(M a)' = b'
a' M' = b'
a a' M' = a b'
```

a a' is of shape *(n_dim + 1, n_dim + 1)* and so can be inverted to solve for *M*.

This approach is the analytical linear least squares solution to the problem at hand. It will have a solution as long as *(a a')* is non-singular, which generally means at least 2 corresponding points are required.

---

aligned_source()
The result of applying `self` to *source*
TypePointCloud

alignment_error()
The Frobenius Norm of the difference between the target and the aligned source.
Type*float*

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns**transformed** (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_non_alignment()
Returns a copy of this *Affine* without its alignment nature.
Returns**transform** (*Affine*) – A version of this affine with the same transform behavior but without the alignment logic.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.
Returns**vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)
A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.
Parameters**transform** (*Transform*) – Transform to be applied **before** `self`

Returns transform (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace (*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters transform (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before (*transform*)

A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters transform (`Transform`) – Transform to be applied **after** `self`

Returns transform (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace (*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters transform (`composes_inplace_with`) – Transform to be applied **after** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

copy ()

Generate an efficient copy of this `HomogFamilyAlignment`.

Returns new_transform (`type(self)`) – A copy of this object

decompose ()

Decompose this transform into discrete Affine Transforms.

Useful for understanding the effect of a complex composite transform.

Returns

transforms (*list* of `DiscreteAffine`) – Equivalent to this affine transform, such that

```
reduce(lambda x, y: x.chain(y), self.decompose()) == self
```

from_vector(*vector*)
Build a new instance of the object from its vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a `deepcopy` of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)
Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** (`(n_parameters,)` *ndarray*) – Flattened representation of this object

has_nan_values()
Tests if the vectorized form of the object contains `nan` values or not. This is particularly useful for objects with unknown values that have been mapped to `nan` values.

Returns **has_nan_values** (*bool*) – If the vectorized object contains `nan` values.

init_identity(*n_dims*)
Creates an identity matrix Affine transform.

Parameters **n_dims** (*int*) – The number of dimensions.

Returns **identity** (*Affine*) – The identity matrix transform.

pseudoinverse()
The pseudoinverse of the transform - that is, the transform that results from swapping source and target, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Returns **transform** (`type(self)`) – The inverse of this transform.

pseudoinverse_vector(*vector*)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters **vector** (`(n_parameters,)` *ndarray*) – A vectorized version of `self`

Returns **pseudoinverse_vector** (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

set_target (*new_target*)
Update this object so that it attempts to recreate the `new_target`.

Parameters **new_target** (`PointCloud`) – The new target that this object should try and regenerate.

composes_inplace_with
`Affine` can swallow composition with any other `Affine`.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.

Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable
Deprecated `True` iff `set_h_matrix()` is permitted on this type of transform.

If this returns `False` calls to `set_h_matrix()` will raise a `NotImplementedError`.

Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.

Type `True`

linear_component
The linear component of this affine transform.

Type `(n_dims, n_dims)` *ndarray*

n_dims
The number of dimensions of the `target`.

Type *int*

n_dims_output
The output of the data from the transform.

Type *int*

n_parameters
`n_dims * (n_dims + 1)` parameters - every element of the matrix but the homogeneous part.

Type int

Examples

2D Affine: 6 parameters:

```
[p1, p3, p5]
[p2, p4, p6]
```

3D Affine: 12 parameters:

```
[p1, p4, p7, p10]
[p2, p5, p8, p11]
[p3, p6, p9, p12]
```

n_points
The number of points on the `target`.

Type *int*

source
The source `PointCloud` that is used in the alignment.

The source is not mutable.
Type `PointCloud`

target
The current `PointCloud` that this object produces.

To change the target, use `set_target()`.
Type `PointCloud`

translation_component
The translation component of this affine transform.
Type `(n_dims,)` *ndarray*

AlignmentSimilarity

class `menpo.transform.`**AlignmentSimilarity**(*source*, *target*, *rotation=True*, *allow_mirror=False*)
Bases: `HomogFamilyAlignment`, `Similarity`

Infers the similarity transform relating two vectors with the same dimensionality. This is simply the procrustes alignment of the *source* to the *target*.

Parameters
- **source** (`PointCloud`) – The source pointcloud instance used in the alignment
- **target** (`PointCloud`) – The target pointcloud instance used in the alignment
- **rotation** (*bool*, optional) – If `False`, the rotation component of the similarity transform is not inferred.
- **allow_mirror** (*bool*, optional) – If `True`, the Kabsch algorithm check is not performed, and mirroring of the Rotation matrix is permitted.

aligned_source()
The result of applying `self` to `source`
Type `PointCloud`

alignment_error()
The Frobenius Norm of the difference between the target and the aligned source.
Type *float*

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns **transformed** (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_non_alignment()
Returns a copy of this similarity without it's alignment nature.
Returns **transform** (*Similarity*) – A version of this similarity with the same transform behavior but without the alignment logic.

as_vector(***kwargs*)
Returns a flattened representation of the object as a single vector.
Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)
A `Transform` that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.
Parameters **transform** (`Transform`) – Transform to be applied **before** `self`
Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace(*transform*)
Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.
Parameters **transform** (`composes_inplace_with`) – Transform to be applied **before** `self`
Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before(*transform*)
A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.
Parameters **transform** (`Transform`) – Transform to be applied **after** `self`
Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace(*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameterstransform (*composes_inplace_with*) – Transform to be applied **after** self

RaisesValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this HomogFamilyAlignment.

Returnsnew_transform (type(self)) – A copy of this object

decompose()

Decompose this transform into discrete Affine Transforms.

Useful for understanding the effect of a complex composite transform.

Returns

transforms (*list* of DiscreteAffine) – Equivalent to this affine transform, such that

```
reduce(lambda x, y: x.chain(y), self.decompose()) == self
```

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returnstransform (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

init_identity(*n_dims*)

Creates an identity transform.

Parametersn_dims (*int*) – The number of dimensions.

Returnsidentity (*Similarity*) – The identity matrix transform.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping source and target, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Returnstransform (type(self)) – The inverse of this transform.

pseudoinverse_vector(*vector*)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parameters **vector** (`(n_parameters,)` *ndarray*) – A vectorized version of `self`

Returns **pseudoinverse_vector** (`(n_parameters,)` *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)
Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates `h_matrix`, optionally performing sanity checks.

Note that it won't always be possible to manually specify the `h_matrix` through this method, specifically if changing the `h_matrix` could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the `h_matrix` is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If `False`, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If `True`, skip checking. Useful for performance.

Raises `NotImplementedError` – If *h_matrix_is_mutable* returns `False`.

set_target(*new_target*)
Update this object so that it attempts to recreate the `new_target`.

Parameters **new_target** (*PointCloud*) – The new target that this object should try and regenerate.

composes_inplace_with
Affine can swallow composition with any other *Affine*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.

Type `(n_dims + 1, n_dims + 1)` *ndarray*

h_matrix_is_mutable
Deprecated `True` iff *set_h_matrix()* is permitted on this type of transform.

If this returns `False` calls to *set_h_matrix()* will raise a `NotImplementedError`.

Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.

Type `True`

linear_component
The linear component of this affine transform.

Type `(n_dims, n_dims)` *ndarray*

n_dims
The number of dimensions of the *target*.

Type *int*

n_dims_output
The output of the data from the transform.

Type*int*

n_parameters

Number of parameters of Similarity

2D Similarity - 4 parameters

```
[(1 + a), -b,      tx]
[b,       (1 + a), ty]
```

3D Similarity: Currently not supported

Returns n_parameters (*int*) – The transform parameters

Raises *DimensionalityError, NotImplementedError* – Only 2D transforms are supported.

n_points

The number of points on the `target`.

Type*int*

source

The source `PointCloud` that is used in the alignment.

The source is not mutable.

Type`PointCloud`

target

The current `PointCloud` that this object produces.

To change the target, use `set_target()`.

Type`PointCloud`

translation_component

The translation component of this affine transform.

Type `(n_dims,)` *ndarray*

AlignmentRotation

class `menpo.transform.`**`AlignmentRotation`**`(`*source*, *target*, *allow_mirror=False*`)`

Bases: `HomogFamilyAlignment`, `Rotation`

Constructs an `Rotation` by finding the optimal rotation transform to align *source* to *target*.

Parameters

- **source** (`PointCloud`) – The source pointcloud instance used in the alignment
- **target** (`PointCloud`) – The target pointcloud instance used in the alignment
- **allow_mirror** (*bool*, optional) – If `True`, the Kabsch algorithm check is not performed, and mirroring of the Rotation matrix is permitted.

aligned_source`()`

The result of applying `self` to `source`

Type`PointCloud`

alignment_error`()`

The Frobenius Norm of the difference between the target and the aligned source.

Type*float*

apply`(`*x*, *batch_size=None*, ***kwargs*`)`

Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters
- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returns **transformed** (type(x)) – The transformed object or array

apply_inplace(**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_non_alignment()

Returns a copy of this rotation without its alignment nature.

Returns **transform** (Rotation) – A version of this rotation with the same transform behavior but without the alignment logic.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

axis_and_angle_of_rotation()

Abstract method for computing the axis and angle of rotation.

Returns
- **axis** ((n_dims,) *ndarray*) – The unit vector representing the axis of rotation
- **angle_of_rotation** (*float*) – The angle in radians of the rotation about the axis. The angle is signed in a right handed sense.

compose_after(*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **before** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace(*transform*)

Update self so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **before** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

compose_before(*transform*)

A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace(*transform*)

Update self so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this HomogFamilyAlignment.

Returns **new_transform** (type(self)) – A copy of this object

decompose()

A DiscreteAffine is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (DiscreteAffine) – Deep copy of *self*.

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returns **transform** (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parameters **vector** ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

init_from_2d_ccw_angle(*theta*, *degrees=True*)

Convenience constructor for 2D CCW rotations about the origin.

Parameters

- **theta** (*float*) – The angle of rotation about the origin
- **degrees** (*bool*, optional) – If True theta is interpreted as a degree. If False, theta is interpreted as radians.

Returnsrotation (Rotation) – A 2D rotation transform.

init_identity(*n_dims*)

Creates an identity transform.

Parametersn_dims (*int*) – The number of dimensions.

Returnsidentity (Rotation) – The identity matrix transform.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping source and target, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Returnstransform (type(self)) – The inverse of this transform.

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parametersvector ((n_parameters,) *ndarray*) – A vectorized version of self

Returnspseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the h_matrix through this method, specifically if changing the h_matrix could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the h_matrix is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If False, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, skip checking. Useful for performance.

RaisesNotImplementedError – If *h_matrix_is_mutable* returns False.

set_rotation_matrix(*value*, *skip_checks=False*)

Sets the rotation matrix.

Parameters

- **value** ((n_dims, n_dims) *ndarray*) – The new rotation matrix.
- **skip_checks** (*bool*, optional) – If True avoid sanity checks on value for performance.

set_target(*new_target*)

Update this object so that it attempts to recreate the new_target.

Parameters **new_target** (*PointCloud*) – The new target that this object should try and regenerate.

composes_inplace_with
Rotation can swallow composition with any other *Rotation*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.
Type (n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable
Deprecated True iff *set_h_matrix()* is permitted on this type of transform.

If this returns False calls to *set_h_matrix()* will raise a NotImplementedError.
Type *bool*

has_true_inverse
The pseudoinverse is an exact inverse.
Type True

linear_component
The linear component of this affine transform.
Type (n_dims, n_dims) *ndarray*

n_dims
The number of dimensions of the *target*.
Type *int*

n_dims_output
The output of the data from the transform.
Type *int*

n_points
The number of points on the *target*.
Type *int*

rotation_matrix
The rotation matrix.
Type (n_dims, n_dims) *ndarray*

source
The source *PointCloud* that is used in the alignment.

The source is not mutable.
Type *PointCloud*

target
The current *PointCloud* that this object produces.

To change the target, use *set_target()*.
Type *PointCloud*

translation_component
The translation component of this affine transform.
Type (n_dims,) *ndarray*

AlignmentTranslation

class menpo.transform.**AlignmentTranslation**(*source*, *target*)

Bases: HomogFamilyAlignment, Translation

Constructs a *Translation* by finding the optimal translation transform to align *source* to *target*.

Parameters

- **source** (*PointCloud*) – The source pointcloud instance used in the alignment
- **target** (*PointCloud*) – The target pointcloud instance used in the alignment

aligned_source()

The result of applying self to *source*

Type *PointCloud*

alignment_error()

The Frobenius Norm of the difference between the target and the aligned source.

Type *float*

apply(*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returns **transformed** (type(x)) – The transformed object or array

apply_inplace(**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_non_alignment()

Returns a copy of this translation without its alignment nature.

Returns **transform** (*Translation*) – A version of this transform with the same transform behavior but without the alignment logic.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returns **vector** (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **before** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace (*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before (*transform*)

A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **after** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace (*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **after** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

copy ()

Generate an efficient copy of this `HomogFamilyAlignment`.

Returns **new_transform** (`type(self)`) – A copy of this object

decompose ()

A `DiscreteAffine` is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (`DiscreteAffine`) – Deep copy of *self*.

from_vector (*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returnstransform (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

init_identity(*n_dims*)

Creates an identity transform.

Parametersn_dims (*int*) – The number of dimensions.

Returnsidentity (*Translation*) – The identity matrix transform.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping source and target, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Returnstransform (type(self)) – The inverse of this transform.

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parametersvector ((n_parameters,) *ndarray*) – A vectorized version of self

Returnspseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the h_matrix through this method, specifically if changing the h_matrix could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the h_matrix is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If False, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, skip checking. Useful for performance.

RaisesNotImplementedError – If *h_matrix_is_mutable* returns False.

set_target(*new_target*)

Update this object so that it attempts to recreate the new_target.

Parameters**new_target** (*PointCloud*) – The new target that this object should try and regenerate.

composes_inplace_with
Affine can swallow composition with any other *Affine*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.
Type(n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable
Deprecated True iff *set_h_matrix()* is permitted on this type of transform.

If this returns False calls to *set_h_matrix()* will raise a NotImplementedError.
Type*bool*

has_true_inverse
The pseudoinverse is an exact inverse.
TypeTrue

linear_component
The linear component of this affine transform.
Type(n_dims, n_dims) *ndarray*

n_dims
The number of dimensions of the *target*.
Type*int*

n_dims_output
The output of the data from the transform.
Type*int*

n_parameters
The number of parameters – n_dims
Type*int*

n_points
The number of points on the *target*.
Type*int*

source
The source *PointCloud* that is used in the alignment.

The source is not mutable.
Type*PointCloud*

target
The current *PointCloud* that this object produces.

To change the target, use *set_target()*.
Type*PointCloud*

translation_component
The translation component of this affine transform.
Type(n_dims,) *ndarray*

AlignmentUniformScale

class menpo.transform.**AlignmentUniformScale**(*source*, *target*)

Bases: HomogFamilyAlignment, UniformScale

Constructs a *UniformScale* by finding the optimal scale transform to align *source* to *target*.

Parameters

- **source** (*PointCloud*) – The source pointcloud instance used in the alignment
- **target** (*PointCloud*) – The target pointcloud instance used in the alignment

aligned_source()

The result of applying self to *source*

Type*PointCloud*

alignment_error()

The Frobenius Norm of the difference between the target and the aligned source.

Type*float*

apply(*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace(**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

as_non_alignment()

Returns a copy of this uniform scale without it's alignment nature.

Returnstransform (UniformScale) – A version of this scale with the same transform behavior but without the alignment logic.

as_vector(***kwargs*)

Returns a flattened representation of the object as a single vector.

Returnsvector (*(N,) ndarray*) – The core representation of the object, flattened into a single vector. Note that this is always a view back on to the original object, but is not writable.

compose_after(*transform*)

A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **before** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace(*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before(*transform*)

A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameters **transform** (`Transform`) – Transform to be applied **after** `self`

Returns **transform** (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace(*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (`composes_inplace_with`) – Transform to be applied **after** `self`

Raises `ValueError` – If `transform` isn't an instance of `composes_inplace_with`

copy()

Generate an efficient copy of this `HomogFamilyAlignment`.

Returns **new_transform** (`type(self)`) – A copy of this object

decompose()

A `DiscreteAffine` is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (`DiscreteAffine`) – Deep copy of *self*.

from_vector(*vector*)

Build a new instance of the object from its vectorized state.

self is used to fill out the missing state required to rebuild a full object from it's standardized flattened state. This is the default implementation, which is a deepcopy of the object followed by a call to *from_vector_inplace()*. This method can be overridden for a performance benefit if desired.

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of the object.

Returnstransform (*Homogeneous*) – An new instance of this class.

from_vector_inplace(*vector*)

Deprecated. Use the non-mutating API, *from_vector*.

For internal usage in performance-sensitive spots, see *_from_vector_inplace()*

Parametersvector ((n_parameters,) *ndarray*) – Flattened representation of this object

has_nan_values()

Tests if the vectorized form of the object contains nan values or not. This is particularly useful for objects with unknown values that have been mapped to nan values.

Returnshas_nan_values (*bool*) – If the vectorized object contains nan values.

init_identity(*n_dims*)

Creates an identity transform.

Parametersn_dims (*int*) – The number of dimensions.

Returnsidentity (*UniformScale*) – The identity matrix transform.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping source and target, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Returnstransform (type(self)) – The inverse of this transform.

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

Parametersvector ((n_parameters,) *ndarray*) – A vectorized version of self

Returnspseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

set_h_matrix(*value*, *copy=True*, *skip_checks=False*)

Deprecated Deprecated - do not use this method - you are better off just creating a new transform!

Updates h_matrix, optionally performing sanity checks.

Note that it won't always be possible to manually specify the h_matrix through this method, specifically if changing the h_matrix could change the nature of the transform. See *h_matrix_is_mutable* for how you can discover if the h_matrix is allowed to be set for a given class.

Parameters

- **value** (*ndarray*) – The new homogeneous matrix to set.
- **copy** (*bool*, optional) – If False, do not copy the h_matrix. Useful for performance.
- **skip_checks** (*bool*, optional) – If True, skip checking. Useful for performance.

RaisesNotImplementedError – If *h_matrix_is_mutable* returns False.

set_target(*new_target*)

Update this object so that it attempts to recreate the new_target.

Parameters**new_target** (*PointCloud*) – The new target that this object should try and regenerate.

composes_inplace_with
UniformScale can swallow composition with any other *UniformScale*.

composes_with
Any Homogeneous can compose with any other Homogeneous.

h_matrix
The homogeneous matrix defining this transform.
Type(n_dims + 1, n_dims + 1) *ndarray*

h_matrix_is_mutable
Deprecated True iff *set_h_matrix()* is permitted on this type of transform.

If this returns False calls to *set_h_matrix()* will raise a NotImplementedError.
Type*bool*

has_true_inverse
The pseudoinverse is an exact inverse.
TypeTrue

linear_component
The linear component of this affine transform.
Type(n_dims, n_dims) *ndarray*

n_dims
The number of dimensions of the *target*.
Type*int*

n_dims_output
The output of the data from the transform.
Type*int*

n_parameters
The number of parameters – 1
Type*int*

n_points
The number of points on the *target*.
Type*int*

scale
The single scale value.
Type*float*

source
The source *PointCloud* that is used in the alignment.

The source is not mutable.
Type*PointCloud*

target
The current *PointCloud* that this object produces.

To change the target, use *set_target()*.
Type*PointCloud*

translation_component
The translation component of this affine transform.
Type(n_dims,) *ndarray*

2.9.4 Group Alignments

GeneralizedProcrustesAnalysis

class menpo.transform.**GeneralizedProcrustesAnalysis**(*sources*, *target=None*, *allow_mirror=False*)

Bases: *MultipleAlignment*

Class for aligning multiple source shapes between them.

After construction, the *AlignmentSimilarity* transforms used to map each *source* optimally to the *target* can be found at *transforms*.

Parameters

- **sources** (*list* of *PointCloud*) – List of pointclouds to be aligned.
- **target** (*PointCloud*, optional) – The target *PointCloud* to align each source to. If None, then the mean of the sources is used.
- **allow_mirror** (*bool*, optional) – If True, the Kabsch algorithm check is not performed, and mirroring of the Rotation matrix is permitted.

Raises ValueError – Need at least two sources to align

mean_aligned_shape()

Returns the mean of the aligned shapes.

Type *PointCloud*

mean_alignment_error()

Returns the average error of the recursive procrustes alignment.

Type *float*

2.9.5 Composite Transforms

TransformChain

class menpo.transform.**TransformChain**(*transforms*)

Bases: *ComposableTransform*

A chain of transforms that can be efficiently applied one after the other.

This class is the natural product of composition. Note that objects may know how to compose themselves more efficiently - such objects implement the ComposableTransform or VComposable interfaces.

Parameters **transforms** (*list* of *Transform*) – The *list* of transforms to be applied. Note that the first transform will be applied first - the result of which is fed into the second transform and so on until the chain is exhausted.

apply(*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.

- **kwargs** (*dict*) – Passed through to `_apply()`.

Returnstransformed (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after(*transform*)

A `Transform` that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, `o`.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameterstransform (`Transform`) – Transform to be applied **before** `self`

Returnstransform (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_after_inplace(*transform*)

Update `self` so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameterstransform (`composes_inplace_with`) – Transform to be applied **before** `self`

Raises`ValueError` – If `transform` isn't an instance of `composes_inplace_with`

compose_before(*transform*)

A `Transform` that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a `TransformChain` as a last resort. See `composes_with` for a description of how the mode of composition is decided.

Parameterstransform (`Transform`) – Transform to be applied **after** `self`

Returnstransform (`Transform` or `TransformChain`) – If the composition was native, a single new `Transform` will be returned. If not, a `TransformChain` is returned instead.

compose_before_inplace(*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

a is permanently altered to be the result of the composition. b is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** self

Raises ValueError – If transform isn't an instance of *composes_inplace_with*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

composes_inplace_with

The *Transform* s that this transform composes inplace with **natively** (i.e. no *TransformChain* will be produced).

An attempt to compose inplace against any type that is not an instance of this property on this class will result in an *Exception*.

Type *Transform* or *tuple* of *Transform* s

composes_with

The *Transform* s that this transform composes with **natively** (i.e. no *TransformChain* will be produced).

If native composition is not possible, falls back to producing a *TransformChain*.

By default, this is the same list as *composes_inplace_with*.

Type *Transform* or *tuple* of *Transform* s

n_dims

The dimensionality of the data the transform operates on.

None if the transform is not dimension specific.

Type *int* or None

n_dims_output

The output of the data from the transform.

None if the output of the transform is not dimension specific.

Type *int* or None

2.9.6 Radial Basis Functions

R2LogR2RBF

class menpo.transform.**R2LogR2RBF**(*c*)

Bases: RadialBasisFunction

The $r^2 \log r^2$ basis function.

The derivative of this function is $2r(\log r^2 + 1)$.

Note: $r = \lVert x - c \rVert$

Parameters **c** ((n_centres, n_dims) *ndarray*) – The set of centers that make the basis. Usually represents a set of source landmarks.

apply (*x*, *batch_size=None*, ***kwargs*)
Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace (**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after (*transform*)
Returns a *TransformChain* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, *o*.

Parameterstransform (*Transform*) – Transform to be applied **before** self
Returnstransform (*TransformChain*) – The resulting transform chain.

compose_before (*transform*)
Returns a *TransformChain* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

Parameterstransform (*Transform*) – Transform to be applied **after** self
Returnstransform (*TransformChain*) – The resulting transform chain.

copy ()
Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returnstype(self) – A copy of this object

n_centres
The number of centres.

Type*int*

n_dims
The RBF can only be applied on points with the same dimensionality as the centres.
Type *int*

n_dims_output
The result of the transform has a dimension (weight) for every centre.
Type *int*

R2LogRRBF

class `menpo.transform.R2LogRRBF`(*c*)
Bases: `RadialBasisFunction`

Calculates the $r^2 \log r$ basis function.

The derivative of this function is $r(1 + 2 \log r)$.

Note: $r = \|x - c\|$

Parameters c (`(n_centres, n_dims)` *ndarray*) – The set of centers that make the basis. Usually represents a set of source landmarks.

apply(*x*, *batch_size=None*, ***kwargs*)
Applies this transform to `x`.

If `x` is `Transformable`, `x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, `x` is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any `kwargs` will be passed to the specific transform `_apply()` method.

Parameters
- **x** (`Transformable` or `(n_points, n_dims)` *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not `None`, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to `_apply()`.

Returns transformed (`type(x)`) – The transformed object or array

apply_inplace(**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after(*transform*)
Returns a `TransformChain` that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

`a` and `b` are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, *o*.

Parameters transform (`Transform`) – Transform to be applied **before** self

Returns transform (`TransformChain`) – The resulting transform chain.

compose_before (*transform*)

Returns a *TransformChain* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

Parameters transform (*Transform*) – Transform to be applied **after** self

Returns transform (*TransformChain*) – The resulting transform chain.

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

n_centres

The number of centres.

Type *int*

n_dims

The RBF can only be applied on points with the same dimensionality as the centres.

Type *int*

n_dims_output

The result of the transform has a dimension (weight) for every centre.

Type *int*

2.9.7 Abstract Bases

Transform

class menpo.transform.**Transform**

Bases: *Copyable*

Abstract representation of any spatial transform.

Provides a unified interface to apply the transform with *apply_inplace()* and *apply()*.

All Transforms support basic composition to form a *TransformChain*.

There are two useful forms of composition. Firstly, the mathematical composition symbol *o* has the following definition:

```
Let a(x) and b(x) be two transforms on x.
(a o b)(x) == a(b(x))
```

This functionality is provided by the *compose_after()* family of methods:

```
(a.compose_after(b)).apply(x) == a.apply(b.apply(x))
```

Equally useful is an inversion the order of composition - so that over time a large chain of transforms can be built to do a useful job, and composing on this chain adds another transform to the end (after all other preceding transforms have been performed).

For instance, let's say we want to rescale a *PointCloud* p around its mean, and then translate it some place else. It would be nice to be able to do something like:

```
t = Translation(-p.centre)  # translate to centre
s = Scale(2.0)  # rescale
move = Translate([10, 0 ,0])  # budge along the x axis
t.compose(s).compose(-t).compose(move)
```

In Menpo, this functionality is provided by the *compose_before()* family of methods:

```
(a.compose_before(b)).apply(x) == b.apply(a.apply(x))
```

For native composition, see the ComposableTransform subclass and the VComposable mix-in.

For inversion, see the Invertible and VInvertible mix-ins.

For alignment, see the Alignment mix-in.

apply (*x*, *batch_size=None*, ***kwargs*)

Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace (**args*, ***kwargs*)

Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after (*transform*)

Returns a *TransformChain* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, *o*.

Parameterstransform (*Transform*) – Transform to be applied **before** self

Returnstransform (*TransformChain*) – The resulting transform chain.

compose_before (*transform*)

Returns a *TransformChain* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

a and b are left unchanged.

Parameters **transform** (*Transform*) – Transform to be applied **after** self

Returns **transform** (*TransformChain*) – The resulting transform chain.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

n_dims

The dimensionality of the data the transform operates on.

None if the transform is not dimension specific.

Type *int* or None

n_dims_output

The output of the data from the transform.

None if the output of the transform is not dimension specific.

Type *int* or None

Transformable

class menpo.transform.base.**Transformable**

Bases: *Copyable*

Interface for objects that know how to be transformed by the *Transform* interface.

When Transform.apply_inplace is called on an object, the *_transform_inplace()* method is called, passing in the transforms' _apply() function.

This allows for the object to define how it should transform itself.

_transform_inplace(*transform*)

Apply the given transform function to self inplace.

Parameters **transform** (*function*) – Function that applies a transformation to the transformable object.

Returns **transformed** (type(self)) – The transformed object, having been transformed in place.

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on self will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns type(self) – A copy of this object

ComposableTransform

class menpo.transform.base.composable.**ComposableTransform**

Bases: Transform

Transform subclass that enables native composition, such that the behavior of multiple *Transform* s is composed together in a natural way.

_compose_after_inplace (*transform*)
Specialised inplace composition. This should be overridden to provide specific cases of composition as defined in *composes_inplace_with*.

Parameterstransform (*composes_inplace_with*) – Transform to be applied **before** self

_compose_before_inplace (*transform*)
Specialised inplace composition. This should be overridden to provide specific cases of composition as defined in *composes_inplace_with*.

Parameterstransform (*composes_inplace_with*) – Transform to be applied **after** self

apply (*x*, *batch_size=None*, ***kwargs*)
Applies this transform to x.

If x is Transformable, x will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, x is assumed to be an *ndarray*. The transformation will be non-destructive, returning the transformed version.

Any kwargs will be passed to the specific transform _apply() method.

Parameters

- **x** (Transformable or (n_points, n_dims) *ndarray*) – The array or object to be transformed.
- **batch_size** (*int*, optional) – If not None, this determines how many items from the numpy array will be passed through the transform at a time. This is useful for operations that require large intermediate matrices to be computed.
- **kwargs** (*dict*) – Passed through to _apply().

Returnstransformed (type(x)) – The transformed object or array

apply_inplace (**args*, ***kwargs*)
Deprecated as public supported API, use the non-mutating *apply()* instead.

For internal performance-specific uses, see *_apply_inplace()*.

compose_after (*transform*)
A *Transform* that represents **this** transform composed **after** the given transform:

```
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

a and b are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, o.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameterstransform (*Transform*) – Transform to be applied **before** self

Returnstransform (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_after_inplace (*transform*)
Update self so that it represents **this** transform composed **after** the given transform:

```
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **before** `self`

Raises `ValueError` – If `transform` isn't an instance of *composes_inplace_with*

compose_before (*transform*)

A *Transform* that represents **this** transform composed **before** the given transform:

```
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

`a` and `b` are left unchanged.

An attempt is made to perform native composition, but will fall back to a *TransformChain* as a last resort. See *composes_with* for a description of how the mode of composition is decided.

Parameters **transform** (*Transform*) – Transform to be applied **after** `self`

Returns **transform** (*Transform* or *TransformChain*) – If the composition was native, a single new *Transform* will be returned. If not, a *TransformChain* is returned instead.

compose_before_inplace (*transform*)

Update `self` so that it represents **this** transform composed **before** the given transform:

```
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

`a` is permanently altered to be the result of the composition. `b` is left unchanged.

Parameters **transform** (*composes_inplace_with*) – Transform to be applied **after** `self`

Raises `ValueError` – If `transform` isn't an instance of *composes_inplace_with*

copy ()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

composes_inplace_with

The *Transform* s that this transform composes inplace with **natively** (i.e. no *TransformChain* will be produced).

An attempt to compose inplace against any type that is not an instance of this property on this class will result in an *Exception*.

Type *Transform* or *tuple* of *Transform* s

composes_with

The *Transform* s that this transform composes with **natively** (i.e. no *TransformChain* will be produced).

If native composition is not possible, falls back to producing a *TransformChain*.

By default, this is the same list as `composes_inplace_with`.

Type `Transform` or *tuple* of `Transform` s

n_dims

The dimensionality of the data the transform operates on.

`None` if the transform is not dimension specific.

Type *int* or `None`

n_dims_output

The output of the data from the transform.

`None` if the output of the transform is not dimension specific.

Type *int* or `None`

Invertible

class `menpo.transform.base.invertible.`**Invertible**

Bases: `object`

Mix-in for invertible transforms. Provides an interface for taking the *pseudo* or true inverse of a transform.

Has to be implemented in conjunction with `Transform`.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type `type(self)`

has_true_inverse

`True` if the pseudoinverse is an exact inverse.

Type *bool*

Alignment

class `menpo.transform.base.alignment.`**Alignment**(*source*, *target*)

Bases: `Targetable`, `Viewable`

Mix-in for `Transform` that have been constructed from an optimisation aligning a source `PointCloud` to a target `PointCloud`.

This is naturally an extension of the `Targetable` interface - we just augment `Targetable` with the concept of a source, and related methods to construct alignments between a source and a target.

Note that to inherit from `Alignment`, you have to be a `Transform` subclass first.

Parameters

- **source** (`PointCloud`) – A PointCloud that the alignment will be based from
- **target** (`PointCloud`) – A PointCloud that the alignment is targeted towards

aligned_source()

The result of applying `self` to `source`

Type `PointCloud`

alignment_error()

The Frobenius Norm of the difference between the target and the aligned source.

Type *float*

copy()

Generate an efficient copy of this object.

Note that Numpy arrays and other *Copyable* objects on `self` will be deeply copied. Dictionaries and sets will be shallow copied, and everything else will be assigned (no copy will be made).

Classes that store state other than numpy arrays and immutable types should overwrite this method to ensure all state is copied.

Returns `type(self)` – A copy of this object

set_target(*new_target*)

Update this object so that it attempts to recreate the `new_target`.

Parameters **new_target** (*PointCloud*) – The new target that this object should try and regenerate.

n_dims

The number of dimensions of the *target*.

Type *int*

n_points

The number of points on the *target*.

Type *int*

source

The source *PointCloud* that is used in the alignment.

The source is not mutable.

Type *PointCloud*

target

The current *PointCloud* that this object produces.

To change the target, use *set_target()*.

Type *PointCloud*

MultipleAlignment

class `menpo.transform.groupalign.base.`**MultipleAlignment**(*sources*, *target=None*)

Bases: `object`

Abstract base class for aligning multiple *source* shapes to a *target* shape.

Parameters

- **sources** (*list* of *PointCloud*) – List of pointclouds to be aligned.
- **target** (*PointCloud*, optional) – The target *PointCloud* to align each source to. If `None`, then the mean of the sources is used.

Raises `ValueError` – Need at least two sources to align

DiscreteAffine

class `menpo.transform.homogeneous.affine.`**DiscreteAffine**

Bases: `object`

A discrete Affine transform operation (such as a `Scale()`, `Translation` or `Rotation()`). Has to be invertable. Make sure you inherit from *DiscreteAffine* first, for optimal *decompose()* behavior.

decompose()

A *DiscreteAffine* is already maximally decomposed - return a copy of self in a *list*.

Returns **transform** (*DiscreteAffine*) – Deep copy of *self*.

2.9.8 Performance Specializations

Mix-ins that provide fast vectorized variants of methods.

VComposable

class menpo.transform.base.composable.**VComposable**

Bases: object

Mix-in for *Vectorizable* ComposableTransform s.

Use this mix-in with ComposableTransform if the ComposableTransform in question is *Vectorizable* as this adds from_vector() variants to the ComposableTransform interface.

These can be tuned for performance.

compose_after_from_vector_inplace(*vector*)

Specialised inplace composition with a vector. This should be overridden to provide specific cases of composition whereby the current state of the transform can be derived purely from the provided vector.

Parameters vector ((n_parameters,) *ndarray*) – Vector to update the transform state with.

VInvertible

class menpo.transform.base.invertible.**VInvertible**

Bases: *Invertible*

Mix-in for *Vectorizable* Invertible *Transform* s.

Prefer this mix-in over Invertible if the *Transform* in question is *Vectorizable* as this adds from_vector() variants to the Invertible interface. These can be tuned for performance, and are, for instance, needed by some of the machinery of fit.

pseudoinverse()

The pseudoinverse of the transform - that is, the transform that results from swapping *source* and *target*, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

Type type(self)

pseudoinverse_vector(*vector*)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```
self.from_vector(vector).pseudoinverse().as_vector()
```

Can be much faster than the explict call as object creation can be entirely avoided in some cases.

Parameters vector ((n_parameters,) *ndarray*) – A vectorized version of self

Returns pseudoinverse_vector ((n_parameters,) *ndarray*) – The pseudoinverse of the vector provided

has_true_inverse

True if the pseudoinverse is an exact inverse.

Type *bool*

2.10 menpo.visualize

2.10.1 Abstract Classes

Renderer

class menpo.visualize.**Renderer**(*figure_id*, *new_figure*)

Bases: object

Abstract class for rendering visualizations. Framework specific implementations of these classes are made in order to separate implementation cleanly from the rest of the code.

It is assumed that the renderers follow some form of stateful pattern for rendering to Figures. Therefore, the major interface for rendering involves providing a *figure_id* or a *bool* about whether a new figure should be used. If neither are provided then the default state of the rendering engine is assumed to be maintained.

Providing both a figure_id and new_figure == True is not a valid state.

Parameters

- **figure_id** (*object*) – A figure id. Could be any valid object that identifies a figure in a given framework (*str*, *int*, *float*, etc.).
- **new_figure** (*bool*) – Whether the rendering engine should create a new figure.

RaisesValueError – It is not valid to provide a figure id AND request a new figure to be rendered on.

get_figure()

Abstract method for getting the correct figure to render on. Should also set the correct *figure_id* for the figure.

Returnsfigure (*object*) – The figure object that the renderer will render on.

render(***kwargs*)

Abstract method to be overridden by the renderer. This will implement the actual rendering code for a given object class.

Parameterskwargs (*dict*) – Passed through to specific rendering engine.

Returnsviewer (*Renderer*) – Pointer to *self*.

save_figure(***kwargs*)

Abstract method for saving the figure of the current *figure_id* to file. It will implement the actual saving code for a given object class.

Parameterskwargs (*dict*) – Options to be set when saving the figure to file.

Viewable

class menpo.visualize.**Viewable**

Bases: object

Abstract interface for objects that can visualize themselves. This assumes that the class has dimensionality as the view method checks the n_dims property to wire up the correct view method.

LandmarkableViewable

class menpo.visualize.**LandmarkableViewable**

Bases: object

Mixin for *Landmarkable* and *Viewable* objects. Provides a single helper method for viewing Landmarks and *self* on the same figure.

MatplotlibRenderer

class `menpo.visualize.`**`MatplotlibRenderer`**(*figure_id*, *new_figure*)
Bases: `Renderer`

Abstract class for rendering visualizations using Matplotlib.

Parameters

- **`figure_id`** (*int* or `None`) – A figure id or `None`. `None` assumes we maintain the Matplotlib state machine and use *plt.gcf()*.
- **`new_figure`** (*bool*) – If `True`, it creates a new figure to render on.

`get_figure`()
Gets the figure specified by the combination of `self.figure_id` and `self.new_figure`. If `self.figure_id == None` then `plt.gcf()` is used. `self.figure_id` is also set to the correct id of the figure if a new figure is created.

Returns **figure** (*Matplotlib figure object*) – The figure we will be rendering on.

`render`(***kwargs*)
Abstract method to be overridden by the renderer. This will implement the actual rendering code for a given object class.

Parameters **`kwargs`** (*dict*) – Passed through to specific rendering engine.

Returns **viewer** (`Renderer`) – Pointer to *self*.

`save_figure`(*filename*, *format='png'*, *dpi=None*, *face_colour='w'*, *edge_colour='w'*, *orientation='portrait'*, *paper_type='letter'*, *transparent=False*, *pad_inches=0.1*, *overwrite=False*)
Method for saving the figure of the current *figure_id* to file.

Parameters

- **`filename`** (*str* or *file*-like object) – The string path or file-like object to save the figure at/into.
- **`format`** (*str*) – The format to use. This must match the file path if the file path is a *str*.
- **`dpi`** (*int* > 0 or `None`, optional) – The resolution in dots per inch.
- **`face_colour`** (*`See Below, optional`*) – The face colour of the figure rectangle. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of len 3
```

- **`edge_colour`** (*`See Below, optional`*) – The edge colour of the figure rectangle. Example options

```
{``r``, ``g``, ``b``, ``c``, ``m``, ``k``, ``w``}
or
``(3, )`` `ndarray`
or
`list` of len 3
```

- **`orientation`** ({`portrait`, `landscape`}, optional) – The page orientation.
- **`paper_type`** (*`See Below, optional`*) – The type of the paper. Example options

```
{``letter``, ``legal``, ``executive``, ``ledger``,
 ``a0`` through ``a10``, ``b0`` through ``b10``}
```

- **transparent** (*bool*, optional) – If `True`, the axes patches will all be transparent; the figure patch will also be transparent unless *face_colour* and/or *edge_colour* are specified. This is useful, for example, for displaying a plot on top of a coloured background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.
- **pad_inches** (*float*, optional) – Amount of padding around the figure.
- **overwrite** (*bool*, optional) – If `True`, the file will be overwritten if it already exists.

save_figure_widget()

Method for saving the figure of the current `figure_id` to file using *menpowidgets.base.save_matplotlib_figure* widget.

2.10.2 Patches

view_patches

menpo.visualize.**view_patches**(*patches*, *patch_centers*, *patches_indices=None*, *offset_index=None*, *figure_id=None*, *new_figure=False*, *background='white'*, *render_patches=True*, *channels=None*, *interpolation='none'*, *cmap_name=None*, *alpha=1.0*, *render_patches_bboxes=True*, *bboxes_line_colour='r'*, *bboxes_line_style='-'*, *bboxes_line_width=1*, *render_centers=True*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour=None*, *marker_edge_width=1.0*, *render_numbering=False*, *numbers_horizontal_align='center'*, *numbers_vertical_align='bottom'*, *numbers_font_name='sans-serif'*, *numbers_font_size=10*, *numbers_font_style='normal'*, *numbers_font_weight='normal'*, *numbers_font_colour='k'*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *axes_x_limits=None*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *figure_size=(10, 8)*)

Method that renders the provided *patches* on a canvas. The user can choose whether to render the patch centers (*render_centers*) as well as rectangle boundaries around the patches (*render_patches_bboxes*).

The patches argument can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods of the `Image` class. Specifically it can be:

1. `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray*
2. *list* of `n_center * n_offset` `Image` objects

Parameters

- **patches** (*ndarray* or *list*) – The values of the patches. It can have any of the two formats that are returned from the *extract_patches()* and *extract_patches_around_landmarks()* methods. Specifically, it can either be an `(n_center, n_offset, self.n_channels, patch_shape)` *ndarray* or a *list* of `n_center * n_offset` `Image` objects.
- **patch_centers** (`PointCloud`) – The centers around which to visualize the patches.
- **patches_indices** (*int* or *list* of *int* or `None`, optional) – Defines the patches that will be visualized. If `None`, then all the patches are selected.
- **offset_index** (*int* or `None`, optional) – The offset index within the provided *patches* argument, thus the index of the second dimension from which to sample. If

None, then 0 is used.

- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If True, a new figure is created.
- **background** ({'black', 'white'}, optional) – If 'black', then the background is set equal to the minimum value of *patches*. If 'white', then the background is set equal to the maximum value of *patches*.
- **render_patches** (*bool*, optional) – Flag that determines whether to render the patch values.
- **channels** (*int* or *list* of *int* or all or None, optional) – If *int* or *list* of *int*, the specified channel(s) will be rendered. If all, all the channels will be rendered in subplots. If None and the image is RGB, it will be rendered in RGB mode. If None and the image is not RGB, it is equivalent to all.
- **interpolation** (*See Below, optional*) – The interpolation used to render the image. For example, if bilinear, the image will be smooth and if nearest, the image will be pixelated. Example options

```
{none, nearest, bilinear, bicubic, spline16, spline36, hanning,
hamming, hermite, kaiser, quadric, catrom, gaussian, bessel,
mitchell, sinc, lanczos}
```

- **cmap_name** (*str*, optional,) – If None, single channel and three channel images default to greyscale and rgb colormaps respectively.
- **alpha** (*float*, optional) – The alpha blending value, between 0 (transparent) and 1 (opaque).
- **render_patches_bboxes** (*bool*, optional) – Flag that determines whether to render the bounding box lines around the patches.
- **bboxes_line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **bboxes_line_style** ({-, --, -., :}, optional) – The style of the lines.
- **bboxes_line_width** (*float*, optional) – The width of the lines.
- **render_centers** (*bool*, optional) – Flag that determines whether to render the patch centers.
- **render_lines** (*bool*, optional) – If True, the edges will be rendered.
- **line_colour** (*See Below, optional*) – The colour of the lines. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** ({-, --, -., :}, optional) – The style of the lines.
- **line_width** (*float*, optional) – The width of the lines.
- **render_markers** (*bool*, optional) – If True, the markers will be rendered.
- **marker_style** (*See Below, optional*) – The style of the markers. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the markers in points.
- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the markers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The width of the markers' edge.
- **render_numbering** (*bool*, optional) – If True, the landmarks will be numbered.
- **numbers_horizontal_align** ({center, right, left}, optional) – The horizontal alignment of the numbers' texts.
- **numbers_vertical_align** ({center, top, bottom, baseline}, optional) – The vertical alignment of the numbers' texts.
- **numbers_font_name** (*See Below, optional*) – The font of the numbers. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **numbers_font_size** (*int*, optional) – The font size of the numbers.
- **numbers_font_style** ({normal, italic, oblique}, optional) – The font style of the numbers.
- **numbers_font_weight** (*See Below, optional*) – The font weight of the numbers. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold, demibold, demi, bold, heavy, extra bold, black}
```

- **numbers_font_colour** (*See Below, optional*) – The font colour of the numbers. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **render_axes** (*bool*, optional) – If True, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** ({normal, italic, oblique}, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **axes_x_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the shape as a percentage of the shape's width. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.

- **axes_y_limits** (*float* or (*float*, *float*) or None, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the shape as a percentage of the shape's height. If *tuple* or *list*, then it defines the axis limits. If None, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or None, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or None, optional) – The ticks of the y axis.
- **figure_size** ((*float*, *float*) *tuple* or None optional) – The size of the figure in inches.

Returnsviewer (*ImageViewer*) – The image viewing object.

2.10.3 Print Utilities

print_progress

menpo.visualize.**print_progress**(*iterable*, *prefix=''*, *n_items=None*, *offset=0*, *show_bar=True*, *show_count=True*, *show_eta=True*, *end_with_newline=True*)

Print the remaining time needed to compute over an iterable.

To use, wrap an existing iterable with this function before processing in a for loop (see example).

The estimate of the remaining time is based on a moving average of the last 100 items completed in the loop.

Parameters

- **iterable** (*iterable*) – An iterable that will be processed. The iterable is passed through by this function, with the time taken for each complete iteration logged.
- **prefix** (*str*, optional) – If provided a string that will be prepended to the progress report at each level.
- **n_items** (*int*, optional) – Allows for iterator to be a generator whose length will be assumed to be *n_items*. If not provided, then iterator needs to be *Sizable*.
- **offset** (*int*, optional) – Useful in combination with n_items - report back the progress as if *offset* items have already been handled. n_items will be left unchanged.
- **show_bar** (*bool*, optional) – If False, The progress bar (e.g. [=========]) will be hidden.
- **show_count** (*bool*, optional) – If False, The item count (e.g. (4/25)) will be hidden.
- **show_eta** (*bool*, optional) – If False, The estimated time to finish (e.g. - 00:00:03 remaining) will be hidden.
- **end_with_newline** (*bool*, optional) – If False, there will be no new line added at the end of the dynamic printing. This means the next print statement will overwrite the dynamic report presented here. Useful if you want to follow up a print_progress with a second print_progress, where the second overwrites the first on the same line.

RaisesValueError – offset provided without n_items

Examples

This for loop:

```
from time import sleep
for i in print_progress(range(100)):
    sleep(1)
```

prints a progress report of the form:

```
[=============       ] 70% (7/10) - 00:00:03 remaining
```

print_dynamic

menpo.visualize.**print_dynamic**(*str_to_print*)

Prints dynamically the provided *str*, i.e. the *str* is printed and then the buffer gets flushed.

Parameters **str_to_print** (*str*) – The string to print.

progress_bar_str

menpo.visualize.**progress_bar_str**(*percentage*, *bar_length=20*, *bar_marker='='*, *show_bar=True*)

Returns an *str* of the specified progress percentage. The percentage is represented either in the form of a progress bar or in the form of a percentage number. It can be combined with the `print_dynamic()` function.

Parameters

- **percentage** (*float*) – The progress percentage to be printed. It must be in the range `[0, 1]`.
- **bar_length** (*int*, optional) – Defines the length of the bar in characters.
- **bar_marker** (*str*, optional) – Defines the marker character that will be used to fill the bar.
- **show_bar** (*bool*, optional) – If `True`, the *str* includes the bar followed by the percentage, e.g. `'[=====     ] 50%'`

  If `False`, the *str* includes only the percentage, e.g. `'50%'`

Returns **progress_str** (*str*) – The progress percentage string that can be printed.

Raises

- `ValueError` – `percentage` is not in the range `[0, 1]`
- `ValueError` – `bar_length` must be an integer >= `1`
- `ValueError` – `bar_marker` must be a string of length 1

Examples

This for loop:

```
n_iters = 2000
for k in range(n_iters):
    print_dynamic(progress_bar_str(float(k) / (n_iters-1)))
```

prints a progress bar of the form:

```
[=============       ] 68%
```

bytes_str

menpo.visualize.**bytes_str**(*num*)

Converts bytes to a human readable format. For example:

```
print_bytes(12345) returns '12.06 KB'
print_bytes(123456789) returns '117.74 MB'
```

Parameters **num** (*int*) – The size in bytes.

Raises `ValueError` – num must be int >= 0

2.10.4 Various

plot_curve

menpo.visualize.**plot_curve**(*x_axis*, *y_axis*, *figure_id=None*, *new_figure=True*, *legend_entries=None*, *title=''*, *x_label=''*, *y_label=''*, *axes_x_limits=0.0*, *axes_y_limits=None*, *axes_x_ticks=None*, *axes_y_ticks=None*, *render_lines=True*, *line_colour=None*, *line_style='-'*, *line_width=1*, *render_markers=True*, *marker_style='o'*, *marker_size=5*, *marker_face_colour=None*, *marker_edge_colour='k'*, *marker_edge_width=1.0*, *render_legend=True*, *legend_title=''*, *legend_font_name='sans-serif'*, *legend_font_style='normal'*, *legend_font_size=10*, *legend_font_weight='normal'*, *legend_marker_scale=None*, *legend_location=2*, *legend_bbox_to_anchor=(1.05, 1.0)*, *legend_border_axes_pad=None*, *legend_n_columns=1*, *legend_horizontal_spacing=None*, *legend_vertical_spacing=None*, *legend_border=True*, *legend_border_padding=None*, *legend_shadow=False*, *legend_rounded_corners=False*, *render_axes=True*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *figure_size=(10, 8)*, *render_grid=True*, *grid_line_style='–'*, *grid_line_width=1*)

Plot a single or multiple curves on the same figure.

Parameters

- **x_axis** (*list* or *array*) – The values of the horizontal axis. They are common for all curves.
- **y_axis** (*list* of *lists* or *arrays*) – A *list* with *lists* or *arrays* with the values of the vertical axis for each curve.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **legend_entries** (*list of 'str* or `None`, optional) – If *list* of *str*, it must have the same length as *errors list* and each *str* will be used to name each curve. If `None`, the CED curves will be named as *'Curve %d'*.
- **title** (*str*, optional) – The figure's title.
- **x_label** (*str*, optional) – The label of the horizontal axis.
- **y_label** (*str*, optional) – The label of the vertical axis.
- **axes_x_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the x axis. If *float*, then it sets padding on the right and left of the graph as a percentage of the curves' width. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_y_limits** (*float* or (*float*, *float*) or `None`, optional) – The limits of the y axis. If *float*, then it sets padding on the top and bottom of the graph as a percentage of the curves' height. If *tuple* or *list*, then it defines the axis limits. If `None`, then the limits are set automatically.
- **axes_x_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the x axis.
- **axes_y_ticks** (*list* or *tuple* or `None`, optional) – The ticks of the y axis.
- **render_lines** (*bool* or *list* of *bool*, optional) – If `True`, the line will be rendered. If *bool*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **line_colour** (*colour* or *list* of *colour* or `None`, optional) – The colour of the lines. If not a *list*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*. If `None`, the colours will be linearly sampled from jet colormap. Example *colour* options are

```
{'r', 'g', 'b', 'c', 'm', 'k', 'w'}
or
(3, ) ndarray
```

- **line_style** ({'-', '--', '-.', ':'} or *list* of those, optional) – The style of the lines. If not a *list*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **line_width** (*float* or *list* of *float*, optional) – The width of the lines. If *float*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **render_markers** (*bool* or *list* of *bool*, optional) – If `True`, the markers will be rendered. If *bool*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **marker_style** (*marker* or *list* of *markers*, optional) – The style of the markers. If not a *list*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*. Example *marker* options

```
{'.', ',', 'o', 'v', '^', '<', '>', '+', 'x', 'D', 'd', 's',
 'p', '*', 'h', 'H', '1', '2', '3', '4', '8'}
```

- **marker_size** (*int* or *list* of *int*, optional) – The size of the markers in points. If *int*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **marker_face_colour** (*colour* or *list* of *colour* or `None`, optional) – The face (filling) colour of the markers. If not a *list*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*. If `None`, the colours will be linearly sampled from jet colormap. Example *colour* options are

```
{'r', 'g', 'b', 'c', 'm', 'k', 'w'}
or
(3, ) ndarray
```

- **marker_edge_colour** (*colour* or *list* of *colour* or `None`, optional) – The edge colour of the markers. If not a *list*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*. If `None`, the colours will be linearly sampled from jet colormap. Example *colour* options are

```
{'r', 'g', 'b', 'c', 'm', 'k', 'w'}
or
(3, ) ndarray
```

- **marker_edge_width** (*float* or *list* of *float*, optional) – The width of the markers' edge. If *float*, this value will be used for all curves. If *list*, a value must be specified for each curve, thus it must have the same length as *y_axis*.
- **render_legend** (*bool*, optional) – If `True`, the legend will be rendered.
- **legend_title** (*str*, optional) – The title of the legend.
- **legend_font_name** (*See below, optional*) – The font of the legend. Example options

```
{'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
```

- **legend_font_style** ({'normal', 'italic', 'oblique'}, optional) – The font style of the legend.
- **legend_font_size** (*int*, optional) – The font size of the legend.
- **legend_font_weight** (*See below, optional*) – The font weight of the legend. Example options

```
{'ultralight', 'light', 'normal', 'regular', 'book', 'medium',
 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy',
 'extra bold', 'black'}
```

- **legend_marker_scale** (*float*, optional) – The relative size of the legend markers with respect to the original
- **legend_location** (*int*, optional) – The location of the legend. The predefined values are:

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

- **legend_bbox_to_anchor** ((*float*, *float*), optional) – The bbox that the legend will be anchored.
- **legend_border_axes_pad** (*float*, optional) – The pad between the axes and legend border.
- **legend_n_columns** (*int*, optional) – The number of the legend's columns.
- **legend_horizontal_spacing** (*float*, optional) – The spacing between the columns.
- **legend_vertical_spacing** (*float*, optional) – The vertical space between the legend entries.
- **legend_border** (*bool*, optional) – If `True`, a frame will be drawn around the legend.
- **legend_border_padding** (*float*, optional) – The fractional whitespace inside the legend border.
- **legend_shadow** (*bool*, optional) – If `True`, a shadow will be drawn behind legend.
- **legend_rounded_corners** (*bool*, optional) – If `True`, the frame's corners will be rounded (fancybox).
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (`See below, optional`) – The font of the axes. Example options

```
{'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{'normal', 'italic', 'oblique'}`, optional) – The font style of the axes.
- **axes_font_weight** (`See below, optional`) – The font weight of the axes. Example options

```
{'ultralight', 'light', 'normal', 'regular', 'book', 'medium',
 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy',
 'extra bold', 'black'}
```

- **figure_size** ((*float*, *float*) or `None`, optional) – The size of the figure in inches.
- **render_grid** (*bool*, optional) – If `True`, the grid will be rendered.
- **grid_line_style** (`{'-', '--', '-.', ':'}`, optional) – The style of the grid lines.

- **grid_line_width** (*float*, optional) – The width of the grid lines.

Raises `ValueError` – legend_entries list has different length than y_axis list

Returns **viewer** (`GraphPlotter`) – The viewer object.

plot_gaussian_ellipses

`menpo.visualize.`**`plot_gaussian_ellipses`**(*covariances*, *means*, *n_std=2*, *render_colour_bar=True*, *colour_bar_label='Normalized Standard Deviation'*, *colour_map='jet'*, *figure_id=None*, *new_figure=False*, *image_view=True*, *line_colour='r'*, *line_style='-'*, *line_width=1.0*, *render_markers=True*, *marker_edge_colour='k'*, *marker_face_colour='k'*, *marker_edge_width=1.0*, *marker_size=5*, *marker_style='o'*, *render_axes=False*, *axes_font_name='sans-serif'*, *axes_font_size=10*, *axes_font_style='normal'*, *axes_font_weight='normal'*, *crop_proportion=0.1*, *figure_size=(10, 8)*)

Method that renders the Gaussian ellipses that correspond to a set of covariance matrices and mean vectors. Naturally, this only works for 2-dimensional random variables.

Parameters

- **covariances** (*list* of `(2, 2)` *ndarray*) – The covariance matrices that correspond to each ellipse.
- **means** (*list* of `(2, )` *ndarray*) – The mean vectors that correspond to each ellipse.
- **n_std** (*float*, optional) – This defines the size of the ellipses in terms of number of standard deviations.
- **render_colour_bar** (*bool*, optional) – If `True`, then the ellipses will be coloured based on their normalized standard deviations and a colour bar will also appear on the side. If `False`, then all the ellipses will have the same colour.
- **colour_bar_label** (*str*, optional) – The title of the colour bar. It only applies if *render_colour_bar* is `True`.
- **colour_map** (*str*, optional) – A valid Matplotlib colour map. For more info, please refer to *matplotlib.cm*.
- **figure_id** (*object*, optional) – The id of the figure to be used.
- **new_figure** (*bool*, optional) – If `True`, a new figure is created.
- **image_view** (*bool*, optional) – If `True` the ellipses will be rendered in the image coordinates system.
- **line_colour** (`See Below, optional`) – The colour of the lines of the ellipses. Example options:

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **line_style** (`{-, --, -., :}`, optional) – The style of the lines of the ellipses.
- **line_width** (*float*, optional) – The width of the lines of the ellipses.
- **render_markers** (*bool*, optional) – If `True`, the centers of the ellipses will be rendered.
- **marker_style** (`See Below, optional`) – The style of the centers of the ellipses. Example options

```
{., ,, o, v, ^, <, >, +, x, D, d, s, p, *, h, H, 1, 2, 3, 4, 8}
```

- **marker_size** (*int*, optional) – The size of the centers of the ellipses in points.

- **marker_face_colour** (*See Below, optional*) – The face (filling) colour of the centers of the ellipses. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_colour** (*See Below, optional*) – The edge colour of the centers of the ellipses. Example options

```
{r, g, b, c, m, k, w}
or
(3, ) ndarray
```

- **marker_edge_width** (*float*, optional) – The edge width of the centers of the ellipses.
- **render_axes** (*bool*, optional) – If `True`, the axes will be rendered.
- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options

```
{serif, sans-serif, cursive, fantasy, monospace}
```

- **axes_font_size** (*int*, optional) – The font size of the axes.
- **axes_font_style** (`{normal, italic, oblique}`, optional) – The font style of the axes.
- **axes_font_weight** (*See Below, optional*) – The font weight of the axes. Example options

```
{ultralight, light, normal, regular, book, medium, roman,
semibold,demibold, demi, bold, heavy, extra bold, black}
```

- **crop_proportion** (*float*, optional) – The proportion to be left around the centers' pointcloud.
- **figure_size** ((*float*, *float*) *tuple* or `None` optional) – The size of the figure in inches.

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z