## CONTENTS

1 API Documentation ................................................. 3  
1.1 menpo.base .................................................. 3  
1.2 menpo.io ..................................................... 8  
1.3 menpo.image .................................................. 18  
1.4 menpo.feature ............................................... 98  
1.5 menpo.landmark ............................................. 112  
1.6 menpo.math .................................................. 144  
1.7 menpo.model .................................................. 149  
1.8 menpo.shape ............................................... 180  
1.9 menpo.transform ............................................ 322  
1.10 menpo.visualize ............................................ 396  
1.11 Changelog ................................................... 409  

Index ............................................................... 429
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:

```python
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.

Finally, please refer to Menpo’s Changelog for a list of changes per release.
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.

1.1 menpo.base

1.1.1 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.

**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.

**property 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.

**property n_dims**
The number of dimensions of the `target`.

**Type** int

**property n_points**
The number of points on the `target`.

**Type** int

**property target**
The current `PointCloud` that this object produces.

**Type** PointCloud

**LazyList**

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

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.

**copy()**
Generate an efficient copy of this LazyList - copying the underlying callables will be lazy and shallow (each callable will not be called nor copied) but they will reside within in a new list.

**Returns** type(self) – A copy of this LazyList.

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

**index** *(value[, start[, stop]])* → 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.

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

classmethod **init_from_iterable** (*iterable*, *f=None*)

Create a lazy list from an existing iterable (think Python list) and optionally a *callable* that expects a single parameter which will be applied to each element of the list. This allows for simply creating a LazyList from an existing list and if no *callable* is provided the identity function is assumed.

**Parameters**

- **iterable** (*collections.Iterable*) – An iterable object such as a list.
- **f** (*callable*, optional) – Callable expecting a single parameter.

**Returns** lazy (*LazyList*) – A LazyList where each element returns each item of the provided iterable, optionally with *f* applied to it.

**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.

Alternatively, *f* may be a list of *callable*, one per entry in the underlying list, with the same specification as above.

**Parameters** *f* (*callable* or iterable of *callable*) – Callable to wrap each element with. If an iterable of callables (think list) is passed then it must be the same length as this LazyList.

**Returns** lazy (*LazyList*) – A new LazyList where each element is wrapped by (each) *f*.

**repeat** (*n*)

Repeat each item of the underlying LazyList *n* times. Therefore, if a list currently has *D* items, the returned list will contain *D* × *n* items and will return immediately (method is lazy).

**Parameters** *n* (*int*) – The number of times to repeat each item.

**Returns** lazy (*LazyList*) – A LazyList where each element returns each item of the provided iterable, optionally with *f* applied to it.

**Examples**

```python
>>> from menpo.base import LazyList
>>> ll = LazyList.init_from_list([0, 1])
>>> repeated_ll = ll.repeat(2)  # Returns immediately
>>> items = list(repeated_ll)  # [0, 0, 1, 1]
```

**view_widget** ()

Visualize this lazy collection of items using menpowidgets.

The type of the first item will be used to determine an appropriate visualization for the list of items.
Returns widget – The appropriate menpowidget to view these items

Raises

• MenpowidgetsMissingError – If menpowidgets is not installed
• ValueError – If menpowidgets cannot locate an appropriate items-visualization for the type of items in this LazyList

1.1.2 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.

Returns path (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.

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

Returns name (str) – The name of the passed object.

1.1.3 Warnings and Exceptions

MenpoDeprecationWarning

class menpo.base.MenpoDeprecationWarning

Bases: Warning

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

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

MenpoMissingDependencyError

class menpo.base.MenpoMissingDependencyError (package_name)

Bases: ImportError

An exception that a dependency required for the requested functionality was not detected.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

msg

exception message
1.2 menpo.io

1.2.1 Input

import_image

menpo.io.import_image(filepath, landmark_resolver=<function same_name>, normalize=None, normalise=None)

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 or None, 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'}
- **normalize** (bool, optional) – If True, normalize 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.
- **normalise** (bool, optional) – Deprecated version of normalize. Please use the normalize arg.

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>, normalize=None, normalise=None, 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 \texttt{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 \texttt{LazyList} in a Python \texttt{list}.

**Parameters**

- \texttt{pattern} (\texttt{str}) – A glob path pattern to search for images. Every image found to match the glob will be imported one by one. See \texttt{image_paths} for more details of what images will be found.

- \texttt{max\_images} (positive \texttt{int}, optional) – If not \texttt{None}, only import the first \texttt{max\_images} found. Else, import all.

- \texttt{shuffle} (\texttt{bool}, optional) – If \texttt{True}, the order of the returned images will be randomised. If \texttt{False}, the order of the returned images will be alphanumerically ordered.

- \texttt{landmark\_resolver} (\texttt{function} or \texttt{None}, 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': \texttt{landmark\_filepath}'\}. Default finds landmarks with the same name as the image file. If \texttt{None}, landmark importing will be skipped.

- \texttt{normalize} (\texttt{bool}, optional) – If \texttt{True}, normalize the image pixels between 0 and 1 and convert to floating point. If false, the native datattype of the image will be maintained (commonly \texttt{uint8}). Note that in general Menpo assumes \texttt{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.

- \texttt{normalise} (\texttt{bool}, optional) – Deprecated version of \texttt{normalize}. Please use the \texttt{normalize} arg.

- \texttt{as\_generator} (\texttt{bool}, optional) – If \texttt{True}, the function returns a generator and assets will be yielded one after another when the generator is iterated over.

- \texttt{verbose} (\texttt{bool}, optional) – If \texttt{True} progress of the importing will be dynamically reported with a progress bar.

**Returns** \texttt{lazy\_list} (\texttt{LazyList} or generator of \texttt{Image}) – A \texttt{LazyList} or generator yielding \texttt{Image} instances found to match the glob pattern provided.

**Raises** \texttt{ValueError} – If no images are found at the provided glob.

**Examples**

Import images at 20% scale from a huge collection:

```python
>>> rescale_20p = lambda x: x.rescale(0.2)
>>> images = menpo.io.import_images('./massive_image_db/*')  # Returns immediately
>>> images = 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>, normalize=None, normalise=None, importer_method='ffmpeg', exact_frame_count=True)

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 ffmpeg to perform the importing. In order to recover accurate frame counts from videos it is necessary to use ffprobe to count the frames. This involves reading the entire video in to memory which may cause a delay in loading despite the lazy nature of the video loading within Menpo. If ffprobe cannot be found, and exact_frame_count is False, Menpo falls back to ffmpeg itself which is not accurate and the user should proceed at their own risk.

**Parameters**

- **filepath** (pathlib.Path or str) – A relative or absolute filepath to a video file.

- **landmark_resolver** (function or None, 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}'. If None, landmark importing will be skipped.

- **normalize** (bool, optional) – If True, normalize 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.

- **normalise** (bool, optional) – Deprecated version of normalize. Please use the normalize arg.

- **importer_method** ({'ffmpeg'}, optional) – A string representing the type of importer to use, by default ffmpeg is used.

- **exact_frame_count** (bool, optional) – If True, the import fails if ffprobe is not available (reading from ffmpeg’s output returns inexact frame count)

**Returns** frames (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**

```python
>>> 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>, normalize=None, normalise=None, importer_method='ffmpeg', exact_frame_count=True, 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 ffmpeg to perform the importing. In order to recover accurate frame counts from videos it is necessary to use ffprobe to count the frames. This involves reading the entire video in to memory which may cause a delay in loading despite the lazy nature of the video loading within Menpo. If ffprobe cannot be found, and exact_frame_count is False, Menpo falls back to ffmpeg itself which is not accurate and the user should proceed at their 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 or None, 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}'. If None, landmark importing will be skipped.
- **normalize** (bool, optional) – If True, normalize 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.
- **normalise** (bool, optional) – Deprecated version of normalize. Please use the normalize arg.
- **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.
- **exact_frame_count** (bool, optional) – If True, the import fails if ffmeprobe is not available (reading from ffmpeg’s output returns inexact frame count)
• **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:

```python
>>> 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, group=None, asset=None)*

Single landmark file importer.

If a landmark file is found at filepath, returns a dictionary of landmarks where keys are the group names and the values are PointCloud or subclasses. If the optional group argument is supplied then a single group with the given name is returned rather than a dictionary.

**Parameters**

- **filepath** *(pathlib.Path or str)* – A relative or absolute filepath to an landmark file.
- **group** *(str, optional)* – The name of the landmark group to return from the landmark dictionary. If None, then a dictionary is returned where keys are the group names and the values are PointCloud or subclasses.
- **asset** *(object, optional)* – The object the landmark belongs to (useful for things like rescaling)

**Returns** landmarks *(dict {str: PointCloud} or PointCloud)* – Dictionary mapping landmark groups to PointCloud or subclasses OR PointCloud or subclass if group == None

### 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 then returns a LabelledPointUndirectedGraph or a PointCloud.

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)* – A `LazyList` or generator yielding `PointCloud` or `LabelledPointUndirectedGraph` instances found to match the glob pattern provided.

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

---

**import_pickle**

```python
menpo.io.import_pickle(filepath, **kwargs)
```

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**

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

Multiple pickle importer.

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.

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

**Parameters**

- **pattern** *(str)* – A glob path pattern to search for pickles. Every pickle found to match the glob will be imported one by one. See `pickle_paths` for more details of what pickles will be found.

- **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 with a progress bar.

**Returns** `lazy_list` *(LazyList or generator of Python objects)* – A LazyList or generator yielding whatever Python objects are present in the Pickle file instances that match the glob pattern provided.

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

### `import_builtin_asset`

```python
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:

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

### `register_image_importer`

```python
menpo.io.register_image_importer(extension, callable)
```

Register a new importer for the given extension.

**Parameters**

• **ext_map** *(‘str’ -> ‘callable’ dict)* – Extensions map to callable.

• **extension** *(str)* – File extension to support. May be multi-part e.g. `.tar.gz`

• **callable** *(callable)* – The callable to invoke if a file with the provided extension is discovered during importing. Should take a single argument (the filepath) and any number of kwargs.

### `register_landmark_importer`

```python
menpo.io.register_landmark_importer(extension, callable)
```

Register a new importer for the given extension.

**Parameters**

• **ext_map** *(‘str’ -> ‘callable’ dict)* – Extensions map to callable.

• **extension** *(str)* – File extension to support. May be multi-part e.g. `.tar.gz`

• **callable** *(callable)* – The callable to invoke if a file with the provided extension is discovered during importing. Should take a single argument (the filepath) and any number of kwargs.
**register_pickle_importer**

`menpo.io.register_pickle_importer(extension, callable)`  
Register a new importer for the given extension.

**Parameters**

- **ext_map** (`{str -> callable}` dict) – Extensions map to callable.
- **extension** (str) – File extension to support. May be multi-part e.g. ‘.tar.gz’
- **callable** (callable) – The callable to invoke if a file with the provided extension is discovered during importing. Should take a single argument (the filepath) and any number of kwargs.

**register_video_importer**

`menpo.io.register_video_importer(extension, callable)`  
Register a new importer for the given extension.

**Parameters**

- **ext_map** (`{str -> callable}` dict) – Extensions map to callable.
- **extension** (str) – File extension to support. May be multi-part e.g. ‘.tar.gz’
- **callable** (callable) – The callable to invoke if a file with the provided extension is discovered during importing. Should take a single argument (the filepath) and any number of kwargs.

## 1.2.2 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_video

`menpo.io.export_video(images, file_path, overwrite=False, fps=30, **kwargs)`  
Exports a given list of images as a video. Ensure that all the images have the same shape, otherwise you might get unexpected results from the ffmpeg writer. The `file_path` argument is a `Path` representing the path to save the video to. At this time, it is not possible to export videos directly to a file buffer.

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

Note that exporting of GIF images is also supported.

**Parameters**

- **images** (list of `Image`) – The images to export as a video.
- **file_path** (`Path`) – The Path to save the video at. File buffers are not supported, unlike other exporting formats.
- **overwrite** (`bool`, optional) – Whether or not to overwrite a file if it already exists.
- **fps** (`int`, optional) – The number of frames per second.
- ****kwargs** (`dict`, optional) – Extra parameters that are passed through directly to the exporter. Please see the documentation in the `menpo.io.output.video` package for information about the supported arguments.

**Raises**

- **ValueError** – File already exists and `overwrite` ! = `True`
- **ValueError** – The input is a buffer and not a valid `Path`
- **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(landmarks_object, fp, extension=None, overwrite=False)`  
Exports a given shape. The `fp` argument can be either an a `str` or any Python type that acts like a file. If a file is provided, the `extension` keyword 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**

- **landmarks_object** (dict or `LandmarkManager` or subclass of `PointCloud`) – The landmarks to export. The type of `PointCloud` or subclass of it are supported by all exporters, while the rest are available only for the LJSON format.
- **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).
• **ValueError** – The provided type for `landmarks_object` 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 the selected Pickle protocol. If `.pkl.gz` the object will be pickled using the selected Pickle protocol 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:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Simplest protocol for text mode, backwards compatible.</td>
</tr>
<tr>
<td>1</td>
<td>Protocol for binary mode, backwards compatible.</td>
</tr>
<tr>
<td>2</td>
<td>Wider support for classes, compatible with python &gt;= 2.3.</td>
</tr>
<tr>
<td>3</td>
<td>Support for byte objects, compatible with python &gt;= 3.0.</td>
</tr>
<tr>
<td>4</td>
<td>Support for large objects, compatible with python &gt;= 3.4.</td>
</tr>
</tbody>
</table>

#### 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).

### 1.2.3 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 built in ./data folder on this machine.

*Returns*

- **path**(pathlib.Path) – The path to the local ./data folder

ls_builtin_assets

menpo.io.**ls_builtin_assets**(())

List all the builtin asset examples provided.

*Returns*

- **file_paths**(list of str) – Filenames of all assets in the data directory shipped with the project.

1.3 menpo.image

1.3.1 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

```python
_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=(7, 7))
```

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 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 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 or (float, float) or None, optional)* – The size of the figure in inches.

Returns **viewer** *(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_color=None, line_style='-', line_width=1, render_markers=True, marker_style='o', marker_size=5, marker_face_color=None, marker_edge_color=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_weight='normal', numbers_font_style='normal', numbers_font_color='k', render_legend=False, legend_title='', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', 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', axes_x_limits=None, axes_y_limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(7, 7))
```

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.
• \texttt{numbers_horizontal_align} \{(center, right, left), \text{optional}\} – The horizontal alignment of the numbers’ texts.

• \texttt{numbers_vertical_align} \{(center, top, bottom, baseline), \text{optional}\} – The vertical alignment of the numbers’ texts.

• \texttt{numbers_font_name} \{(See Below, \text{optional}\) – The font of the numbers. Example options
  \begin{verbatim}
  \{serif, sans-serif, cursive, fantasy, monospace\}
  \end{verbatim}

• \texttt{numbers_font_size} \{(int, \text{optional}\} – The font size of the numbers.

• \texttt{numbers_font_style} \{(normal, italic, oblique), \text{optional}\} – The font style of the numbers.

• \texttt{numbers_font_weight} \{(See Below, \text{optional}\) – The font weight of the numbers. Example options
  \begin{verbatim}
  \{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black\}
  \end{verbatim}

• \texttt{numbers_font_colour} \{(See Below, \text{optional}\) – The font colour of the numbers. Example options
  \begin{verbatim}
  \{r, g, b, c, m, k, w\}
or
  (3, ) ndarray
  \end{verbatim}

• \texttt{render_legend} \{(bool, \text{optional}\} – If True, the legend will be rendered.

• \texttt{legend_title} \{(str, \text{optional}\} – The title of the legend.

• \texttt{legend_font_name} \{(See below, \text{optional}\) – The font of the legend. Example options
  \begin{verbatim}
  \{serif, sans-serif, cursive, fantasy, monospace\}
  \end{verbatim}

• \texttt{legend_font_style} \{(normal, italic, oblique), \text{optional}\} – The font style of the legend.

• \texttt{legend_font_size} \{(int, \text{optional}\} – The font size of the legend.

• \texttt{legend_font_weight} \{(See Below, \text{optional}\) – The font weight of the legend. Example options
  \begin{verbatim}
  \{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black\}
  \end{verbatim}

• \texttt{legend_marker_scale} \{(float, \text{optional}\} – The relative size of the legend markers with respect to the original

• \texttt{legend_location} \{(int, \text{optional}\} – The location of the legend. The predefined values are:
• **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

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

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)* –

<table>
<thead>
<tr>
<th>mode</th>
<th>Greyscale Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Equal average of all channels</td>
</tr>
<tr>
<td>luminosity</td>
<td>Calculates the luminance using the CCIR 601 formula:</td>
</tr>
<tr>
<td></td>
<td>(Y' = 0.2989R' + 0.5870G' + 0.1140B')</td>
</tr>
<tr>
<td>channel</td>
<td>A specific channel is chosen as the intensity value.</td>
</tr>
</tbody>
</table>

- **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

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** *(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:

```python
>>> hist, bin_edges = image.as_histogram()
>>> for k in range(len(hist)):
...    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=<class '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** *(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

---

**clip_pixels** *(minimum=None, maximum=None)*

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

**Parameters**

- **minimum** *(float, optional)* – The minimal value of the clipped pixels. If None is provided, the default value will be 0.

- **maximum** *(float, optional)* – The maximal value of the clipped pixels. If None is provided, the default value will depend on the dtype.

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

---

**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:

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

\texttt{type(self)} – A copy of this object

crop\((\text{min\_indices}, \text{max\_indices}, \text{constrain\_to\_boundary}=\text{False}, \text{return\_transform}=\text{False})\)

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

• \texttt{min\_indices\((n\_dims,\) ndarray)} – The minimum index over each dimension.

• \texttt{max\_indices\((n\_dims,\) ndarray)} – The maximum index over each dimension.

• \texttt{constrain\_to\_boundary\(\text{bool, optional}\)} – If \texttt{True} the crop will be snapped to not go beyond this images boundary. If \texttt{False}, an \texttt{ImageBoundaryError} will be raised if an attempt is made to go beyond the edge of the image.

• \texttt{return\_transform\(\text{bool, optional}\)} – If \texttt{True}, then the \texttt{Transform} object that was used to perform the cropping is also returned.

Returns

• \texttt{cropped\_image\(\text{type(self)}\)} – A new instance of self, but cropped.

• \texttt{transform\(\text{Transform}\)} – The transform that was used. It only applies if \texttt{return\_transform} is \texttt{True}.

Raises

• \texttt{ValueError}\(\text{min\_indices and max\_indices both have to be of length n\_dims. All max\_indices must be greater than min\_indices.}\)

• \texttt{ImageBoundaryError}\(\text{Raised if constrain\_to\_boundary=False, and an attempt is made to crop the image in a way that violates the image bounds.}\)

crop\_to\_landmarks\((\text{group}=\text{None}, \text{boundary}=0, \text{constrain\_to\_boundary}=\text{True}, \text{return\_transform}=\text{False})\)

Return a copy of this image cropped so that it is bounded around a set of landmarks with an optional \texttt{n\_pixel boundary}.

Parameters

• \texttt{group\(\text{str, optional}\)} – The key of the landmark set that should be used. If \texttt{None} and if there is only one set of landmarks, this set will be used.

• \texttt{boundary\(\text{int, optional}\)} – An extra padding to be added all around the landmarks bounds.

• \texttt{constrain\_to\_boundary\(\text{bool, optional}\)} – If \texttt{True} the crop will be snapped to not go beyond this images boundary. If \texttt{False}, an \texttt{ImageBoundaryError} will be raised if an attempt is made to go beyond the edge of the image.

• \texttt{return\_transform\(\text{bool, optional}\)} – If \texttt{True}, then the \texttt{Transform} object that was used to perform the cropping is also returned.

Returns

• \texttt{image\(\text{Image}\)} – A copy of this image cropped to its landmarks.

• \texttt{transform\(\text{Transform}\)} – The transform that was used. It only applies if \texttt{return\_transform} is \texttt{True}.

1.3. menpo.image
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 image’s 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 image’s 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 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 a border proportional to the pointcloud spread or range.

**Parameters**

- `pointcloud` (*PointCloud*) – The pointcloud to crop around.
- `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.
- `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

```python
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** *(ndarray)* – Flattened representation of this object

```python
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.

```python
has_landmarks_outside_bounds()
```

Indicates whether there are landmarks located outside the image bounds.

**Type** **bool**

```python
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.

```python
indices()
```

Return the indices of all pixels in this image.

**Type** *(n_dims, n_pixels) ndarray*

```python
classmethod init_blank(shape, n_channels=1, fill=0, dtype=<class '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.

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

```python
classmethod init_from_channels_at_back(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.

**Raises** **ValueError** – If image is not at least 2D, i.e. has at least 2 dimensions plus the channels in the end.

**classmethod init_from_pointcloud**

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**

Deprecated - please use the equivalent init_from_channels_at_back method.

**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.

**pixels_with_channels_at_back** ([*out_dtype=None]*)

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.

If this image is single channel, the final axis is dropped.

**Parameters**

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

**Returns**

* rolled_channels ([*ndarray]*) – Pixels with channels as the back (last) axis. If single channel, the last axis will be dropped.

**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** (str, 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]
• **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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 be 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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>
• **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

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

**Returns**

• **resized_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 the number of dimensions of the new shape does not match the number of dimensions of the image.

**rolled_channels()**

Deprecated - please use the equivalent `pixels_with_channels_at_back` method.

**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`.

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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*
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)`
**transform_about_centre** (transform, retain_shape=False, cval=0.0, round='round', order=1, return_transform=False)

Return a copy of this image, transformed about its centre.

Note that the *retain_shape* argument defines the shape of the transformed image. If *retain_shape=True*, then the shape of the transformed 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.

**Note:** This method will not work for transforms that result in a transform chain as *TransformChain* is not invertible.

**Note:** Be careful when defining transforms for warping images. All pixel locations must fall within a valid range as expected by the transform. Therefore, your transformation must accept ‘negative’ pixel locations as the pixel locations provided to your transform will have the object centre subtracted from them.

**Parameters**

- **transform** (*ComposableTransform* and *VInvertible* type) – A composable transform. *pseudoinverse* will be invoked on the resulting transform so it must implement a valid inverse.

- **retain_shape** (bool, optional) – If True, then the shape of the sheared 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 sheared 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*.

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear <em>(default)</em></td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

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

**Returns**

- **transformed_image** (*type(self]*) – The transformed image.

- **transform** (*Transform*) – The transform that was used. It only applies if *return_transform* is True.
Examples

This is an example for rotating an image about its center. Let’s first load an image, create the rotation transform and then apply it:

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Rotation

# Load image
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
rot_tr = Rotation.init_from_2d_ccw_angle(45)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')

# Render rotated image
plt.subplot(132)
im.transform_about_centre(rot_tr).view_landmarks()
plt.title('Rotated')

# Render rotated image that has shape equal as original image
plt.subplot(133)
im.transform_about_centre(rot_tr, retain_shape=True).view_landmarks()
plt.title('Rotated (Retain original shape)')
```

Similarly, in order to apply a shear transform:

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Affine

# Load image
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
shear_tr = Affine.init_from_2d_shear(25, 10)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')

# Render sheared image
plt.subplot(132)
im.transform_about_centre(shear_tr).view_landmarks()
plt.title('Sheared')

# Render sheared image that has shape equal as original image
plt.subplot(133)
im.transform_about_centre(shear_tr, retain_shape=True).view_landmarks()
```

(continues on next page)
plt.title('Sheared (Retain original shape)')

view_widget (figure_size=(7, 7))
Visualizes the image using an interactive widget.

Parameters figure_size ((int, int), optional) – The initial size of the rendered figure.

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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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 zoomed 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.

**property has_landmarks**

Whether the object has landmarks.

Type bool

**property 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

**property landmarks**

The landmarks object.

Type LandmarkManager

**property n_channels**

The number of channels on each pixel in the image.

Type int

**property n_dims**

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

Type int

**property n_elements**

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

Type int

**property n_landmark_groups**

The number of landmark groups on this object.

Type int

**property n_parameters**

The length of the vector that this object produces.

Type int

**property n_pixels**

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

Type int

**property shape**

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

Type tuple

**property 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=<class '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) –
<table>
<thead>
<tr>
<th><strong>mode</strong></th>
<th><strong>Greyscale Algorithm</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Equal average of all channels</td>
</tr>
<tr>
<td>luminosity</td>
<td>Calculates the luminance using the CCIR 601 formula: $Y' = 0.2989R' + 0.5870G' + 0.1140B'$</td>
</tr>
<tr>
<td>channel</td>
<td>A specific channel is chosen as the intensity value.</td>
</tr>
</tbody>
</table>

- **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:

```python
>>> 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=<class '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)

**clip_pixels** (*minimum=None, maximum=None*)

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

**Parameters**

• **minimum** (*float*, optional) – The minimal value of the clipped pixels. If None is provided, the default value will be 0.

• **maximum** (*float*, optional) – The maximal value of the clipped pixels. If None is provided, the default value will depend on the dtype.

**Returns**

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

**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:

```python
>>> im.constrain_landmarks_to_bounds()  # Equivalent to below
>>> im.landmarks['test'] = im.landmarks['test'].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. Triangulation 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 `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 image’s 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 image’s 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 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 a border proportional to the pointcloud spread or range.

**Parameters**

- **pointcloud** *(PointCloud)* – The pointcloud to crop around.
- **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.
- **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 False.

**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_channels_at_back(pixels)

This method is not required for BooleanImage types as boolean images do not expect a channel axis for construction.

Parameters pixels ((M, N ..., Q) ndarray) – Array representing the image pixels, with NO channel axis.

Returns image(BooleanImage) – A new image from the given boolean pixels.
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 con-
  strained 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.

classmethod init_from_rolled_channels(pixels)

Deprecated - please use the equivalent init_from_channels_at_back method.

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 \texttt{int} \\
\texttt{n_true()}

The number of \texttt{True} values in the mask.

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

Returns \texttt{image} (\texttt{type(self)}) – A copy of this image, normalized.

\texttt{normalize_std(mode='all', **kwargs)}

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

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

Returns \texttt{image} (\texttt{type(self)}) – A copy of this image, normalized.

\texttt{pixels_range()} \\

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

Returns \texttt{min_max} (\texttt{(dtype, dtype)}) – The minimum and maximum value of the pixels array.

\texttt{pixels_with_channels_at_back(out_dtype=None)}

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.

If this image is single channel, the final axis is dropped.

Parameters \texttt{out_dtype} (\texttt{np.dtype}, optional) – The dtype the output array should be.

Returns \texttt{rolled_channels} (\texttt{ndarray}) – Pixels with channels as the back (last) axis. If single channel, the last axis will be dropped.

\texttt{proportion_false()} \\

The proportion of the mask which is \texttt{False}

Type \texttt{float} \\

\texttt{proportion_true()} \\

The proportion of the mask which is \texttt{True}.

Type \texttt{float} \\

\texttt{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

\begin{itemize}
  \item \texttt{n_levels} (\texttt{int}, optional) – Total number of levels in the pyramid, including the original unmodified image
  \item \texttt{downscale} (\texttt{float}, optional) – Downscale factor.
\end{itemize}

Yields \texttt{image_pyramid} (\texttt{generator}) – Generator yielding pyramid layers as \texttt{Image} objects.
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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear <em>(default)</em></td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear <em>(default)</em></td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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 be 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]
<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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]

:: raises ValueError:: If the number of dimensions of the new shape does not match the number of dimensions of the image.

**rolled_channels** *

Deprecated - please use the equivalent `pixels_with_channels_at_back` method.

**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 \(\text{retain\_shape}=False\), then the returned image has the correct size so that the whole area of the current image is included.

**Parameters**

- \(\text{theta} \ (\text{float})\) – The angle of rotation about the centre.
- \(\text{degrees} \ (\text{bool}, \text{optional})\) – If True, \(\text{theta}\) is interpreted in degrees. If False, \(\text{theta}\) is interpreted as radians.
- \(\text{retain\_shape} \ (\text{bool}, \text{optional})\) – 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.
- \(\text{cval} \ (\text{float}, \text{optional})\) – The value to be set outside the rotated image boundaries.
- \(\text{round} \ ((\text{'ceil'}, \text{'floor'}, \text{'round'}), \text{optional})\) – Rounding function to be applied to floating point shapes. This is only used in case \(\text{retain\_shape}=True\).
- \(\text{order} \ (\text{int}, \text{optional})\) – The order of interpolation. The order has to be in the range \([0, 5]\). This is only used in case \(\text{retain\_shape}=True\).

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (\text{(default)})</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- \(\text{return\_transform} \ (\text{bool}, \text{optional})\) – If True, then the \(\text{Transform}\) object that was used to perform the rotation is also returned.

**Returns**

- \(\text{rotated\_image} \ (\text{type(self)})\) – The rotated image.
- \(\text{transform} \ (\text{Transform})\) – The transform that was used. It only applies if \(\text{return\_transform}\) is True.

**Raises** \(\text{ValueError}\) – Image rotation is presently only supported on 2D images

\(\text{sample} \ (\text{points\_to\_sample}, \text{mode=\text{'constant'}}, \text{cval=\text{False}}, **\text{kwargs})\)

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**

- \(\text{points\_to\_sample} \ (\text{PointCloud})\) – Array of points to sample from the image. Should be \((n\_points, n\_dims)\)
- \(\text{mode} \ ((\text{constant}, \text{nearest}, \text{reflect}, \text{wrap}), \text{optional})\) – Points outside the boundaries of the input are filled according to the given mode.
- \(\text{cval} \ (\text{float}, \text{optional})\) – Used in conjunction with \text{mode constant}, the value outside the image boundaries.

**Returns** \(\text{sampled\_pixels} \ ((n\_points, n\_channels) \text{ bool ndarray})\) – 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 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)

Returns new_image (BooleanImage) – A new boolean image where the provided patch lo-
cations 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 \((1, 2)\)

### **transform** about centre

(transform, retain_shape=False, cval=0.0, round='round', order=1, return_transform=False)

Return a copy of this image, transformed about its centre.

Note that the *retain_shape* argument defines the shape of the transformed image. If *retain_shape=True*, then the shape of the transformed 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.

**Note:** This method will not work for transforms that result in a transform chain as *TransformChain* is not invertible.

**Note:** Be careful when defining transforms for warping images. All pixel locations must fall within a valid range as expected by the transform. Therefore, your transformation must accept ‘negative’ pixel locations as the pixel locations provided to your transform will have the object centre subtracted from them.

**Parameters**

• **transform** *(ComposableTransform and VInvertible type)* – A composable transform. *pseudoinverse* will be invoked on the resulting transform so it must implement a valid inverse.

• **retain_shape** *(bool, optional)* – If True, then the shape of the sheared 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 sheared 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*.

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear*(default)*</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>
• **return_transform** (bool, optional) – If True, then the Transform object that was used to perform the shearing is also returned.

**Returns**

• **transformed_image** (type(self)) – The transformed image.

• **transform** (Transform) – The transform that was used. It only applies if return_transform is True.

**Examples**

This is an example for rotating an image about its center. Let’s first load an image, create the rotation transform and then apply it

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Rotation

# Load image
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
rot_tr = Rotation.init_from_2d_ccw_angle(45)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')

# Render rotated image
plt.subplot(132)
im.transform_about_centre(rot_tr).view_landmarks()
plt.title('Rotated')

# Render rotated image that has shape equal as original image
plt.subplot(133)
im.transform_about_centre(rot_tr, retain_shape=True).view_landmarks()
plt.title('Rotated (Retain original shape)')
```

Similarly, in order to apply a shear transform

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Affine

# Load image
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
shear_tr = Affine.init_from_2d_shear(25, 10)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')
```

(continues on next page)
true_indices()

The indices of pixels that are True.

Type (n_dims, n_true) ndarray

view_widget(figure_size=(7, 7))

Visualizes the image using an interactive widget.

Parameters

- **figure_size** ((int, int), optional) – The initial size of the rendered figure.

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 zoomed 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.
property has_landmarks
Whether the object has landmarks.

    Type bool

property 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

property landmarks
The landmarks object.

    Type LandmarkManager

property 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

property n_channels
The number of channels on each pixel in the image.

    Type int

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

    Type int

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

    Type int

property n_landmark_groups
The number of landmark groups on this object.

    Type int

property n_parameters
The length of the vector that this object produces.

    Type int

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

    Type int

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

    Type tuple

property 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**

```python
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.

**Raises** `ValueError` – Mask is not the same shape as the image

```python
 Velvet 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=(7, 7))
```

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

  ```python
  {none, nearest, bilinear, bicubic, spline16, spline36, hanning, hamming, hermite, kaiser, quadratic, 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
Menpo Documentation, Release 0.9.0

- **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*

```python
_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_color=None, line_style='-', line_width=1, render_markers=True, marker_style='o', marker_size=5, marker_face_color=None, marker_edge_color=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_color='k', render_legend=False, legend_title='', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', 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=(7, 7))
```

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

```plaintext
{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:

```plaintext
{r, g, b, c, m, k, w}
```

• **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

```plaintext
{.,,,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

```plaintext
{r, g, b, c, m, k, w}
```

• **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:

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'best'</td>
<td>0</td>
</tr>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>

• **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

<table>
<thead>
<tr>
<th>Font Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>serif</td>
</tr>
<tr>
<td>sans-serif</td>
</tr>
<tr>
<td>cursive</td>
</tr>
<tr>
<td>fantasy</td>
</tr>
<tr>
<td>monospace</td>
</tr>
</tbody>
</table>

• **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

<table>
<thead>
<tr>
<th>Font Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>ultralight</td>
</tr>
<tr>
<td>light</td>
</tr>
<tr>
<td>normal</td>
</tr>
<tr>
<td>regular</td>
</tr>
<tr>
<td>book</td>
</tr>
<tr>
<td>medium</td>
</tr>
<tr>
<td>roman</td>
</tr>
<tr>
<td>semibold</td>
</tr>
<tr>
<td>demibold</td>
</tr>
<tr>
<td>demi</td>
</tr>
<tr>
<td>bold</td>
</tr>
<tr>
<td>heavy</td>
</tr>
<tr>
<td>extra bold</td>
</tr>
<tr>
<td>black</td>
</tr>
</tbody>
</table>

• **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=<class '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) –

<table>
<thead>
<tr>
<th>mode</th>
<th>Greyscale Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Equal average of all channels</td>
</tr>
<tr>
<td>luminosity</td>
<td>Calculates the luminance using the CCIR 601 formula:</td>
</tr>
<tr>
<td></td>
<td>[Y' = 0.2989R' + 0.5870G' + 0.1140B']</td>
</tr>
<tr>
<td>channel</td>
<td>A specific channel is chosen as the intensity value.</td>
</tr>
</tbody>
</table>

• 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** *(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:

```python
global hist, bin_edges = image.as_histogram()
for k in range(len(hist)):
    plt.subplot(1, len(hist), k + 1)
    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=\<class '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.

**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*

**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*

**clip_pixels** *(minimum=None, maximum=None)*

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

**Parameters**

• **minimum** (*float*, optional) – The minimal value of the clipped pixels. If *None* is provided, the default value will be 0.

• **maximum** (*float*, optional) – The maximal value of the clipped pixels. If *None* is provided, the default value will depend on the dtype.
Returns `rescaled_image` (type(self)) – A copy of this image with pixels linearly rescaled to fit in the range provided.

`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:

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

Returns constrained (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.

Returns constrained (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.

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

**Returns**
- **bounded_points** *(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** *(ndarray)* – The minimum index over each dimension.
- **max_indices** *(ndarray)* – The maximum index over each dimension.
- **constrain_to_boundary** *(bool, optional)* – If True the crop will be snapped to not go beyond this image's 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 :class:`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 :exc:`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 :exc:`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 :class:`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 :exc:`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 `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 a border proportional to the pointcloud spread or range.

**Parameters**

• **pointcloud** *(PointCloud)* – The pointcloud to crop around.

• **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.

• **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 `constrain_to_boundary=False`, and an attempt is made to crop the image in a way that violates the image bounds.

```python
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.
```

```python
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.
```

```python
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.
```

```python
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
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 \((n_{\text{center}}, n_{\text{offset}}, n_{\text{channels}}, \text{patch\_shape})\) \(ndarray\), thus a single numpy array is returned containing each patch. If False, a list of \(n_{\text{center}} \times n_{\text{offset}}\) \(Image\) objects is returned representing each patch.

Returns patches (list or \(ndarray\)) – Returns the extracted patches. Returns a list if as_single_array=True and a \(ndarray\) 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** (\((n_{\text{offsets}}, n_{\text{dims}})\) \(ndarray\) 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 \((n_{\text{center}}, n_{\text{offset}}, n_{\text{channels}}, \text{patch\_shape})\) \(ndarray\), thus a single numpy array is returned containing each patch. If False, a list of \(n_{\text{center}} \times n_{\text{offset}}\) \(Image\) objects is returned representing each patch.

Returns patches (list or \(ndarray\)) – Returns the extracted patches. Returns a list if as_single_array=True and a \(ndarray\) 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 \(n_{\text{channels}}\) 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** (\((n_{\text{pixels}},)\)) – 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.

Returns image (\(MaskedImage\)) – 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()

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 true pixels in this image.

Type (n_dims, n_true_pixels) ndarray

classmethod init_blank(shape, n_channels=1, fill=0, dtype=<class '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_channels_at_back**( `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.

**classmethod init_from_pointcloud**( `pointcloud`, `group=None`, `boundary=0`, `constrain_mask=True`, `n_channels=1`, `fill=0`, `dtype=<class ‘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` )

Deprecated - please use the equivalent `init_from_channels_at_back` method.

**masked_pixels()**

Get the pixels covered by the `True` values in the mask.

**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.

**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.

**pixels_with_channels_at_back**(out_dtype=None)
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.

If this image is single channel, the final axis is dropped.

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

**Returns** rolled_channels**(ndarray)** – Pixels with channels as the back (last) axis. If single channel, the last axis will be dropped.

**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.

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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear <em>(default)</em></td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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 be 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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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.

:raises ValueError: If the number of dimensions of the new shape does not match the number of dimensions of the image.

**rolled_channels**()

Deprecated - please use the equivalent pixels_with_channels_at_back method.

**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.
Menpo Documentation, Release 0.9.0

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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.
Returns **new_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*)
Depreciated - 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. \((n\text{\_center}, n\text{\_offset}, self.n\text{\_channels}, patch\_shape)\) ndarray
2. \(\text{list of } n\text{\_center} \times n\text{\_offset} \text{ Image objects}\)

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 \((n\text{\_center}, n\text{\_offset}, self.n\text{\_channels}, patch\_shape)\) ndarray or a \(\text{list of } n\text{\_center} \times n\text{\_offset} \text{ Image objects}\).
- **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. \((n\text{\_center}, n\text{\_offset}, self.n\text{\_channels}, patch\_shape)\) ndarray
2. \(\text{list of } n\text{\_center} \times n\text{\_offset} \text{ Image objects}\)

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 \((n\text{\_center}, n\text{\_offset}, self.n\text{\_channels}, patch\_shape)\) ndarray or a \(\text{list of } n\text{\_center} \times n\text{\_offset} \text{ Image objects}\).
- **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)

### transform_about_centre(transform, retain_shape=False, cval=0.0, round='round', order=1, return_transform=False)

Return a copy of this image, transformed about its centre.

Note that the retain_shape argument defines the shape of the transformed image. If retain_shape=True, then the shape of the transformed 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.

**Note:** This method will not work for transforms that result in a transform chain as TransformChain is not invertible.

**Note:** Be careful when defining transforms for warping images. All pixel locations must fall within a valid range as expected by the transform. Therefore, your transformation must accept ‘negative’ pixel locations as the pixel locations provided to your transform will have the object centre subtracted from them.

**Parameters**

- **transform**(ComposableTransform and VInvertible type) – A composable transform. pseudoinverse will be invoked on the resulting transform so it must implement a valid inverse.
- **retain_shape**(bool, optional) – If True, then the shape of the sheared 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 sheared 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.
<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (<strong>default</strong>)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

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

**Returns**

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

**Examples**

This is an example for rotating an image about its center. Let’s first load an image, create the rotation transform and then apply it.

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Rotation

# Load image
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
rot_tr = Rotation.init_from_2d_ccw_angle(45)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')

# Render rotated image
plt.subplot(132)
im.transform_about_centre(rot_tr).view_landmarks()
plt.title('Rotated')

# Render rotated image that has shape equal as original image
plt.subplot(133)
im.transform_about_centre(rot_tr, retain_shape=True).view_landmarks()
plt.title('Rotated (Retain original shape)')
```

Similarly, in order to apply a shear transform:

```python
import matplotlib.pyplot as plt
import menpo.io as mio
from menpo.transform import Affine

# Load image
```
```
im = mio.import_builtin_asset.lenna_png()

# Create shearing transform
shear_tr = Affine.init_from_2d_shear(25, 10)

# Render original image
plt.subplot(131)
im.view_landmarks()
plt.title('Original')

# Render sheared image
plt.subplot(132)
im.transform_about_centre(shear_tr).view_landmarks()
plt.title('Sheared')

# Render sheared image that has shape equal as original image
plt.subplot(133)
im.transform_about_centre(shear_tr, retain_shape=True).view_landmarks()
plt.title('Sheared (Retain original shape)')
```

**view_widget** *(figure_size=(7, 7))*
Visualizes the image using an interactive widget.

**Parameters**

- **figure_size** *(int, int, optional)* – The initial size of the rendered figure.

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

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear (default)</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

- **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

(warped_image, 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]

<table>
<thead>
<tr>
<th>Order</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest-neighbor</td>
</tr>
<tr>
<td>1</td>
<td>Bi-linear</td>
</tr>
<tr>
<td>2</td>
<td>Bi-quadratic</td>
</tr>
<tr>
<td>3</td>
<td>Bi-cubic</td>
</tr>
<tr>
<td>4</td>
<td>Bi-quartic</td>
</tr>
<tr>
<td>5</td>
<td>Bi-quintic</td>
</tr>
</tbody>
</table>

• **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 zoomed 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.

**property has_landmarks**

Whether the object has landmarks.

*Type* `bool`

**property 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`

**property landmarks**

The landmarks object.

*Type* `LandmarkManager`

**property n_channels**

The number of channels on each pixel in the image.

*Type* `int`

**property n_dims**

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

*Type* `int`

**property n_elements**

Total number of data points in the image \((\text{prod(shape)}, \text{n_channels})\)

*Type* `int`

**property n_landmark_groups**

The number of landmark groups on this object.

*Type* `int`

**property n_parameters**

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

**property n_pixels**
Total number of pixels in the image \((\prod(\text{shape}),)\)

Type `int`

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

Type `tuple`

**property 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`

### 1.3.2 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.
- `snapped_max` – The per-dimension maximum index that could be used if the crop was constrained to the image boundaries.

```python
with_traceback ()
```

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

**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.
Menpo Documentation, Release 0.9.0

```python
with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```

1.4 menpo.feature

1.4.1 Features

no_op

```python
menpo.feature.no_op(pixels)
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

```python
menpo.feature.gradient(pixels)
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.
```

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. If the image is 2-dimensional the pixels should be of type float/double (int is not supported).

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

```python
menpo.feature.gaussian_filter(pixels, sigma)
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(pixels, double_angles=False, verbose=False)

Extracts Image Gradient Orientation (IGO) features from the input image. The output image has \(N \times C\) 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 \(\cos(\phi), \sin(\phi), \cos(2 \times \phi), \sin(2 \times \phi)\), 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(pixels, verbose=False)

Extracts Edge Structure (ES) features from the input image. The output image has \(N \times C\) 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
**Functions**

- `menpo.feature.lbp` (pixels, radius=None, samples=None, mapping_type='riu2', window_step_vertical=1, window_step_horizontal=1, window_step_unit='pixels', padding=True, verbose=False, skip_checks=False)

  Extracts Local Binary Pattern (LBP) features from the input image. The output image has $N \times C$ 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 $C = \text{len}(\text{radius}) \times \text{len}(\text{samples})$.

**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

```python
menpo.feature.hog(pixels, mode='dense', algorithm='dalaltriggs', num_bins=9, cell_size=8, block_size=2, signed_gradient=True, l2_norm_clip=0.2, window_height=1, window_width=1, window_unit='blocks', window_step_vertical=1, window_step_horizontal=1, window_step_unit='pixels', padding=True, verbose=False)
```

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_vertical = window_step_horizontal = cell_size`.

  The sparse case of `zhuramanan` algorithm sets `window_height = window_width = 3 * cell_size` and `window_step_horizontal = window_step_unit = 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.
Menpo Documentation, Release 0.9.0

- **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 $K = \text{num\_bins} \times \text{block\_size} \times \text{block\_size}$ 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 *(pixels, step=1, radius=15, rings=2, histograms=2, orientations=8, normalization='l1', sigmas=None, ring_radii=None, verbose=False)*

Extracts Daisy features from the input image. The output image has $N \times C$ 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, $C = (rings \times histograms + 1) \times orientations$.

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

\[
\text{len(ring_radii)} = \text{len(sigmas)} + 1
\]

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 $C = (rings \times histograms + 1) \times orientations$.

Raises

- **ValueError** – len(sigmas)-1 != len(ring_radii)

- **ValueError** – Invalid normalization method.

References

1.4.2 Optional Features

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.

### \texttt{dsift}

```python
menpo.feature.dsift(pixels, window_step_horizontal=1, window_step_vertical=1, num_bins_horizontal=2, num_bins_vertical=2, num_or_bins=9, cell_size_horizontal=6, cell_size_vertical=6, fast=True, verbose=False)
```

Computes a 2-dimensional dense SIFT features image with \( C \) number of channels, where \( C = \text{num_bins_horizontal} \times \text{num_bins_vertical} \times \text{num_or_bins} \). 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.

---


fast_dsift

`menpo.feature.fast_dsift(pixels,  window_step_horizontal=1,  window_step_vertical=1,  *,
num_bins_horizontal=1,  num_bins_vertical=1,  num_or_bins=8,
cell_size_horizontal=5,  cell_size_vertical=5,  fast=True,  verbose=False)`

Computes a 2-dimensional dense SIFT features image with $C$ number of channels, where $C = num\_bins\_horizontal \times num\_bins\_vertical \times num\_or\_bins$. 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**


**vector_128_dsift**

`menpo.feature.vector_128_dsift(x, dtype=<class '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=<class '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 normalized 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**

\(^1\) Arandjelovic, Relja, and Andrew Zisserman. “Three things everyone should know to improve object retrieval.”, CVPR, 2012.

---

### 1.4.3 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(pixels, *, double_angles=True, verbose=False)`

Extracts Image Gradient Orientation (IGO) features from the input image. The output image has \(N \times C\) 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**

---

\(^1\) Arandjelovic, Relja, and Andrew Zisserman. “Three things everyone should know to improve object retrieval.”, CVPR, 2012.
• **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**

**sparse_hog**

```python
menpo.feature.sparse_hog(pixels, *, mode='sparse', algorithm='dalaltriggs', num_bins=9, cell_size=8, block_size=2, signed_gradient=True, l2_norm_clip=0.2, window_height=1, window_width=1, window_unit='blocks', window_step_vertical=1, window_step_horizontal=1, window_step_unit='pixels', padding=True, verbose=False)
```

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 \( K = \text{num\_bins} \times \text{block\_size} \times \text{block\_size} \) 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

### 1.4.4 Normalization

The following functions perform some kind of normalization on an image.

#### normalize

```python
menpo.feature.normalize(img, scale_func=None, mode='all', error_on_divide_by_zero=True)
```

Normalize the pixel values via mean centering and an optional scaling. By default the scaling will be 1.0. The `mode` parameter selects whether the normalisation is computed across all pixels in the image or per-channel.

- **Parameters**
  - `img` *(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.
  - `scale_func` *(callable, optional)* – Compute the scaling factor. Expects a single parameter and an optional `axis` keyword argument and will be passed the entire pixel array. Should return a 1D numpy array of one or more values.
  - `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.
  - `error_on_divide_by_zero` *(bool, optional)* – If `True`, will raise a `ValueError` on dividing by zero. If `False`, will merely raise a warning and only those values with non-zero denominators will be normalized.

- **Returns** `pixels` *(Image or subclass or (X, Y, ..., Z, C) ndarray)* – A normalized copy of the image that was passed in.

- **Raises** `ValueError` – If any of the denominators are 0 and `error_on_divide_by_zero` is `True`.

#### normalize_norm

```python
menpo.feature.normalize_norm(pixels, mode='all', error_on_divide_by_zero=True)
```

Normalize the pixels to be mean centred and have unit norm. The `mode` parameter selects whether the normalisation is computed across all pixels in the image or per-channel.

- **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` *(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.
• **error_on_divide_by_zero** *(bool, optional)* – If True, will raise a `ValueError` on dividing by zero. If False, will merely raise a warning and only those values with non-zero denominators will be normalized.

**Returns pixels** *(Image or subclass or (X, Y, ..., Z, C) ndarray)* – A normalized copy of the image that was passed in.

**Raises ValueError** – If any of the denominators are 0 and `error_on_divide_by_zero` is True.

### normalize_std

`menpo.feature.normalize_std(pixels, mode='all', error_on_divide_by_zero=True)`

Normalize the pixels to be mean centred and have unit standard deviation. The `mode` parameter selects whether the normalisation is computed across all pixels in the image or per-channel.

**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** *(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.

• **error_on_divide_by_zero** *(bool, optional)* – If True, will raise a `ValueError` on dividing by zero. If False, will merely raise a warning and only those values with non-zero denominators will be normalized.

**Returns pixels** *(Image or subclass or (X, Y, ..., Z, C) ndarray)* – A normalized copy of the image that was passed in.

**Raises ValueError** – If any of the denominators are 0 and `error_on_divide_by_zero` is True.

### normalize_var

`menpo.feature.normalize_var(pixels, mode='all', error_on_divide_by_zero=True)`

Normalize the pixels to be mean centred and normalize according to the variance. The `mode` parameter selects whether the normalisation is computed across all pixels in the image or per-channel.

**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** *(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.

• **error_on_divide_by_zero** *(bool, optional)* – If True, will raise a `ValueError` on dividing by zero. If False, will merely raise a warning and only those values with non-zero denominators will be normalized.

**Returns pixels** *(Image or subclass or (X, Y, ..., Z, C) ndarray)* – A normalized copy of the image that was passed in.
Raises `ValueError` – If any of the denominators are 0 and `error_on_divide_by_zero` is True.

### 1.4.5 Visualization

**glyph**

**sum_channels**

```python
menpo.feature.sum_channels(pixels, channels=None)
```

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.

### 1.4.6 Widget

**features_selection_widget**

```python
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:

```python
features_image = features_function[0](image)
```

**Examples**

The widget can be invoked as

```python
from menpo.feature import features_selection_widget
features_fun = features_selection_widget()
```

And the returned function can be used as

```python
import menpo.io as mio
image = mio.import_builtin_asset.lenna_png()
features_image = features_fun[0](image)
```
1.4.7 References

1.5 menpo.landmark

1.5.1 Abstract Classes

**Landmarkable**

```python
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` or `LabelledPointUndirectedGraph`.

    `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`
```

1.5.2 Exceptions

**LabellingError**

```python
class menpo.landmark.LabellingError
    Bases: Exception

    Raised when labelling a landmark manager and the set of landmarks does not match the expected semantic
layout.

    `with_traceback()`
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```
1.5.3 Landmarks & Labeller

**LandmarkManager**

```
class menpo.landmark.LandmarkManager
    Bases: MutableMapping, Transformable

Store for PointCloud or LabelledPointUndirectedGraph 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. The LandmarkManager will contain instances of PointCloud or LabelledPointUndirectedGraph or subclasses thereof. LabelledPointUndirectedGraph is unique in its ability to include labels that refer to subsets of the underlying points that represent interesting semantic labels. These PointCloud or LabelledPointUndirectedGraph (or subclasses) are stored under string keys - these keys are referred to as the group name. A special case is where there is a single unambiguous group attached to a LandmarkManager - in this case None can be used as a key to access this sole group.

Note that all groups 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.
```

```get(k[, d]) → D[k] if k in D, else d. d defaults to None.
```

```items() → a set-like object providing a view on D’s items
```

```items_matching(glob_pattern) Yield only items (group, PointCloud) where the key matches a given glob.
```

```Parameters glob_pattern (str) – A glob pattern e.g. ‘frontal_face_*’
```

```Yields item ((group, PointCloud)) – Tuple of (str, PointCloud) where the group matches the glob.
```

```keys() → a set-like object providing a view on 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() → an object providing a view on D’s values
```
```python
property group_labels
    All the labels for the landmark set sorted by insertion order.
    Type  list of str

property has_landmarks
    Whether the object has landmarks or not
    Type  int

property n_dims
    The total number of dimensions.
    Type  int

property n_groups
    Total number of labels.
    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 LabelledPointUndirectedGraph or 
  PointCloud

•  group (str) – The group label of the existing pointcloud 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, LabelledPointUndirectedGraph)) – A labelling 
  function taken from this module. Takes as input a PointCloud or 
  LabelledPointUndirectedGraph or subclass and returns a tuple of (new 
  group label, new LabelledPointUndirectedGraph with semantic labels applied).

Returns landmarkable (Landmarkable) – Augmented landmarkable (this is just for conve-
  nience, the object will actually be modified in place)

1.5.4 Bounding Box Labels

bounding_box_mirrored_to_bounding_box

menpo.landmark.bounding_box_mirrored_to_bounding_box(bbox)
    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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input 
  labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 of str -> int ndarray, optional) – Only returned if return_mapping==True. Used for building LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/pointcloud contains less than the expected number of points.

### bounding_box_to_bounding_box

**menpo.landmark.bounding_box_to_bounding_box**(bbox)

Apply a single ‘all’ label to a given bounding box. This bounding box must be as specified by the bounding_box method.

**Parameters**

• **x** (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 of str -> int ndarray, optional) – Only returned if return_mapping==True. Used for building LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/pointcloud contains less than the expected number of points.
1.5.5 Labels

Pre-defined landmark labels that normally correspond to standard database mark-ups.

Human Face

`face_ibug_68_to_face_ibug_49`

```python
menpo.landmark.face_ibug_68_to_face_ibug_49(pcloud)
```

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` (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- `x_labelled` (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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` (*ordreddict*) (`str` -> *int* or *ndarray*), optional) – Only returned if `return_mapping==True`. Used for building *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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(pcloud)`

Apply the IBUG 49-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

References

Parameters

- `x (LabelledPointUndirectedGraph or PointCloud or ndarray)` – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- `x_labelled (LabelledPointUndirectedGraph or PointCloud)` – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)`

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

1.5. menpo.landmark
Parameters

- **x** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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** (*orderteddict* {str -> int, ndarray}, optional) – Only returned if return_mapping==True. Used for building *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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(pcloud)`

Apply the IBUG 51-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- **tri**

References
• **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 `LabelledPointUndirectedGraph`). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)* – If a `LabelledPointUndirectedGraph` was passed, a `LabelledPointUndirectedGraph` is returned. This labelled pointgraph 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** *(ordredict {str -> int ndarray}, optional)* – Only returned if `return_mapping==True`. Used for building `LabelledPointUndirectedGraph`.

:raises: `LabellingError`: If the given labelled point graph/pointcloud contains less than the expected number of points.

**face_ibug_68_to_face_ibug_65**

```python
menpo.landmark.face_ibug_68_to_face_ibug_65(pcloud)
```

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** *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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
LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)*  
  - If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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** *(ord dict {str -> int ndarray}, optional)*  
  - Only returned if return_mapping==True. Used for building LabelledPointUndirectedGraph.

:raises : LabellingError: If the given labelled point graph/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 (pcloud)

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** *(LabelledPointUndirectedGraph or PointCloud or ndarray)*  
  - The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns
**x_labelled** *(LabelledPointUndirectedGraph or PointCloud)*
- If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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 *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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*(pcloud)

Apply the IBUG 66-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:
- tri

---

**References**

---

**Parameters**

- **x** *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)* – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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 *LabelledPointUndirectedGraph*. 

---

1.5. *menpo.landmark* 121
:raises: LabellingError: If the given labelled point graph/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(pcloud)

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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- x_labelled (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)*

Apply the IBUG 68-point semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

**Parameters**

- **x** *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)* – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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 *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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(pcloud)*

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** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• **x_labelled** (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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 *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/pointcloud contains less than the expected number of points.

```python
menpo.landmark.face_ibug_49_to_face_ibug_49(pcloud)
```

Apply the IBUG 49-point semantic labels.

The semantic labels applied are as follows:

• left_eyebrow
• right_eyebrow
• nose
• left_eye
• right_eye
• mouth
Parameters

- **x** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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** (*ordredict {str -> int ndarray}*, optional) – Only returned if return_mapping==True. Used for building *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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*(pcloud)

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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• x_labelled (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 (orddict {str -> int ndarray}, optional) – Only returned if return_mapping==True. Used for building LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)

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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those or array to label. If a pointcloud is
passed, then only the connectivity information is propagated to the pointcloud (a subclass of \texttt{PointCloud} may be returned).

- **\texttt{return_mapping} (\texttt{bool}, optional)** – Only applicable if a \texttt{PointCloud} or \texttt{ndarray} is passed. Returns the mapping dictionary which maps labels to indices into the resulting \texttt{PointCloud} (which is then used to for building a \texttt{LabelledPointUndirectedGraph}. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- **\texttt{x_labelled} (\texttt{LabelledPointUndirectedGraph} or \texttt{PointCloud})** – If a \texttt{LabelledPointUndirectedGraph} was passed, a \texttt{LabelledPointUndirectedGraph} is returned. This labelled pointgraph will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a \texttt{PointCloud} was passed, a \texttt{PointCloud} is returned. Only the connectivity information is propagated to the pointcloud (a subclass of \texttt{PointCloud} may be returned).

- **\texttt{mapping_dict} (\texttt{ordeddict \{str -> int ndarray\}}, optional)** – Only returned if \texttt{return_mapping==True}. Used for building \texttt{LabelledPointUndirectedGraph}.

:raises: \texttt{LabellingError}: If the given labelled point graph/pointcloud contains less than the expected number of points.

---

### \texttt{face\_bu3dfe\_83\_to\_face\_bu3dfe\_83}

\texttt{menpo.landmark.f\_ace\_bu3dfe\_83\_to\_face\_bu3dfe\_83 (pcloud)}

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
• \textbf{x} (\texttt{LabelledPointUndirectedGraph} or \texttt{PointCloud} or \texttt{ndarray}) – The input labelled point graph, pointcloud, subclass of those or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of \texttt{PointCloud} may be returned).

• \textbf{return\_mapping} (\texttt{bool}, optional) – Only applicable if a \texttt{PointCloud} or \texttt{ndarray} is passed. Returns the mapping dictionary which maps labels to indices into the resulting \texttt{PointCloud} (which is then used to for building a \texttt{LabelledPointUndirectedGraph}). This parameter is only provided for internal use so that other labellers can piggyback off one another.

\textbf{Returns}

• \textbf{x\_labelled} (\texttt{LabelledPointUndirectedGraph} or \texttt{PointCloud}) – If a \texttt{LabelledPointUndirectedGraph} was passed, a \texttt{LabelledPointUndirectedGraph} is returned. This labelled point graph will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a \texttt{PointCloud} was passed, a \texttt{PointCloud} is returned. Only the connectivity information is propagated to the pointcloud (a subclass of \texttt{PointCloud} may be returned).

• \textbf{mapping\_dict} (\texttt{orddict} \{\texttt{str} \rightarrow \texttt{int} \texttt{ndarray}\}, optional) – Only returned if \texttt{return\_mapping==True}. Used for building \texttt{LabelledPointUndirectedGraph}.

:raises: \texttt{LabellingError}: If the given labelled point graph/pointcloud contains less than the expected number of points.

\textbf{Human Eyes}

\texttt{eye\_ibug\_close\_17\_to\_eye\_ibug\_close\_17}

\texttt{menpo.landmark.\texttt{eye\_ibug\_close\_17\_to\_eye\_ibug\_close\_17}(pcloud)}

Apply the IBUG 17-point close eye semantic labels.

The semantic labels applied are as follows:

• upper\_eyelid

• lower\_eyelid

\textbf{Parameters}

• \textbf{x} (\texttt{LabelledPointUndirectedGraph} or \texttt{PointCloud} or \texttt{ndarray}) – The input labelled point graph, pointcloud, subclass of those or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of \texttt{PointCloud} may be returned).

• \textbf{return\_mapping} (\texttt{bool}, optional) – Only applicable if a \texttt{PointCloud} or \texttt{ndarray} is passed. Returns the mapping dictionary which maps labels to indices into the resulting \texttt{PointCloud} (which is then used to for building a \texttt{LabelledPointUndirectedGraph}). This parameter is only provided for internal use so that other labellers can piggyback off one another.

\textbf{Returns}

• \textbf{x\_labelled} (\texttt{LabelledPointUndirectedGraph} or \texttt{PointCloud}) – If a \texttt{LabelledPointUndirectedGraph} was passed, a
LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: **LabellingError**: If the given labelled point graph/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 (pcloud)

Apply the IBUG 17-point close eye semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

• tri

**Parameters**

• **x** *(LabelledPointUndirectedGraph or PointCloud or ndarray) –* The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** *(LabelledPointUndirectedGraph or PointCloud) –* If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: **LabellingError**: If the given labelled point graph/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(pcloud)
```

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` *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- `x_labelled` *(LabelledPointUndirectedGraph or PointCloud)* – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)
```

Apply the IBUG 38-point open eye semantic labels, with trimesh connectivity.

The semantic labels applied are as follows:

- tri

**Parameters**
• **x** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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** (*orderrdict* {*str* : *int* *ndarray*}, optional) – Only returned if *return_mapping==True*. Used for building *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/pointcloud contains less than the expected number of points.

**Human Hand**

**hand_ibug_39_to_hand_ibug_39**

*menpo.landmark.hand_ibug_39_to_hand_ibug_39* (*pcloud*)

Apply the IBUG 39-point semantic labels.

The semantic labels applied are as follows:

• thumb
• index
• middle
• ring
• pinky
• palm

**Parameters**

• **x** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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
LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)*
  - If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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** *(orderrdict {str -> int ndarray}, optional)*
  - Only returned if return_mapping==True. Used for building LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/pointcloud contains less than the expected number of points.

Human Body Pose

**pose_flic_11_to_pose_flic_11**

menpo.landmark.pose_flic_11_to_pose_flic_11(pcloud)

Apply the flic 11-point semantic labels.

The semantic labels applied are as follows:

- left_arm
- right_arm
- hips
- face

References

Parameters

- **x** *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)*
  - If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 infor-
mation 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
  LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)

Apply the human3.6M 17-point semantic labels (based on the original semantic labels of Human3.6 but remov-
ing 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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input
  labelled point graph, pointcloud, subclass of those 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 in-
dices into the resulting PointCloud (which is then used to for building a
  LabelledPointUndirectedGraph. This parameter is only provided for internal use
  so that other labellers can piggyback off one another.

**Returns**

- x_labelled (LabelledPointUndirectedGraph or PointCloud) – If a
  LabelledPointUndirectedGraph was passed, a
  LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 `LabelledPointUndirectedGraph`.

:raises: `LabellingError`: If the given labelled point graph/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(pcloud)`

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** (`LabelledPointUndirectedGraph` or `PointCloud` or `ndarray`) – The input labelled point graph, pointcloud, subclass of those 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 `LabelledPointUndirectedGraph`). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- **x_labelled** (`LabelledPointUndirectedGraph` or `PointCloud`) – If a `LabelledPointUndirectedGraph` was passed, a `LabelledPointUndirectedGraph` is returned. This labelled pointgraph 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 `LabelledPointUndirectedGraph`.

:raises: `LabellingError`: If the given labelled point graph/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**(pcloud)

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** (`LabelledPointUndirectedGraph` or `PointCloud` or `ndarray`) – The input labelled point graph, pointcloud, subclass of those 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 `LabelledPointUndirectedGraph`). This parameter is only provided for internal use so that other labellers can piggyback off one another.

### Returns

- **x_labelled** (`LabelledPointUndirectedGraph` or `PointCloud`) – If a `LabelledPointUndirectedGraph` was passed, a `LabelledPointUndirectedGraph` is returned. This labelled pointgraph 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 `LabelledPointUndirectedGraph`. 

1.5. menpo.landmark
:raises: `LabellingError`: If the given labelled point graph/pointcloud contains less than the expected number of points.

**pose_stickmen_12_to_pose_stickmen_12**

```python
def pose_stickmen_12_to_pose_stickmen_12(pcloud):
    # 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` *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 `LabelledPointUndirectedGraph`). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- `x_labelled` *(LabelledPointUndirectedGraph or PointCloud)* – If a `LabelledPointUndirectedGraph` was passed, a `LabelledPointUndirectedGraph` is returned. This labelled pointgraph 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 a `LabelledPointUndirectedGraph`.

:raises: `LabellingError`: If the given labelled point graph/pointcloud contains less than the expected number of points.
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

Parameters

- `x` *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- `x_labelled` *(LabelledPointUndirectedGraph or PointCloud)* – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/pointcloud contains less than the expected number of points.

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

• \( x \) (\textit{LabelledPointUndirectedGraph} or \textit{PointCloud} or \textit{ndarray}) – The input labelled point graph, pointcloud, subclass of those or array to label. If a pointcloud is passed, then only the connectivity information is propagated to the pointcloud (a subclass of \textit{PointCloud} may be returned).

• \texttt{return_mapping} (\texttt{bool}, optional) – Only applicable if a \textit{PointCloud} or \textit{ndarray} is passed. Returns the mapping dictionary which maps labels to indices into the resulting \textit{PointCloud} (which is then used to for building a \textit{LabelledPointUndirectedGraph}. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• \texttt{x_labelled} (\textit{LabelledPointUndirectedGraph} or \textit{PointCloud}) – If a \textit{LabelledPointUndirectedGraph} was passed, a \textit{LabelledPointUndirectedGraph} is returned. This labelled pointgraph will contain specific labels and these labels may refer to sub-pointclouds with specific connectivity information.

If a \textit{PointCloud} was passed, a \textit{PointCloud} is returned. Only the connectivity information is propagated to the pointcloud (a subclass of \textit{PointCloud} may be returned).

• \texttt{mapping_dict} (\texttt{orderrdict \{str -> int \ ndarray\}}, optional) – Only returned if \texttt{return_mapping==True}. Used for building \textit{LabelledPointUndirectedGraph}.

:raises: \texttt{LabellingError}: If the given labelled point graph/pointcloud contains less than the expected number of points.

car_streetscene_20_to_car_streetscene_view_2_10

\texttt{menpo.landmark.car_streetscene_20_to_car_streetscene_view_2_10(pcloud)}

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
• **x** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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 *LabelledPointUndirectedGraph*). This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• **x_labelled** (*LabelledPointUndirectedGraph* or *PointCloud*) – If a *LabelledPointUndirectedGraph* was passed, a *LabelledPointUndirectedGraph* is returned. This labelled pointgraph 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 *LabelledPointUndirectedGraph*.

:raises: *LabellingError*: If the given labelled point graph/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** (*pcloud*)

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** (*LabelledPointUndirectedGraph* or *PointCloud* or *ndarray*) – The input labelled point graph, pointcloud, subclass of those 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
LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled**: (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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**: (ordeddict {str -> int ndarray}, optional) – Only returned if return_mapping=True. Used for building LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)

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**: (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

- **x_labelled**: (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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(pcloud)

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** *(LabelledPointUndirectedGraph or PointCloud or ndarray)* – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

- **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)* – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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 (pcloud)
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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• x_labelled (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/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 (pcloud)
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 (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 LabelledPointUndirectedGraph. This parameter is only provided for internal use so that other labellers can piggyback off one another.

Returns

• x_labelled (LabelledPointUndirectedGraph or PointCloud) – If a LabelledPointUndirectedGraph was passed, a LabelledPointUndirectedGraph is returned. This labelled pointgraph 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 LabelledPointUndirectedGraph.

:raises: LabellingError: If the given labelled point graph/pointcloud contains less than the expected number of points.

Human Tongue

tongue_ibug_19_to_tongue_ibug_19

menpo.landmark.tongue_ibug_19_to_tongue_ibug_19(pcloud)

Apply the IBUG 19-point tongue semantic labels.

The semantic labels applied are as follows:

• outline
• bisector

Parameters

• x (LabelledPointUndirectedGraph or PointCloud or ndarray) – The input labelled point graph, pointcloud, subclass of those 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 `LabelledPointUndirectedGraph`). This parameter is only provided for internal use so that other labellers can piggyback off one another.

**Returns**

• **x_labelled** *(LabelledPointUndirectedGraph or PointCloud)* – If a `LabelledPointUndirectedGraph` was passed, a `LabelledPointUndirectedGraph` is returned. This labelled pointgraph 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 `LabelledPointUndirectedGraph`.

:raises: **LabellingError**: If the given labelled point graph/pointcloud contains less than the expected number of points.

## 1.6 menpo.math

### 1.6.1 Decomposition

#### eigenvalue_decomposition

```python
menpo.math.eigenvalue_decomposition(C, is_inverse=False, eps=1e-10)
```

Eigenvalue decomposition of a given covariance (or scatter) matrix.

**Parameters**

• **C** *(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)* – If 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

```python
limit = np.max(np.abs(eigenvalues)) * eps
```

**Returns**

• **pos_eigenvectors** *(ndarray)* – The matrix with the eigenvectors corresponding to positive eigenvalues.

• **pos_eigenvalues** *(ndarray)* – The array of positive eigenvalues.
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**
- \( X ((n\_samples, n\_dims) \text{ndarray}) \) – 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) \text{ndarray})` – Eigenvectors of the data matrix.
- `l (eigenvalues) ((n\_components,) \text{ndarray})` – Positive eigenvalues of the data matrix.
- `m (mean vector) ((n\_dimensions,) \text{ndarray})` – Mean that was subtracted from the data matrix.

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) \text{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) – If `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) \text{ndarray})` – Eigenvectors of the data matrix.
- `l (eigenvalues) ((n\_components,) \text{ndarray})` – Positive eigenvalues of the data matrix.
Menpo Documentation, Release 0.9.0

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, l_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

1.6.2 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.

Returns **c** ((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**

```python
menpo.math.dot_inplace_left(a, b, block_size=1000)
```

Inplace dot product for memory efficiency. It computes \( a \cdot b = c \), 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`, \(n\_small \leq k\)) – 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.

**Returns**

- **c** (\((n\_big, n\_small)\) `ndarray`) – The output of the operation. Exactly the same as a memory view onto \(a[\[:, :n\_small]]\) as \(a\) is modified inplace to store the result.

**as_matrix**

```python
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** (\((\text{length}, n\_features)\) `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**

```python
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.

### 1.6.3 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
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default 3D</td>
<td>4</td>
</tr>
</tbody>
</table>

- **num_phi_orientations** (int, optional) – Number of filter orientations in the xy-plane

|  | Default 2D | 6
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default 3D</td>
<td>6</td>
</tr>
</tbody>
</table>

- **num_theta_orientations** (int, optional) – Only required for 3D. Number of filter orientations in the z-plane

|  | Default 2D | N/A
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default 3D</td>
<td>4</td>
</tr>
</tbody>
</table>

- **min_wavelength** (int, optional) – Wavelength of smallest scale filter.

|  | Default 2D | 3
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default 3D</td>
<td>3</td>
</tr>
</tbody>
</table>

- **scaling_constant** (int, optional) – Scaling factor between successive filters.

|  | Default 2D | 2
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default 3D</td>
<td>2</td>
</tr>
</tbody>
</table>
• **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.

<table>
<thead>
<tr>
<th></th>
<th>Default 2D</th>
<th>Default 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>0.65</td>
<td>0.65</td>
</tr>
</tbody>
</table>

• **d_phi_sigma** *(float, optional)* – Angular bandwidth in xy-plane

<table>
<thead>
<tr>
<th></th>
<th>Default 2D</th>
<th>Default 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>1.3</td>
<td>1.5</td>
</tr>
</tbody>
</table>

• **d_theta_sigma** *(float, optional)* – **Only required for 3D**. Angular bandwidth in z-plane

<table>
<thead>
<tr>
<th></th>
<th>Default 2D</th>
<th>Default 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>N/A</td>
<td>1.5</td>
</tr>
</tbody>
</table>

**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**

### 1.7 menpo.model

#### 1.7.1 Abstract Classes

**LinearModel**

```
menpo.model.LinearModel
```

alias of LinearVectorModel
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.

Parameters components((n_components, n_features) ndarray) – The components array.

component(index)
    A particular component of the model.

Parameters index(int) – The component that is to be returned.

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 the 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.

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.
\[ \text{dot}(\text{component}_\text{vector}(i), \text{component}_\text{vector}(j)) = \text{dirac_delta}. \]

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 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.

property components
The components matrix of the linear model.

Type (n_available_components, n_features) ndarray

property n_components
The number of bases of the model.

Type int
Menpo Documentation, Release 0.9.0

property n_features
The number of elements in each linear component.

   Type  int

MeanLinearModel

menpo.model.MeanLinearModel
alias of MeanLinearVectorModel

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.

**property components**

The components matrix of the linear model.

Type `(n_available_components, n_features) ndarray`

**property n_components**

The number of bases of the model.

Type `int`

**property n_features**

The number of elements in each linear component.

Type `int`

### 1.7.2 Principal Component Analysis

**PCAModel**

```python
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 its 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** (*index, with_mean=True, scale=1.0*)
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** (*array_like*) – 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** (*array_like*) – 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 samples. 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** *(ndarray)* – The eigenvectors to be used.

• **eigenvalues** *(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** *(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)* – If `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 \( \text{len}(\text{weights}) \) 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.

**Raises** ValueException – If \( \text{n_weights} > \text{n_components} \)

**Returns** instance *(type(self.template_instance))* – An instance of the model.

**instance_vector** *(weights, normalized_weights=False)*  
Creates a new instance of the model using the first \( \text{len}(\text{weights}) \) components.

**Parameters**

- **weights** *(n_weights,)*  
  *ndarray or list*  
  weights[i] is the linear contribution of the i’th component to the instance vector.

**Raises** ValueException – If \( \text{n_weights} > \text{n_components} \)

**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*  
  weights[i, j] is the linear contribution of the j’th principal component to the i’th instance vector produced. Note that if \( \text{n_weights} < \text{n_components} \), only the first \( \text{n_weights} \) 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)*  
The instance vectors for the weighting provided.

**Raises** ValueException – If \( \text{n_weights} > \text{n_components} \)

**inverse_noise_variance** ()  
Returns the inverse of the noise variance.

**Returns** inverse_noise_variance *(float)* – Inverse of the noise variance.

**Raises** ValueException – If \( \text{noise_variance}() == 0 \)

**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 \( \text{noise_variance} == 0.0 \).
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 lose 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

```
```
or
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
**marker_style** *(See Below, optional)* – The style of the markers. Example options

```python
['.', ',', '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

```python
['r', 'g', 'b', 'c', 'm', 'k', 'w']
```

or

```python```
```
3, )` 'ndarray`
```

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

```python
['r', 'g', 'b', 'c', 'm', 'k', 'w']
```

or

```python```
```
3, )` 'ndarray`
```

**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

```python
['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

```python
['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 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'
  ```
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

```python
('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

```python
('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_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', '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
• **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** (*instance_vector*)
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 vec-
tors.
Returns `projected_out` `((n_vectors, n_features) ndarray)` – A copy of `vectors` with all bases of the model projected out.

**project_vector** *(instance_vector)*
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,) ndarray)` – A vector of whitened linear weightings.

**project_whitened_vector** *(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** *(instance)*
Projects a `instance` onto the linear space and rebuilds from the weights found.

Syntactic sugar for:

```python
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** *(instance_vector)*
Projects a `instance` onto the linear space and rebuilds from the weights found.

Syntactic sugar for:

```python
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.

- **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.

Returns variance_ratio (float) – Ratio of active components variance and total variance present in original samples.

view_widget (figure_size=(7, 7))

Visualizes the model using an interactive widget. It only works if it is a 2D/3D shape or appearance model.

Parameters figure_size ((int, int), optional) – The initial size of the rendered figure.

whitened_components ()

Returns the active components of the model, whitened.

Returns whitened_components ((n_active_components, n_features) ndarray) – The whitened components.

property components

Returns the active components of the model.

Type (n_active_components, n_features) ndarray

property 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

property mean_vector

Return the mean of the model as a 1D vector.

Type ndarray

property n_active_components

The number of components currently in use on this model.
Type `int`

**property n_components**
The number of bases of the model.

**Type** `int`

**property n_features**
The number of elements in each linear component.

**Type** `int`

### PCAVectorModel

#### class `menpo.model.PCAVectorModel`

```python
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 its 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.

Returns \( \text{type}(\text{self}) - \) A copy of this object

\text{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 \( \text{eigenvalues\_cumulative\_ratio}((n\_active\_components,)) \text{ndarray} - \) Array of cumulative eigenvalues.

\text{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 \( \text{eigenvalues\_ratio}((n\_active\_components,)) \text{ndarray} - \) The active eigenvalues array scaled by the original variance.

\text{increment}\,(\text{data}, \text{n\_samples=\text{None}}, \text{forgetting\_factor=1.0}, \text{verbose=False})

Update the eigenvectors, eigenvalues and mean vector of this model by performing incremental PCA on the given samples.

Parameters

- \text{samples} (list of \text{Vectorizable}) – List of new samples to update the model from.
- \text{n\_samples} (int, optional) – If provided then \text{samples} must be an iterator that yields \text{n\_samples}. If not provided then \text{samples} has to be a list (so we know how large the data matrix needs to be).
- \text{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

\text{classmethod init\_from\_components}(\text{components, eigenvalues, mean, n\_samples, centred, max\_n\_components=\text{None}})

Build the Principal Component Analysis (PCA) using the provided components (eigenvectors) and eigenvalues.

Parameters

- \text{components} ((n\_components, n\_features) \text{ndarray}) – The eigenvectors to be used.
- \text{eigenvalues} ((n\_components, ) \text{ndarray}) – The corresponding eigenvalues.
- \text{mean} ((n\_features, ) \text{ndarray}) – The mean vector.
- \text{n\_samples} (int) – The number of samples used to generate the eigenvectors.
- \text{centred} (bool) – 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.

```python
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** (*ndarray* or *scipy.sparse*) – The Covariance/Scatter matrix. If it is a precision matrix (inverse covariance), then set `is_inverse=True`.
- **mean** (*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) – If 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.

```python
instance(weights, normalized_weights=False)
```
Creates a new vector instance of the model by weighting together the components.

**Parameters**

- **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** (*ndarray*) – The instance vector for the weighting provided.

```python
instance_vectors(weights, normalized_weights=False)
```
Creates new vectorized instances of the model using the first components in a particular weighting.

**Parameters**

- **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** (*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()**

Returns 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 lose 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
• **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_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
marker_edge_colour (See Below, optional) – The edge colour of the markers. Example options

- 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

- 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

- 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_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

```python
('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

```python
('.', ',', '.', '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

```python
('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

```python
('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 *(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 \texttt{projected\_out} \((\text{n\_features,}) \ \text{ndarray}\) – A copy of \texttt{vector} with all basis of the model projected out.

\texttt{project\_out\_vectors} \((\text{vectors})\)

Returns a version of \texttt{vectors} where all the bases of the model have been projected out.

\textbf{Parameters}\n
\texttt{vectors} \((\text{n\_vectors, n\_features}) \ \text{ndarray}\) – A matrix of novel vectors.

\texttt{Returns} \texttt{projected\_out} \((\text{n\_vectors, n\_features}) \ \text{ndarray}\) – A copy of \texttt{vectors} with all bases of the model projected out.

\texttt{project\_vectors} \((\text{vectors})\)

Projects each of the \texttt{vectors} onto the model, retrieving the optimal linear reconstruction weights for each instance.

\textbf{Parameters}\n
\texttt{vectors} \((\text{n\_samples, n\_features}) \ \text{ndarray}\) – Array of vectorized novel instances.

\texttt{Returns} \texttt{projected} \((\text{n\_samples, n\_components}) \ \text{ndarray}\) – The matrix of optimal linear weights.

\texttt{project\_whitened} \((\text{vector\_instance})\)

Projects the \texttt{vector\_instance} onto the whitened components, retrieving the whitened linear weightings.

\textbf{Parameters}\n
\texttt{vector\_instance} \((\text{n\_features,}) \ \text{ndarray}\) – A novel vector.

\texttt{Returns} \texttt{projected} \((\text{n\_features,}) \ \text{ndarray}\) – A vector of whitened linear weightings

\texttt{reconstruct} \((\text{vector})\)

Project a \texttt{vector} onto the linear space and rebuild from the weights found.

\textbf{Parameters}\n
\texttt{vector} \((\text{n\_features,}) \ \text{ndarray}\) – A vectorized novel instance to project.

\texttt{Returns} \texttt{reconstructed} \((\text{n\_features,}) \ \text{ndarray}\) – The reconstructed vector.

\texttt{reconstruct\_vectors} \((\text{vectors})\)

Projects the \texttt{vectors} onto the linear space and rebuilds vectors from the weights found.

\textbf{Parameters}\n
\texttt{vectors} \((\text{n\_vectors, n\_features}) \ \text{ndarray}\) – A set of vectors to project.

\texttt{Returns} \texttt{reconstructed} \((\text{n\_vectors, n\_features}) \ \text{ndarray}\) – The reconstructed vectors.

\texttt{trim\_components} \((\text{n\_components=None})\)

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 \texttt{self.n\_components} down to \texttt{n\_components} (if \texttt{None}, \texttt{self.n\_active\_components} will be used), freeing up memory in the process.

Once the model is trimmed, the trimmed components cannot be recovered.

\textbf{Parameters}\n
\texttt{n\_components} \((\text{int >= 1 or float > 0.0}} \ \text{or None, optional})\) – The number of components that are kept or else the amount (ratio) of variance that is kept. If \texttt{None}, \texttt{self.n\_active\_components} is used.

\textbf{Notes}\n
In case \texttt{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.

Returns variance_ratio (float) – Ratio of active components variance and total variance present in original samples.

whitened_components()  
Returns the active components of the model, whitened.

Returns whitened_components ((n_active_components, n_features) ndarray) – The whitened components.

property components  
Returns the active components of the model.

Type (n_active_components, n_features) ndarray

property 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

property n_active_components  
The number of components currently in use on this model.

Type int

property n_components  
The number of bases of the model.

Type int

property n_features  
The number of elements in each linear component.

Type int

1.7.3 Gaussian Markov Random Field

GMRFModel

class menpo.model.GMRFModel(samples, graph, mode='concatenation', n_components=None, dtype=<class '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 \(x_i\) and \(x_j\) 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

\[
[x_i^T, x_j^T]^T
\]

and in the case of 'subtraction'

\[
x_i - x_j
\]

• **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 components.

• **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. \(x_i, i : v_i \in V\).

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

\[
p(X = x | G) \sim \mathcal{N}(\mu, \Sigma)
\]

We denote by \(Q\) the block-sparse precision matrix that is the inverse of the covariance matrix \(\Sigma\), i.e. \(Q = \Sigma^{-1}\). 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.

\[
Q_{ij} = 0_{k \times k}, \forall i, j : (v_i, v_j) \notin E
\]

## References

1.7. `menpo.model`
**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 $x$ or an array of samples $X$, i.e.

$$\sqrt{(x - \mu)^T \Sigma (x - \mu)}$$

**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**

A class for modeling 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.
Menpo Documentation, Release 0.9.0

• **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).

• **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

$$[\mathbf{x}_i^T, \mathbf{x}_j^T]^T$$

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 components.

• **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 $V = \{v_1, v_2, \ldots, v_{|V|}\}$ 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

$$p(X = \mathbf{x}|G) \sim \mathcal{N}(\mu, \Sigma)$$

We denote by $Q$ the block-sparse precision matrix that is the inverse of the covariance matrix $\Sigma$, i.e. $Q = \Sigma^{-1}$. 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.

$$Q_{ij} = 0_{k \times k}, \forall i, j : (v_i, v_j) \notin E$$

**References**

1.7. menpo.model
**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 x or an array of samples X, i.e.

\[
\sqrt{(x - \mu)^T Q (x - \mu)} \quad \text{or} \quad \sqrt{(X - \mu)^T Q (X - \mu)}
\]

**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.

## 1.8 menpo.shape

### 1.8.1 Base Class

**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

property has_landmarks
    Whether the object has landmarks.
    
    Type bool

property landmarks
    The landmarks object.
    
    Type LandmarkManager

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

property n_parameters
    The length of the vector that this object produces.
    
    Type int
1.8.2 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=(7, 7), 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
• **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
  
  ```python
  {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
  
  ```python
  {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
  
  ```python
  {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
  
  ```python
  {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
  
  ```python
  {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 Point-
Cloud'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.

Returns **viewer** (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_markers=True, marker_style='s', marker_size=7, marker_face_colour='k', marker_edge_width=1.0, render_lines_lms=True, line_colour_lms=None, line_style_lms='- ', line_width_lms=1, render_markers_lms=True, marker_style_lms='o', marker_size_lms=5, marker_face_colour_lms=None, marker_edge_colour_lms=None, marker_edge_width_lms=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', 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_pixels_width=1.0, legend_pixels_height=1.0, 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=(7, 7))

Visualize the landmarks. This method will appear on the PointCloud as `view_landmarks`.

**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_markers** (bool, optional) – If True, the markers will be rendered.
- **marker_style** (See Below, optional) – The style of the markers. Example options
• **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_lines_lms** (*bool, optional*) – If True, the edges of the landmarks will be rendered.

• **line_colour_lms** (*See Below, optional*) – The colour of the lines of the landmarks. Example options:

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

• **line_style_lms** (*{-, --, -., :}, optional*) – The style of the lines of the landmarks.

• **line_width_lms** (*float, optional*) – The width of the lines of the landmarks.

• **render_markers** – If True, the markers of the landmarks will be rendered.

• **marker_style** – The style of the markers of the landmarks. Example options

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

• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **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:
• **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, 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 \texttt{tuple} or \texttt{list}, then it defines the axis limits. If \texttt{None}, then the limits are set automatically.

- \texttt{axes_x_ticks} (\texttt{list} or \texttt{tuple} or \texttt{None}, optional) – The ticks of the x axis.
- \texttt{axes_y_ticks} (\texttt{list} or \texttt{tuple} or \texttt{None}, optional) – The ticks of the y axis.
- \texttt{figure_size} ((\texttt{float}, \texttt{float}) \texttt{tuple} or \texttt{None} optional) – The size of the figure in inches.

\textbf{Raises}

- \texttt{ValueError} – If both \texttt{with_labels} and \texttt{without_labels} are passed.
- \texttt{ValueError} – If the landmark manager doesn’t contain the provided group label.

\texttt{as_vector(**kwargs)}

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

\textbf{Returns} \texttt{vector} ((\texttt{N}) \texttt{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.

\texttt{bounding_box()}

Return a bounding box from two corner points as a directed graph. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

\begin{verbatim}
0<--3
|   ^
|   |
| v  |
1-->2
\end{verbatim}

In the case of a pointcloud, the ordering will appear as:

\begin{verbatim}
3<--2
|   ^
|   |
| v  |
0-->1
\end{verbatim}

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

\textbf{Returns} \texttt{bounding_box (PointDirectedGraph)} – The axis aligned bounding box of the PointCloud.

\texttt{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.

\textbf{Parameters} \texttt{boundary} (\texttt{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.

\textbf{Returns}

- \texttt{min_b} ((\texttt{n_dims,}) \texttt{ndarray}) – The minimum extent of the \texttt{PointCloud} and boundary along each dimension
- \texttt{max_b} ((\texttt{n_dims,}) \texttt{ndarray}) – The maximum extent of the \texttt{PointCloud} and boundary along each dimension

\texttt{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.

**with_dims (dims)**
Return a copy of this shape with only particular dimensions retained.

Parameters **dims** (*valid numpy array slice*) – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, [0, 1] could be provided or np.array([True, True, False]).

Returns **copy of self, with only the requested dims**

**property has_landmarks**
Whether the object has landmarks.

**Type** *bool*

**property landmarks**
The landmarks object.

**Type** *LandmarkManager*

**property lms**
Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the 'underlying' *PointCloud* subclass.

**Type** *self*

**property n_dims**
The number of dimensions in the pointcloud.

**Type** *int*

**property n_landmark_groups**
The number of landmark groups on this object.

**Type** *int*

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

**Type** *int*

**property n_points**
The number of points in the pointcloud.

**Type** *int*

### 1.8.3 Graphs

**UndirectedGraph**

```python
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 as-
  signment.

• 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, ), ({adjac-
  ency_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

```python
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0],
                             [1, 0, 1, 1, 0, 0],
                             [1, 1, 0, 1, 0, 0],
                             [0, 1, 0, 0, 1, 1],
                             [0, 0, 1, 1, 0, 0],
                             [0, 0, 0, 1, 0, 0]])
graph = UndirectedGraph(adjacency_matrix)
```

or

```python
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
can be defined as

```python
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

```python
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.

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

- **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 \( n_{\text{vertices}} \) 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 \( n_{\text{vertices}} \)) – 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.

classmethod init_from_edges (edges, n_vertices, skip_checks=False)
Initialize graph from edges array.

**Parameters**
- **edges** (\((n_{\text{edges}}, 2, ) \) 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
```
(continues on next page)
can be defined as

```python
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}.

property edges
Returns the ndarray of edges, i.e. all the pairs of vertices that are connected with an edge.

Type (n_edges, 2, ) ndarray

property n_edges
Returns the number of edges.

Type int

property n_vertices
Returns the number of vertices.

Type int

property 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

```python
import numpy as np
adjacency_matrix = np.array([ [0, 0, 0, 0, 0, 0],
                               [1, 0, 1, 1, 0, 0],
                               [1, 1, 0, 1, 0, 0],
                               [0, 0, 0, 0, 1, 1],
                               [0, 0, 0, 0, 0, 0],
                               [0, 0, 0, 0, 0, 0] ])

graph = DirectedGraph(adjacency_matrix)
```

or

```python
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
|   |
 v  
 5
```

can be defined as

```python
import numpy as np
adjacency_matrix = np.array([ [0, 0, 0, 0, 0, 0],
                               [1, 0, 1, 1, 0, 0],
                               [1, 1, 0, 1, 0, 0],
                               [0, 0, 0, 0, 1, 1],
                               [0, 0, 0, 0, 0, 0],
                               [0, 0, 0, 0, 0, 0] ])

graph = DirectedGraph(adjacency_matrix)
```
```python
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

```python
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:
  
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘dijkstra’</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>‘bellman-ford’</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>‘johnson’</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>‘floyd-warshall’</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>‘auto’</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>
  
- **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**
Menpo Documentation, Release 0.9.0

• **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:

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

• **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.
Returns **adjacency_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.

**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

```python
from menpo.shape import UndirectedGraph
import numpy as np
deges = 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
```

(continues on next page)
can be represented as

```python
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

```python
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 \([0, \ n_\text{vertices} - 1]\).

\[\text{n_parents}(\text{vertex}, \skip\text{checks}=\text{False})\]

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 \([0, \ n_\text{vertices} - 1]\).

\[\text{n_paths}(\text{start}, \text{end})\]

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.

\[\text{parents}(\text{vertex}, \skip\text{checks}=\text{False})\]

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 \([0, \ n_\text{vertices} - 1]\).

\[\text{property \text{edges}}\]

Returns the `ndarray` of edges, i.e. all the pairs of vertices that are connected with an edge.

Type `(n_\text{edges}, 2, ) \text{ndarray}`

\[\text{property n_edges}\]

Returns the number of edges.

Type `int`

\[\text{property n_vertices}\]

Returns the number of vertices.

Type `int`

\[\text{property vertices}\]

Returns the `list` of vertices.

Type `list`
Tree

```python
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

```python
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]])
```
```python
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.

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:
  - ‘auto’
  - ‘bellman_ford’
  - ‘dijkstra’
  - ‘floyd_warshall’

...
Menpo Documentation, Release 0.9.0

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra's algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson's algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

- **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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>
• **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.

**Returns**  
*adjacency_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.

**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

```python
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.

`property edges`
Returns the `ndarray` of edges, i.e. all the pairs of vertices that are connected with an edge.

Type `(n_edges, 2,) ndarray`

`property leaves`
Returns a `list` with all the leaves of the tree.

Type `list`
**property maximum_depth**

Returns the maximum depth of the tree.

**Type** int

**property n_edges**

Returns the number of edges.

**Type** int

**property n_leaves**

Returns the number of leaves of the tree.

**Type** int

**property n_vertices**

Returns the number of vertices.

**Type** int

**property vertices**

Returns the list of vertices.

**Type** list

### 1.8.4 PointGraphs

Mix-ins of Graphs and *PointCloud* for graphs with geometry.

**PointUndirectedGraph**

```python
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

```python
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0],
                            [1, 0, 1, 1, 0, 0],
                            [1, 1, 0, 1, 0, 1],
                            [0, 1, 1, 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

```python
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

```python
import numpy as np
adjacency_matrix = np.array([[0, 1, 1, 0, 0, 0],
                            [1, 0, 1, 1, 0, 0],
                            [1, 1, 0, 1, 0, 1],
                            [0, 1, 1, 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]])
```
```python
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

```python
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=(7, 7), label=None, **kwargs)

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:

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

- **line_style** (See Below, 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:
• **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

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

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

```python
{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)* – The horizontal alignment of the numbers’ texts.

• **numbers_vertical_align** *(center, top, bottom, baseline)* – The vertical alignment of the numbers’ texts.

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

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

• **numbers_font_size** *(int, optional)* – The font size of the numbers.

• **numbers_font_style** *(normal, italic, oblique)* – The font style of the numbers.

• **numbers_font_weight** *(See Below, optional)* – The font weight of the numbers. Example options

```python
{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

```python
{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

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

• **axes_font_size** *(int, optional)* – The font size of the axes.

• **axes_font_style** *(normal, italic, oblique)* – 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.

**Returns** viewer *(PointGraphViewer2d)* – The viewer object.


Visualize the landmarks. This method will appear on the `PointGraph` as `view_landmarks`.

**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\}
  \textbf{or}
  (3, ) \text{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\}
  \textbf{or}
  (3, ) \text{ndarray}
  ```

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

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

• **marker_edge_width** (*float*, optional) – The width of the markers’ edge.

• **render_lines_lms** (*bool*, optional) – If True, the edges of the landmarks will be rendered.

• **line_colour_lms** (*See Below, optional*) – The colour of the lines of the landmarks. Example options:

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

• **line_style_lms** (*\{-, --, -., :\}, optional*) – The style of the lines of the landmarks.

• **line_width_lms** (*float*, optional) – The width of the lines of the landmarks.

• **render_markers** – If True, the markers of the landmarks will be rendered.

• **marker_style** – The style of the markers of the landmarks. Example options
• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options

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

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

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

• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **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:

  ```
  
<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'best'</td>
<td>0</td>
</tr>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>
  ```

• **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, 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. In the case of a 2D pointcloud, 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
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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 be-
tween. 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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

• **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 its vectorized state.

`self` is used to fill out the missing state required to rebuild a full object from its 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_doms + 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, 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.

classmethod 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** \(\text{(n_vertices, n_vertices)}\) `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** \(\text{(n_vertices, n_dims, ) ndarray}\) – The array of point locations.
- **edges** \(\text{(n_edges, 2, ) ndarray}\) – 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

```python
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
can be defined as

```python
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 a 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** *(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.
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}`.

`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 PointGraph to a dictionary representation suitable for inclusion in the LJSON landmark format.

Returns `json (dict)`: Dictionary with `points` and `connectivity` keys.

`with_dims (dims)`
Return a copy of this shape with only particular dimensions retained.

Parameters `dims (valid numpy array slice)`: The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, `[0, 1]` could be provided or `np.array([True, True, False])`.

Returns `copy of self, with only the requested dims`.

`property edges`
Returns the `ndarray` of edges, i.e. all the pairs of vertices that are connected with an edge.

Type `(n_edges, 2, ) ndarray`
**property has_landmarks**
Whether the object has landmarks.
Type bool

**property landmarks**
The landmarks object.
Type LandmarkManager

**property lms**
Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the ‘underlying’ PointCloud subclass.
Type self

**property n_dims**
The number of dimensions in the pointcloud.
Type int

**property n_edges**
Returns the number of edges.
Type int

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

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

**property n_points**
The number of points in the pointcloud.
Type int

**property n_vertices**
Returns the number of vertices.
Type int

**property 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

```python
import numpy as np
adjacency_matrix = np.array([[0, 0, 0, 0, 0, 0],
                             [1, 1, 0, 0, 1, 0],
                             [1, 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

```python
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

```python
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

```python
from scipy.sparse import csr_matrix
def 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_line_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=(7, 7), label=None, **kwargs)  

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:
• **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
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.

Returns viewer (PointGraphViewer2d) – The viewer object.
Visualize the landmarks. This method will appear on the PointGraph as `view_landmarks`.

**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
- **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:

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

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

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

- **marker_edge_width** *(float, optional)* – The width of the markers’ edge.

- **render_lines_lms** *(bool, optional)* – If True, the edges of the landmarks will be rendered.

- **line_colour_lms** *(See Below, optional)* – The colour of the lines of the landmarks. Example options:

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

- **line_style_lms** *(\{-, --, -, ., :\}, optional)* – The style of the lines of the landmarks.

- **line_width_lms** *(float, optional)* – The width of the lines of the landmarks.

- **render_markers** – If True, the markers of the landmarks will be rendered.

- **marker_style** – The style of the markers of the landmarks. Example options:

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

- **marker_size** – The size of the markers of the landmarks in points.

- **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options:

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

- **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options:

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

- **marker_edge_width** – The width of the markers’ edge of the landmarks.

- **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

```python
{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

```python
{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

```python
{r, g, b, c, m, k, w}
```

or

```python
(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

```python
{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

```python
{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:
• **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

\[
\{\text{serif, sans-serif, cursive, fantasy, monospace}\}
\]

• **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

\[
\{\text{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black}\}
\]

• **axes_x_limits** ((float, float) tuple 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. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
\|   \\
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
\|   \\
0-->1
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>
  
- **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:
Menpo Documentation, Release 0.9.0

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

- **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 PointDirectedGraph 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 its vectorized state.

- **self** is used to fill out the missing state required to rebuild a full object from its 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()*.

**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.

`classmethod 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.

`classmethod 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 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
|     |
```

can be defined as

```python
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<--|
|     |
|     |
```

(continues on next page)
can be represented as

```python
from menpo.shape import PointDirectedGraph
import numpy as np
def points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                      [0, 0]])
def edges = np.array([[1, 0], [2, 0], [1, 2], [2, 1], [1, 3], [2, 4],
                      [3, 4], [3, 5]])
def graph = PointDirectedGraph.init_from_edges(points, edges)
```

Finally, the following graph with isolated vertices

```python
0---|
|    |
1 2
|    |
|
3------4
5
```

can be defined as

```python
from menpo.shape import PointUndirectedGraph
import numpy as np
def points = np.array([[10, 30], [0, 20], [20, 20], [0, 10], [20, 10],
                      [0, 0]])
def edges = np.array([[0, 2], [2, 0], [2, 4], [4, 2], [3, 4], [4, 3]])
def 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

\[
 l_i - l_j = \begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} x_j \\ y_j \end{bmatrix} = \begin{bmatrix} x_i - x_j \\ y_i - y_j \end{bmatrix}
\]

**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:

\[
 l_i - l_j = \begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} x_j \\ y_j \end{bmatrix} = \begin{bmatrix} x_i - x_j \\ y_i - y_j \end{bmatrix}
\]

**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.

**with_dims** (*dims*)

Return a copy of this shape with only particular dimensions retained.

**Parameters** **dims** (*valid numpy array slice*) – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, [0, 1] could be provided or np.array([True, True, False]).

**Returns** **copy of self, with only the requested dims**

**property edges**

Returns the *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge.

**Type** (*n_edges, 2, *) *ndarray*

**property has_landmarks**

Whether the object has landmarks.

**Type** *bool*

**property landmarks**

The landmarks object.

**Type** *LandmarkManager*
property lms
    Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this
    object, which previously would have held the ‘underlying’ PointCloud subclass.

    Type  self

property n_dims
    The number of dimensions in the pointcloud.

    Type  int

property n_edges
    Returns the number of edges.

    Type  int

property n_landmark_groups
    The number of landmark groups on this object.

    Type  int

property n_parameters
    The length of the vector that this object produces.

    Type  int

property n_points
    The number of points in the pointcloud.

    Type  int

property n_vertices
    Returns the number of vertices.

    Type  int

property 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.

      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 as-
      signment.
• **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 \(n\), \(n\) instead of \((n, n)\).

• **ValueError** – The provided edges do not represent a tree.

• **ValueError** – The root_vertex must be in the range \([0, n-1]\).

• **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

```python
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

```python
from scipy.sparse import csr_matrix
adjacency_matrix = csr_matrix(([1] * 8, ([1], [0, 0, 1, 1, 2, 3, 4, 5]),
                               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)
```

(continues on next page)
tree = PointTree(points, adjacency_matrix, root_vertex=0)

```python
_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=(7, 7),
    label=None,**kwargs)
```

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:

  ```python
  {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

  ```python
  {., , 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

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

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

  ```python
  {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
  
  ```{mermaid}
  {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
  
  ```{mermaid}
  {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
  
  ```{mermaid}
  {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
  
  ```{mermaid}
  {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
  
  ```{mermaid}
  {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.

Returns viewer (PointGraphViewer2d) – The viewer object.

```python
_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='k', line_style='-', line_width=2, render_markers=True, marker_style='s', marker_size=7, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_lines_lms=True, line_colour_lms=None, line_style_lms='-', line_width_lms=1, render_markers_lms=True, marker_style_lms='o', marker_size_lms=5, marker_face_colour_lms=None, marker_edge_colour_lms=None, marker_edge_width_lms=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, 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=(7, 7))
```

Visualize the landmarks. This method will appear on the `PointGraph` as `view_landmarks`.

**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_lines_lms** *(bool, optional)* – If True, the edges of the landmarks will be rendered.

• **line_colour_lms** *(See Below, optional)* – The colour of the lines of the landmarks. Example options:

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

• **line_style_lms** *(–, --, -. , : ), optional)* – The style of the lines of the landmarks.

• **line_width_lms** *(float, optional)* – The width of the lines of the landmarks.

• **render_markers** – If True, the markers of the landmarks will be rendered.

• **marker_style** – The style of the markers of the landmarks. Example options

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

• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

```
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```
• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **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

```python
{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

```python
{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

```python
{r, g, b, c, m, k, w}
```

• **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

```python
{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

```python
{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:
• `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. In the case of a 2D pointcloud, 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
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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

**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.

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:
  
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dijkstra</code></td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td><code>bellman-ford</code></td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td><code>johnson</code></td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td><code>floyd-warshall</code></td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td><code>auto</code></td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>
- **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.

Parameters *mask* (*n_points,*) *ndarray* – 1D array of booleans

Returns **pointtree** (*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 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 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 \textit{nan} values or not. This is particularly useful for objects
with unknown values that have been mapped to \textit{nan} values.

Returns has_nan_values (bool) – If the vectorized object contains \textit{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

• \textbf{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.

• \textbf{spacing (int or tuple of 2 int, optional)} – The spacing between points. If a single \textit{int}
  is provided, this is applied uniformly across each dimension. If a \textit{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.

• \textbf{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.

• \textbf{root_vertex (int)} – The vertex to be set as root.

• \textbf{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 area 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** (*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 **depth_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** (*ndarray*) – The array of point locations.

- **edges** (*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

```
1.8. menpo.shape
```
can be defined as

```python
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, skipChecks=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

\[
\mathbf{l}_i - \mathbf{l}_j = ([x_i], [y_i]) - ([x_j], [y_j]) = \\
= ([x_i - x_j], [y_i - y_j])
\]

**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:

\[
\mathbf{l}_i - \mathbf{l}_j = ([x_i], [y_i]) - ([x_j], [y_j]) = \\
= ([x_i - x_j], [y_i - y_j])
\]

**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.

**with_dims** *(dims)*

Return a copy of this shape with only particular dimensions retained.

**Parameters**

- **dims** *(valid numpy array slice)* – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, [0, 1] could be provided or np.array([True, True, False]).

**Returns**

**copy of self, with only the requested dims**
**property has_landmarks**
Whether the object has landmarks.

   **Type** bool

**property landmarks**
The landmarks object.

   **Type** LandmarkManager

**property leaves**
Returns a list with the all leaves of the tree.

   **Type** list

**property lms**
Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the ‘underlying’ PointCloud subclass.

   **Type** self

**property maximum_depth**
Returns the maximum depth of the tree.

   **Type** int

**property n_dims**
The number of dimensions in the pointcloud.

   **Type** int

**property n_edges**
Returns the number of edges.

   **Type** int

**property n_landmark_groups**
The number of landmark groups on this object.

   **Type** int

**property n_leaves**
Returns the number of leaves of the tree.

   **Type** int

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

   **Type** int

**property n_points**
The number of points in the pointcloud.

   **Type** int

**property n_vertices**
Returns the number of vertices.

   **Type** int

**property vertices**
Returns the list of vertices.

   **Type** list
1.8.5 LabelledPointGraph

A subclass of `PointUndirectedGraph` that allows the attaching of `labels` associated with semantic parts of the object.

**LabelledPointUndirectedGraph**

class `menpo.shape.LabelledPointUndirectedGraph(points, adjacency_matrix, labels_to_masks, copy=True, skip_checks=False)`

Bases: `PointUndirectedGraph`

A subclass of `PointUndirectedGraph` that allows the attaching of ‘labels’ associated with semantic parts of an object. For example, for a face the semantic parts might be the eyes, nose and mouth. These ‘labels’ are defined as a dictionary of string keys that map to boolean mask arrays that define which of the underlying points belong to a given label.

The labels to masks must be within an `OrderedDict` so that semantic ordering can be maintained.

**Parameters**

- **points**(ndarray) – The points representing the landmarks.
- **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.
- **labels_to_masks**(ordereddict {str -> bool ndarray}) – For each label, the mask that specifies the indices in to the points that belong to the label.
- **copy**(bool, optional) – If True, a copy of the data is stored.

**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 points.
- **ValueError** – If there exists any point in the points that is not covered by a label.

`_view_2d`(with_labels=None, without_labels=None, group='group', 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=True, legend_title=None, 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', axes_x Limits=None, axes_y Limits=None, axes_x_ticks=None, axes_y_ticks=None, figure_size=(10, 8))

Visualize the labelled point undirected graph.
Parameters

- **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.
- **group** *(str or None, optional)* – The name of the labelled point undirected graph. It is used in the legend.
- **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 x and y axes are flipped.
- **render_lines** *(bool, optional)* – If True, the edges will be rendered.
- **line_colour** *(See Below, optional)* – The colour of the lines. Example options:

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

It can either be one of the above or a list of those defining a value per label.
- **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

```python
{., ,, 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

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

It can either be one of the above or a list of those defining a value per label.
- **marker_edge_colour** *(See Below, optional)* – The edge colour of the markers. Example options

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

It can either be one of the above or a list of those defining a value per label.
- **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. Possible 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. Possible 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}
  ```
- `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 LabelledPointUndirectedGraph as a percentage of the LabelledPointUndirectedGraph’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 LabelledPointUndirectedGraph as
a percentage of the LabelledPointUndirectedGraph’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) or None, optional) – The size of the figure in inches.

Raises **ValueError** – If both with_labels and without_labels are passed.

```python
_view_landmarks_2d(group=None, with_labels=None, without_labels=None, figure_id=None, new_figure=True, render_lines=True, line_colour='k', line_style='-', line_width=2, render_markers=True, marker_style='s', marker_size=7, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_lines_lms=True, line_colour_lms=None, line_style_lms='-', line_width_lms=1, render_markers_lms=True, marker_style_lms='o', marker_size_lms=5, marker_face_colour_lms=None, marker_edge_colour_lms=None, marker_edge_width_lms=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=(7, 7))
```

Visualize the landmarks. This method will appear on the `PointGraph` as `view_landmarks`.

**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:
• 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\}

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

\{r, g, b, c, m, k, w\}

• marker_edge_width (float, optional) – The width of the markers’ edge.
• render_lines_lms (bool, optional) – If True, the edges of the landmarks will be rendered.
• line_colour_lms (See Below, optional) – The colour of the lines of the landmarks. Example options:

\{r, g, b, c, m, k, w\}

• line_style_lms (\{-, --, -., :\}, optional) – The style of the lines of the landmarks.
• line_width_lms (float, optional) – The width of the lines of the landmarks.
• render_markers – If True, the markers of the landmarks will be rendered.
• marker_style – The style of the markers of the landmarks. Example options

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

• marker_size – The size of the markers of the landmarks in points.
• marker_face_colour – The face (filling) colour of the markers of the landmarks. Example options
**marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

```
{r, g, b, c, m, k, w}
```

**marker_edge_width** – The width of the markers’ edge of the landmarks.

**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}
```
Menpo Documentation, Release 0.9.0

- **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:

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'best'</td>
<td>0</td>
</tr>
<tr>
<td>'upper right'</td>
<td>1</td>
</tr>
<tr>
<td>'upper left'</td>
<td>2</td>
</tr>
<tr>
<td>'lower left'</td>
<td>3</td>
</tr>
<tr>
<td>'lower right'</td>
<td>4</td>
</tr>
<tr>
<td>'right'</td>
<td>5</td>
</tr>
<tr>
<td>'center left'</td>
<td>6</td>
</tr>
<tr>
<td>'center right'</td>
<td>7</td>
</tr>
<tr>
<td>'lower center'</td>
<td>8</td>
</tr>
<tr>
<td>'upper center'</td>
<td>9</td>
</tr>
<tr>
<td>'center'</td>
<td>10</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>Font Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>serif</td>
</tr>
<tr>
<td>sans-serif</td>
</tr>
<tr>
<td>cursive</td>
</tr>
<tr>
<td>fantasy</td>
</tr>
<tr>
<td>monospace</td>
</tr>
</tbody>
</table>

- **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
• **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.

### add_label(label, indices)

Add a new label by creating a new mask over the points. A new *LabelledPointUndirectedGraph* is returned.

**Parameters**

• **label** *(string)* – Label of landmark.

• **indices** *(K,) ndarray)* – Array of indices in to the points. Each index implies membership to the label.

**Returns** *labelled_pointgraph* *(LabelledPointUndirectedGraph)* – A new labelled pointgraph with the new label specified by indices.

### 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. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|    |
| ^  |
|    |
1-->2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|    |
| ^  |
|    |
0-->1
```
In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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 *LabelledPointUndirectedGraph*.

**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. 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.
Menpo Documentation, Release 0.9.0

- **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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘dijkstra’</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>‘bellman-ford’</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>‘johnson’</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>‘floyd-warshall’</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>‘auto’</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

- **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. \( \text{distances}[i, j] \) 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 \( \text{predecessors}[i, j] \) 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 \( \text{predecessors}[i, j] = -9999 \).

**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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dijkstra'</td>
<td>Dijkstra’s algorithm with Fibonacci heaps</td>
</tr>
<tr>
<td>'bellman-ford'</td>
<td>Bellman-Ford algorithm</td>
</tr>
<tr>
<td>'johnson'</td>
<td>Johnson’s algorithm</td>
</tr>
<tr>
<td>'floyd-warshall'</td>
<td>Floyd-Warshall algorithm</td>
</tr>
<tr>
<td>'auto'</td>
<td>Select the best among the above</td>
</tr>
</tbody>
</table>

• **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.

**get_label** *(label)*

Returns a new `PointUndirectedGraph` that contains the subset of points that this label represents.

**Parameters**

- **label** (string) – Label to filter on.
Returns **graph** (*PointUndirectedGraph*) – The `PointUndirectedGraph` containing the subset of points that this label masks. Will be a subset of the entire group’s points.

**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*) – `True` if the vectorized object contains `nan` values.

**classmethod 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** (*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.

**classmethod 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, labels_to_masks, copy=True, skip_checks=False*)

Construct a *LabelledPointUndirectedGraph* from an edges array.

See *PointUndirectedGraph* for more information.

**Parameters**

- **points** (*((n_vertices, n_dims, ) ndarray]*) – The array of point locations.

- **edges** (*((n_edges, 2, ) ndarray or None]*) – The *ndarray* of edges, i.e. all the pairs of vertices that are connected with an edge. If None, then an empty adjacency matrix is created.

- **labels_to_masks** (*ordereddict {str -> bool ndarray]*) – For each label, the mask that specifies the indices in to the points that belong to the label.

- **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.

**classmethod init_from_indices_mapping**(*points, adjacency, labels_to_indices, copy=True*)

Static constructor to create a *LabelledPointUndirectedGraph* from an ordered dictionary that maps a set of indices .

**Parameters**

- **points** (*PointCloud*) – The points representing the landmarks.

- **adjacency** (*((n_vertices, n_vertices, ) ndarray, csr_matrix or list of edges]*) – The adjacency matrix of the graph, or a list of edges representing adjacency.

- **labels_to_indices** (*ordereddict {str -> int ndarray]*) – For each label, the indices in to the points that belong to the label.

- **copy** (*boolean, optional*) – If True, a copy of the data is stored on the group.

**Returns** labelled_pointgraph (*LabelledPointUndirectedGraph*) – Labelled point undirected graph wrapping the given points 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 points.
- `ValueError` – If there exists any point in the points that is not covered by a label.

**class method `init_with_all_label`(points, adjacency_matrix, copy=True)**

Static constructor to create a `LabelledPointUndirectedGraph` with a single default `all` label that covers all points.

**Parameters**

- `points` (*ndarray*) – The points representing the landmarks.
- `adjacency_matrix` (*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 True, a copy of data is stored on the group.

**Returns** `labelled_pointgraph` (*LabelledPointUndirectedGraph*) – Labelled point-graph wrapping the given points with a single label called `all` that is True for all points.

**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` (*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.

**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)*.

**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.

**remove_label** (*label*)

Returns a new *LabelledPointUndirectedGraph* that does not contain the given label.

**Note:** You cannot delete a semantic label and leave the labelled point graph partially unlabelled. Labelled point graphs must contain labels for every point.

**Parameters** **label** (*string*) – The label to remove.

**Raises** **ValueError** – If deleting the label would leave some points unlabelled.

**tojson**()

Convert this *LabelledPointUndirectedGraph* to a dictionary JSON representation.

**Returns** **json** (*dict*) – Dictionary conforming to the LJSON v2 specification.
with_dims(dims)

Return a copy of this shape with only particular dimensions retained.

Parameters dims (valid numpy array slice) – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, [0, 1] could be provided or np.array([True, True, False]).

Returns copy of self, with only the requested dims

with_labels(labels)

A new labelled point undirected graph that contains only the given labels.

Parameters labels (str or list of str) – Label(s) that should be kept in the returned labelled point graph.

Returns labelled_pointgraph (LabelledPointUndirectedGraph) – A new labelled point undirected graph with the same group label but containing only the given label(s).

without_labels(labels)

A new labelled point undirected graph that excludes certain labels.

Parameters labels (str or list of str) – Label(s) that should be excluded in the returned labelled point graph.

Returns labelled_pointgraph (LabelledPointUndirectedGraph) – A new labelled point undirected graph with the same group label but containing all labels except the given label.

property edges

Returns the ndarray of edges, i.e. all the pairs of vertices that are connected with an edge.

Type (n_edges, 2, ) ndarray

property has_landmarks

Whether the object has landmarks.

Type bool

property labels

The list of labels that belong to this group.

Type list of str

property landmarks

The landmarks object.

Type LandmarkManager

property lms

Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the ‘underlying’ PointCloud subclass.

Type self

property n_dims

The number of dimensions in the pointcloud.

Type int

property n_edges

Returns the number of edges.

Type int
property n_labels
    Number of labels in the group.
    Type int

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

property n_landmarks
    The total number of points in the group.
    Type int

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

property n_points
    The number of points in the pointcloud.
    Type int

property n_vertices
    Returns the number of vertices.
    Type int

property vertices
    Returns the list of vertices.
    Type list

1.8.6 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 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 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 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 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 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.

stencil_grid

```
menpo.shape.stencil_grid(stencil, shape, dtype=None, format=None)
```

Construct a sparse matrix form a local matrix stencil

This function is useful for building sparse adjacency matrices according to a specific connectivity pattern.

This function is borrowed from the PyAMG project, under the permission of the MIT license:

The MIT License (MIT)

Copyright (c) 2008-2015 PyAMG Developers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The original version of this file can be found here:

https://github.com/pyamg/pyamg/blob/621d63411895898660e5ea078840118905bec061/pyamg/gallery/stencil.py

This file has been modified to fit the style standards of the Menpo project.

**Parameters**

- **S** (*ndarray*) – Matrix stencil stored in N-d array
- **grid** (*tuple*) – Tuple containing the N shape dimensions (shape)
• **dtype** *(np.dtype, optional)* – Numpy data type of the result
  
• **format** *(str, optional)* – Sparse matrix format to return, e.g. “csr”, “coo”, etc.

**Returns**  
**A** *(sparse matrix)* – Sparse matrix which represents the operator given by applying stencil  
estencil at each vertex of a regular shape with given dimensions.

**Notes**

The shape vertices are enumerated as `arange(prod(shape)).reshape(shape)`. This implies that the  
last shape dimension cycles fastest, while the first dimension cycles slowest. For example, if `shape=(2, 3)`  
then the shape vertices are ordered as `(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)`.

This coincides with the ordering used by the NumPy functions `ndenumerate()` and `mgrid()`.

**Raises**

• **ValueError** – If the stencil shape is not odd.

• **ValueError** – If the stencil dimension does not equal the number of shape dimensions

• **ValueError** – If the shape dimensions are not all positive

**Examples**

```python
>>> import numpy as np
>>> from menpo.shape import stencil_grid
>>> stencil = [[0,-1,0],[-1,4,-1],[0,-1,0]]  # 2D Poisson stencil
>>> shape = (3, 3)  # 2D shape with shape 3x3
>>> A = stencil_grid(stencil, shape, dtype=np.float, format='csr')
>>> A.todense()
matrix([[ 4., -1., 0., -1., 0., 0., 0., 0., 0.],
       [-1., 4., -1., 0., 0., 0., 0., 0., 0.],
       [ 0., -1., 4., 0., 0., -1., 0., 0., 0.],
       [-1., 0., 0., 4., -1., 0., -1., 0., 0.],
       [ 0., -1., 0., -1., 4., -1., 0., -1., 0.],
       [ 0., 0., -1., 0., -1., 4., 0., 0., -1.],
       [ 0., 0., 0., -1., 0., 0., 4., -1., 0.],
       [ 0., 0., 0., 0., -1., 0., -1., 4., -1.],
       [ 0., 0., 0., 0., 0., -1., 0., -1., 4.]])

>>> stencil = [[0,1,0],[1,0,1],[0,1,0]]  # 2D Lattice Connectivity
>>> shape = (3, 3)  # 2D shape with shape 3x3
>>> A = stencil_grid(stencil, shape, dtype=np.float, format='csr')
>>> A.todense()
matrix([[ 0., 1., 0., 1., 0., 0., 0., 0., 0.],
       [ 1., 0., 1., 0., 1., 0., 0., 0., 0.],
       [ 0., 1., 0., 0., 1., 0., 0., 0., 0.],
       [ 1., 0., 0., 0., 1., 0., 1., 0., 0.],
       [ 0., 1., 0., 1., 0., 0., 1., 0., 0.],
       [ 0., 0., 1., 0., 1., 0., 0., 0., 1.],
       [ 0., 0., 0., 1., 0., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 1., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 0., 1., 0., 1., 0.],
       [ 0., 0., 0., 0., 0., 0., 1., 0., 1.]])
```
1.8.7 Triangular Meshes

**TriMesh**

```python
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_edge_width=1.0, marker_edge_face_colour='k', marker_face_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', 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=(7, 7), label=None, **kwargs)`

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

```plaintext
[r, g, b, c, m, k, w]
or
(3, ) ndarray
```

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

```plaintext
[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

```plaintext
{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

```plaintext
{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

```plaintext
[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

```plaintext
{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
**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.

Returns \( \text{viewer} \) *(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_properties=['k', 'o--', 'r', 'g', 'c', 'm'], line_label='\textit{point graph viewer 2d}', line_color='k', line_style='-', line_width=2, render_markers=True, marker_properties=['k', 'o', 's', 'p', 'x', 'v', 'D'], marker_label='\textit{point graph viewer 2d}', marker_color='k', marker_style='s', marker_size=7, marker_face_color='k', marker_edge_color='k', marker_edge_width=1.0, render_lines_lms=True, line_properties_lms=['k', 'o--', 'r', 'g', 'c', 'm'], line_label_lms='\textit{landmark graph viewer 2d}', line_color_lms='k', line_style_lms='-', line_width_lms=1, render_markers_lms=True, marker_properties_lms=['k', 'o', 's', 'p', 'x', 'v', 'D'], marker_label_lms='\textit{landmark graph viewer 2d}', marker_color_lms='k', marker_style_lms='s', marker_size_lms=7, marker_face_color_lms='k', marker_edge_color_lms='k', marker_edge_width_lms=1.0, render_numbering=False, numbers_horizontal_align='center', numbers_vertical_align='bottom', numbers_font_name='sans-serif', numbers_font_size=10, numbers_font_weight='normal', numbers_font_color='k', render_legend=False, legend_title='\textit{point graph viewer 2d}', legend_font_name='sans-serif', legend_font_style='normal', legend_font_size=10, legend_font_weight='normal', legend_location=1, legend_bbox_to_anchor=(1.0, 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=(7, 7))

Visualize the landmarks. This method will appear on the TriMesh as view_landmarks.

### 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:

  ```python
  {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

  ```python
  {., ,, 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

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

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

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

- **marker_edge_width** (*float*, optional) – The width of the markers’ edge.
- **render_lines_lms** (*bool*, optional) – If True, the edges of the landmarks will be rendered.
- **line_colour_lms** *(See Below, optional)* – The colour of the lines of the landmarks. Example options:

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

- **line_style_lms** (*{-, --, -., :}*), optional) – The style of the lines of the landmarks.
- **line_width_lms** (*float*, optional) – The width of the lines of the landmarks.
- **render_markers** – If True, the markers of the landmarks will be rendered.
- **marker_style** – The style of the markers of the landmarks. Example options
• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks.
  Example options

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

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

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

• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **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:

  
<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>best</td>
<td>0</td>
</tr>
<tr>
<td>upper right</td>
<td>1</td>
</tr>
<tr>
<td>upper left</td>
<td>2</td>
</tr>
<tr>
<td>lower left</td>
<td>3</td>
</tr>
<tr>
<td>lower right</td>
<td>4</td>
</tr>
<tr>
<td>right</td>
<td>5</td>
</tr>
<tr>
<td>center left</td>
<td>6</td>
</tr>
<tr>
<td>center right</td>
<td>7</td>
</tr>
<tr>
<td>lower center</td>
<td>8</td>
</tr>
<tr>
<td>upper center</td>
<td>9</td>
</tr>
<tr>
<td>center</td>
<td>10</td>
</tr>
</tbody>
</table>

• **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 its 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. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

<table>
<thead>
<tr>
<th>0&lt;--3</th>
<th>^</th>
</tr>
</thead>
</table>

(continues on next page)
In the case of a pointcloud, the ordering will appear as:

```
 v  |
1-->2
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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. `distance_matrix[i, j]` is the distance between the `i`th point of this PointCloud and the `j`th point of the input PointCloud.

distance_matrix

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.

distance_to

distance_matrix

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 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.

Parameters mask ((n_points,) ndarray) – 1D array of booleans

Returns mesh (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.

Parameters mask ((n_tris,) ndarray) – 1D array of booleans

Returns mesh (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.

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 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.

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 `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 neighbour 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 neighbour 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

**with_dims(dims)**
Return a copy of this shape with only particular dimensions retained.

**Parameters**
* dims (valid numpy array slice) – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, [0, 1] could be provided or np.array([True, True, False]).

**Returns** copy of self, with only the requested dims

**property has_landmarks**
Whether the object has landmarks.

**Type** bool

**property landmarks**
The landmarks object.

**Type** LandmarkManager

**property lms**
Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the ‘underlying’ PointCloud subclass.

**Type** self

**property n_dims**
The number of dimensions in the pointcloud.

**Type** int

**property n_landmark_groups**
The number of landmark groups on this object.

**Type** int

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

**Type** int

**property n_points**
The number of points in the pointcloud.
Type  int

property 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.

Raises  ValueError – 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='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=(7, 7), label=None, **kwargs)

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:

```python
{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.
Menpo Documentation, Release 0.9.0

- **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

  

  \[
  (r, g, b, c, m, k, w) \\
  \text{or} \\
  (3, ) \text{ndarray}
  \]

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

  

  \[
  (r, g, b, c, m, k, w) \\
  \text{or} \\
  (3, ) \text{ndarray}
  \]

- **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

  

  \[
  \text{serif, sans-serif, cursive, fantasy, monospace}
  \]

- **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

  

  \[
  \text{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black}
  \]

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

  

  \[
  (r, g, b, c, m, k, w) \\
  \text{or} \\
  (3, ) \text{ndarray}
  \]

- **render_axes** (*bool*, optional) – If *True*, the axes will be rendered.

- **axes_font_name** (*See Below, optional*) – The font of the axes. Example options
• *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*), *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.

Returns `viewer` (*PointGraphViewer2d*) – The viewer object.

Raises `warning` – 2D Viewing of Coloured TriMeshes is not supported, automatically falls back to 2D `TriMesh` viewing.

```python
_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='k', line_style='-', line_width=2, render_markers=True, marker_style='s', marker_size=7, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_lines_lms=True, line_colour_lms=None, line_style_lms='-', line_width_lms=1, render_markers_lms=True, marker_style_lms='o', marker_size_lms=5, marker_face_colour_lms=None, marker_edge_colour_lms=None, marker_edge_width_lms=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', 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_ticks=None, axes_y_ticks=None, axes_x_limits=None, axes_y_limits=None, figure_size=(7, 7))
```

Visualize the landmarks. This method will appear on the `TriMesh` as `view_landmarks`.

1.8. menpo.shape
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_lines_lms** *(bool, optional)* – If True, the edges of the landmarks will be rendered.

- **line_colour_lms** *(See Below, optional)* – The colour of the lines of the landmarks. Example options:
• **line_style_lms**\(\{-, \--, \.-, :\}\), \(\text{optional}\) – The style of the lines of the landmarks.

• **line_width_lms** \((\text{float}), \text{optional}\) – The width of the lines of the landmarks.

• **render_markers** – If True, the markers of the landmarks will be rendered.

• **marker_style** – The style of the markers of the landmarks. Example options

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

• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options

\[
\{r, g, b, c, m, k, w\}
\]

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

\[
\{r, g, b, c, m, k, w\}
\]

• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **render_numbering** \((\text{bool}, \text{optional})\) – If True, the landmarks will be numbered.

• **numbers_horizontal_align** \({\text{center, right, left}}, \text{optional}\) – The horizontal alignment of the numbers’ texts.

• **numbers_vertical_align** \({\text{center, top, bottom, baseline}}, \text{optional}\) – The vertical alignment of the numbers’ texts.

• **numbers_font_name** \(\text{(See Below, optional)}\) – The font of the numbers. Example options

\[
\{\text{serif, sans-serif, cursive, fantasy, monospace}\}
\]

• **numbers_font_size** \((\text{int}, \text{optional})\) – The font size of the numbers.

• **numbers_font_style** \({\text{normal, italic, oblique}}, \text{optional}\) – The font style of the numbers.

• **numbers_font_weight** \(\text{(See Below, optional)}\) – The font weight of the numbers. Example options

\[
\{\text{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black}\}
\]

• **numbers_font_colour** \(\text{(See Below, optional)}\) – 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, 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)*
Converting 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. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
 v---|---2
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
 v---|---0
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

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.

`clip_texture (range=(0.0, 1.0))`

Method that returns a copy of the object with the coloured values clipped in range (0, 1).

Parameters `range (float, float, optional)` – The clipping range.

Returns `self (ColouredTriMesh)` – A copy of self with its texture clipped.
**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. \( \text{distance_matrix}[i, j] \) 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 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.

Parameters mask((n_points,) ndarray) – 1D array of booleans

Returns mesh (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.

Parameters mask((n_tris,) ndarray) – 1D array of booleans

Returns mesh (TriMesh) – A new mesh that has been masked by triangles.

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 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, 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.

**Returns** **trimesh** (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.

**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.

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

A copy of this mesh with colours linearly rescaled to fit a range.

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

**Returns** `coloured_mesh` (*type(self)*) – A copy of this mesh with colours linearly rescaled to fit in the range provided.

**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

**with_dims** *(dims)*
Return a copy of this shape with only particular dimensions retained.

**Parameters**
- **dims** *(valid numpy array slice)* – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, `[0, 1]` could be provided or `np.array([True, True, False])`.

**Returns**
- copy of self, with only the requested dims

**property has_landmarks**
Whether the object has landmarks.

**Type** `bool`

**property landmarks**
The landmarks object.

**Type** `LandmarkManager`

**property lms**
Deprecated. Maintained for compatibility, will be removed in a future version. Returns a copy of this object, which previously would have held the ‘underlying’ `PointCloud` subclass.

**Type** `self`

**property n_channels**
The number of channels of colour used (e.g. 3 for RGB).

**Type** `int`

**property n_dims**
The number of dimensions in the pointcloud.

**Type** `int`

**property n_landmark_groups**
The number of landmark groups on this object.

**Type** `int`

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

**Type** `int`

**property n_points**
The number of points in the pointcloud.

**Type** `int`

**property 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=(7, 7), label=None, **kwargs)
```

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

\[
\{r, g, b, c, m, k, w\} \\
\text{or} \\
(3, ) \text{ ndarray}
\]

• **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

\[
\{\text{serif, sans-serif, cursive, fantasy, monospace}\}
\]

• **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

\[
\{\text{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black}\}
\]

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

\[
\{r, g, b, c, m, k, w\} \\
\text{or} \\
(3, ) \text{ ndarray}
\]

• **render_axes** *(bool, optional)* – If True, the axes will be rendered.

• **axes_font_name** *(See Below, optional)* – The font of the axes. Example options

\[
\{\text{serif, sans-serif, cursive, fantasy, monospace}\}
\]

• **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

\[
\{\text{ultralight, light, normal, regular, book, medium, roman, semibold, demibold, demi, bold, heavy, extra bold, black}\}
\]
• **`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.

Returns **viewer** *(PointGraphViewer2d)* – The viewer object.

Raises **warning** – 2D Viewing of Coloured TriMeshes is not supported, automatically falls back to 2D TriMesh viewing.

```python
_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='k', line_style='-', line_width=2, render_markers=True, marker_style='s', marker_size=7, marker_face_colour='k', marker_edge_colour='k', marker_edge_width=1.0, render_lines_lms=True, line_colour_lms=None, line_style_lms='-', line_width_lms=1, render_markers_lms=True, marker_style_lms='o', marker_size_lms=5, marker_face_colour_lms=None, marker_edge_colour_lms=None, marker_edge_width_lms=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, render_xlabel=True, render_ylabel=True, render_lines=True, line_colour=None, line_style=None, line_width=2, render_markers=True, marker_style='s', marker_size=7, marker_face_colour=None, marker_edge_colour=None, marker_edge_width=1.0, render_numbering=False, numbers_horizontal_align=None, numbers_vertical_align=None, numbers_font_name=None, numbers_font_size=None, numbers_font_style=None, numbers_font_weight=None, numbers_font_colour=None, render_legend=False, legend_title=None, legend_font_name=None, legend_font_style=None, legend_font_size=None, legend_font_weight=None, legend_marker_scale=None, legend_location=None, legend_bbox_to_anchor=None, legend_border_axes_pad=None, legend_n_columns=None, legend_horizontal_spacing=None, legend_vertical_spacing=None, legend_border=None, legend_border_padding=None, legend_shadow=None, legend_rounded_corners=None, render_axes=None, render_xlabel=None, render_ylabel=None, render_lines=None, line_colour=None, line_style=None, line_width=None, render_markers=None, marker_style=None, marker_size=None, marker_face_colour=None, marker_edge_colour=None, marker_edge_width=None, render_numbering=None, numbers_horizontal_align=None, numbers_vertical_align=None, numbers_font_name=None, numbers_font_size=None, numbers_font_style=None, numbers_font_weight=None, numbers_font_colour=None, render_legend=None, legend_title=None, legend_font_name=None, legend_font_style=None, legend_font_size=None, legend_font_weight=None, legend_marker_scale=None, legend_location=None, legend_bbox_to_anchor=None, legend_border_axes_pad=None, legend_n_columns=None, legend_horizontal_spacing=None, legend_vertical_spacing=None, legend_border=None, legend_border_padding=None, legend_shadow=None, legend_rounded_corners=None)
```

Visualize the landmarks. This method will appear on the TriMesh as `view_landmarks`.

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_lines_lms** (*bool, optional*) – If True, the edges of the landmarks will be rendered.

• **line_colour_lms** (*See Below, optional*) – The colour of the lines of the landmarks. Example options:

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

• **line_style_lms** (*{-, --, -., :}, optional*) – The style of the lines of the landmarks.

• **line_width_lms** (*float, optional*) – The width of the lines of the landmarks.

• **render_markers** – If True, the markers of the landmarks will be rendered.

• **marker_style** – The style of the markers of the landmarks. Example options
• **marker_size** – The size of the markers of the landmarks in points.

• **marker_face_colour** – The face (filling) colour of the markers of the landmarks. Example options

```
{r, g, b, c, m, k, w}
```

or

```
(3, ) ndarray
```

• **marker_edge_colour** – The edge colour of the markers of the landmarks. Example options

```
{r, g, b, c, m, k, w}
```

or

```
(3, ) ndarray
```

• **marker_edge_width** – The width of the markers’ edge of the landmarks.

• **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:

<table>
<thead>
<tr>
<th>Legend Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>best</td>
<td>0</td>
</tr>
<tr>
<td>upper right</td>
<td>1</td>
</tr>
<tr>
<td>upper left</td>
<td>2</td>
</tr>
<tr>
<td>lower left</td>
<td>3</td>
</tr>
<tr>
<td>lower right</td>
<td>4</td>
</tr>
<tr>
<td>right</td>
<td>5</td>
</tr>
<tr>
<td>center left</td>
<td>6</td>
</tr>
<tr>
<td>center right</td>
<td>7</td>
</tr>
<tr>
<td>lower center</td>
<td>8</td>
</tr>
<tr>
<td>upper center</td>
<td>9</td>
</tr>
<tr>
<td>center</td>
<td>10</td>
</tr>
</tbody>
</table>

• **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, 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** ()

Returns a bounding box from two corner points as a directed graph. In the case of a 2D pointcloud, first point (0) should be nearest the origin. In the case of an image, this ordering would appear as:

```
0<--3
|   ^
|   |
```

(continues on next page)
In the case of a pointcloud, the ordering will appear as:

```
|   |   |   |
| 2 | ^ | 3|
|   |   |   |
| 0 | ^ | 1
```

In the case of a 3D pointcloud, the first point (0) should be the near closest to the origin and the second point is the far opposite corner.

**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.

**clip_texture** *(range=(0.0, 1.0))*

Method that returns a copy of the object with the texture values clipped in range (0, 1).

**Parameters**

- **range** *(float, float), optional* – The clipping range.

**Returns**

- **self** *(ColouredTriMesh)* – A copy of self with its texture clipped.

**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**

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**

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**

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**

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.

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

A copy of this mesh with texture linearly rescaled to fit a range.

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

**Returns**
- `textured_mesh` (*type(self)*) – A copy of this mesh with texture linearly rescaled to fit in the range provided.
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 LJJSON 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` \((n_{\text{tris}} \times 3, )\ \text{ndarray}\) – 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` \((n_{\text{unique edges}}, n_{\text{dims}})\ \text{ndarray}\) – 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` \((n_{\text{points}}, 3)\ \text{ndarray}\) – Normal at each point.

Raises `ValueError` – If mesh is not 3D

`with_dims` \((\text{dims})\)

Return a copy of this shape with only particular dimensions retained.

Parameters `dims` \((\text{valid numpy array slice})\) – The slice that will be used on the dimensionality axis of the shape under transform. For example, to go from a 3D shape to a 2D one, \([0, 1]\) could be provided or `np.array([True, True, False])`.

Returns `copy of self, with only the requested dims`
property n_points
The number of points in the pointcloud.
Type int

property n_tris
The number of triangles in the triangle list.
Type int

1.8.8 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.

1.8.9 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
```

In the case of a pointcloud, the ordering will appear as:

```
3<--2
|   ^
|   |
v--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.

**bounding_cuboid**

*menpo.shape.bounding_cuboid*(near_closest_to_origin, far_opposite_corner)

Return a bounding cuboid from the near closest and far opposite corners as a directed graph.

**Parameters**

- **near_closest_to_origin** *((float, float, float))* – Three floats representing the coordinates of the near corner closest to the origin.
- **far_opposite_corner** *((float, float, float))* – Three floats representing the coordinates of the far opposite corner compared to near_closest_to_origin.

**Returns bounding_box** *(PointDirectedGraph)* – The axis aligned bounding cuboid from the two given corners.

### 1.9 menpo.transform

#### 1.9.1 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.

**Returns transform** *(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.

**Returns transform** *(Homogeneous)* – A homogeneous transform that implements the scaling.
shear_about_centre

menpo.transform.shear_about_centre(obj, phi, psi, degrees=True)

Return an affine transform that implements shearing (distorting) 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.
- **phi** (*float*) – The angle of shearing in the X direction.
- **psi** (*float*) – The angle of shearing in the Y direction.
- **degrees** (*bool*, optional) – If True, then phi and psi are interpreted as degrees. If False they are interpreted as radians.

Returns **transform** (*Affine*) – An affine transform that implements the shearing.

Raises **ValueError** – Shearing can only be applied on 2D objects

transform_about_centre

menpo.transform.transform_about_centre(obj, transform)

Return a Transform that implements transforming an object about its centre. The given object must be transformable and must implement a method to provide the object centre. More precisely, the object will be translated to the origin (according to it’s centre), transformed, and then translated back to it’s previous position.

Parameters

- **obj** (*Transformable*) – A transformable object that has the `centre` method.
- **transform** (*ComposableTransform*) – A composable transform.

Returns **transform** (*Homogeneous*) – A homogeneous transform that implements the scaling.

1.9.2 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 \((x, batch\_size=\text{None}, **\text{kwargs})\)

Applies this transform to \(x\).

If \(x\) is \text{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 \text{ndarray}. The transformation will be non-destructive, returning the transformed version.

Any \text{kwargs} will be passed to the specific transform \text{\_apply()} method.

 Parameters

- \text{x} (\text{Transformable or (n\_points, n\_dims) ndarray}) – The array or object to be transformed.
- **batch\_size** (\text{int}, optional) – If not \text{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** (\text{dict}) – Passed through to \text{\_apply()}.

 Returns transformed \((\text{type}(x))\) – The transformed object or array

apply\_inplace \(*\text{args}, **\text{kwargs}\)

Depreciated as public supported API, use the non-mutating apply() instead.

For internal performance-specific uses, see \text{\_apply\_inplace()}.

as\_vector \((**\text{kwargs})\)

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

 Returns vector \((\text{\text{N,}}) \text{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 \((\text{transform})\)

A \text{Transform} that represents this transform composed after the given transform:

\[
c = a.\text{compose\_after}(b)\]
\[
c.\text{apply}(p) == a.\text{apply}(b.\text{apply}(p))\]

\(a\) and \(b\) are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, \(\circ\).

An attempt is made to perform native composition, but will fall back to a \text{TransformChain} as a last resort. See \text{composes\_with} for a description of how the mode of composition is decided.

 Parameters transform \((\text{Transform})\) – Transform to be applied before \text{self}

 Returns transform \((\text{Transform or TransformChain})\) – If the composition was native, a single new \text{Transform} will be returned. If not, a \text{TransformChain} is returned instead.

compose\_after\_from\_vector\_inplace \((\text{vector})\)

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 \((\text{\text{n\_parameters, \text{ndarray})})\) – Vector to update the transform state with.

compose\_after\_inplace \((\text{transform})\)

Update \text{self} so that it represents this transform composed after the given transform:
Menpo Documentation, Release 0.9.0

```python
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:

```python
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:

```python
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 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 NotImplemented ValueError – If h_matrix_is_mutable returns False.

property composes_inplace_with
    Homogeneous can swallow composition with any other Homogeneous, subclasses will have to over-
    ride and be more specific.

property composes_with
    Any Homogeneous can compose with any other Homogeneous.
property h_matrix
  The homogeneous matrix defining this transform.
  Type (n_dims + 1, n_dims + 1) ndarray

property 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

property has_true_inverse
  The pseudoinverse is an exact inverse.
  Type True

property n_dims
  The dimensionality of the data the transform operates on.
  Type int

property n_dims_output
  The output of the data from the transform.
  Type int

property 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_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.

**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

\[
\text{reduce}(\lambda x, y: x.chain(y), self.decompose()) == self
\]

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_from_2d_shear\((phi, psi, degrees=True)\)**
Convenience constructor for 2D shear transformations about the origin.

**Parameters**

- **phi** \((float)\) – The angle of shearing in the X direction.
- **psi** \((float)\) – The angle of shearing in the Y direction.
- **degrees** \((bool, optional)\) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

**Returns** shear_transform \((Affine)\) – A 2D shear transform.

**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:

```python
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 - 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** NotImplementError – If h_matrix_is_mutable returns False.
property composes_inplace_with
    Affine can swallow composition with any other Affine.

property composes_with
    Any Homogeneous can compose with any other Homogeneous.

property h_matrix
    The homogeneous matrix defining this transform.
    
    Type (n_dims + 1, n_dims + 1) ndarray

property 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

property has_true_inverse
    The pseudoinverse is an exact inverse.
    
    Type True

property linear_component
    The linear component of this affine transform.
    
    Type (n_dims, n_dims) ndarray

property n_dims
    The dimensionality of the data the transform operates on.
    
    Type int

property n_dims_output
    The output of the data from the transform.
    
    Type int

property 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]
```

property 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()`.

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:

```python
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, \( \circ \).

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_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.

`compose_after_inplace (transform)`
Update `self` so that it represents this transform composed after the given transform:

```python
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:

```python
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:

```python
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

```python
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_from_2d_shear(phi, psi, degrees=True)**
Convenience constructor for 2D shear transformations about the origin.

**Parameters**

• phi (float) – The angle of shearing in the X direction.

• psi (float) – The angle of shearing in the Y direction.

• degrees (bool, optional) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

**Returns** shear_transform (Affine) – A 2D shear transform.

**classmethod init_identity(n_dims)**
Creates an identity transform.

**Parameters**

n_dims (int) – The number of dimensions.

**Returns** identity (Similarity) – The identity matrix transform.

**pseudo_inverse()**
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:

```python
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

<table>
<thead>
<tr>
<th>Deprecation</th>
<th>The new homogeneous matrix to set.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value</strong></td>
<td><em>ndarray</em></td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td><em>bool</em>, optional – If <em>False</em>, do not copy the <em>h_matrix</em>. Useful for performance.</td>
</tr>
<tr>
<td><strong>skip_checks</strong></td>
<td><em>bool</em>, optional – If <em>True</em>, skip checking. Useful for performance.</td>
</tr>
</tbody>
</table>

| Raises | **NotImplementedError** – If *h_matrix_is_mutable* returns *False*. |

**property**  **composes_inplace_with**

*Affine* can swallow composition with any other *Affine*.

**property**  **composes_with**

Any Homogeneous can compose with any other Homogeneous.

**property**  **h_matrix**
The homogeneous matrix defining this transform.

| Type | *(n_dims + 1, n_dims + 1)* *ndarray* |

**property**  **h_matrix_is_mutable**

<table>
<thead>
<tr>
<th>Deprecation</th>
<th>True if <em>set_h_matrix()</em> is permitted on this type of transform.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>h_matrix</strong></td>
<td><em>ndarray</em></td>
</tr>
</tbody>
</table>

| Raises | **NotImplementedError** – If *h_matrix_is_mutable* returns *False*. |

**property**  **has_true_inverse**
The pseudoinverse is an exact inverse.

| Type | True |

**property**  **linear_component**
The linear component of this affine transform.

| Type | *(n_dims, n_dims)* *ndarray* |

**property**  **n_dims**
The dimensionality of the data the transform operates on.

| Type | *int* |
property n_dims_output
The output of the data from the transform.

Type int

property n_parameters
Number of parameters of Similarity

2D Similarity - 4 parameters

\[
\begin{bmatrix}
(1 + a), -b, & tx \\
(b, & (1 + a), & ty)
\end{bmatrix}
\]

3D Similarity: Currently not supported

Returns n_parameters (int) – The transform parameters

Raises DimensionalityError, NotImplementedError – Only 2D transforms are supported.

property 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.
- kwags (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.

**as_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.

**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:

```python
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_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.

**compose_after_inplace**(transform)

Update self so that it represents this transform composed after the given transform:

```python
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:
```python
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:

```python
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**


### 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_3d_from_quaternion (q)`**
Convenience constructor for 3D rotations based on quaternion parameters.

**Parameters**
- `q` ((4,) `ndarray`) – The quaternion parameters.

**Returns**
- `rotation (Rotation)` – A 3D rotation transform.

**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_from_2d_shear (phi, psi, degrees=True)`**
Convenience constructor for 2D shear transformations about the origin.

**Parameters**
- `phi` (`float`) – The angle of shearing in the X direction.
- `psi` (`float`) – The angle of shearing in the Y direction.
- `degrees` (`bool`, optional) – If `True` `phi` and `psi` are interpreted as degrees. If `False`, `phi` and `psi` are interpreted as radians.

**Returns**
- `shear_transform (Affine)` – A 2D shear transform.

**classmethod `init_from_3d_ccw_angle_around_x (theta, degrees=True)`**
Convenience constructor for 3D CCW rotations around the x axis

**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 3D rotation transform.

**classmethod `init_from_3d_ccw_angle_around_y (theta, degrees=True)`**
Convenience constructor for 3D CCW rotations around the y axis

**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 3D rotation transform.

**classmethod `init_from_3d_ccw_angle_around_z (theta, degrees=True)`**
Convenience constructor for 3D CCW rotations around the z axis

**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 3D 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:
    ```python
    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 NotImplemented - 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.

property composes_inplace_with
    Rotation can swallow composition with any other Rotation.

property composes_with
    Any Homogeneous can compose with any other Homogeneous.

property h_matrix
    The homogeneous matrix defining this transform.
    Type (n_dims + 1, n_dims + 1) ndarray
**property h_matrix_is_mutable**

Depreciated 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*

**property has_true_inverse**

The pseudoinverse is an exact inverse.

*Type: True*

**property linear_component**

The linear component of this affine transform.

*Type: (n_dims, n_dims) ndarray*

**property n_dims**

The dimensionality of the data the transform operates on.

*Type: int*

**property n_dims_output**

The output of the data from the transform.

*Type: int*

**property n_parameters**

Number of parameters of Rotation. Only 3D rotations are currently supported.

*Returns n_parameters (int) – The transform parameters. Only 3D rotations are currently supported which are parametrized with quaternions.*

*Raisers DimensionalityError, NotImplementedError – Non-3D Rotations are not yet vectorizable*

**property rotation_matrix**

The rotation matrix.

*Type: (n_dims, n_dims) ndarray*

**property translation_component**

The translation component of this affine transform.

*Type: (n_dims,) ndarray*

---

**Translation**

```python
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 \( \text{ndarray} \). The transformation will be non-destructive, returning the transformed version.

Any \( \text{kwargs} \) will be passed to the specific transform \( \_\text{apply()} \) method.

**Parameters**

- \( \text{x} \) (\( \text{Transformable} \) or \( (n\_\text{points}, n\_\text{dims}) \) \( \text{ndarray} \)) – The array or object to be transformed.

- \( \text{batch\_size} \) (int, optional) – If not \( \text{None} \), 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.

- \( \text{kwargs} \) (dict) – Passed through to \( \_\text{apply()} \).

**Returns** transformed (\( \text{type}(x) \)) – The transformed object or array

**apply\_inplace** (*\text{args}, **\text{kwargs}*)

Deprecated as public supported API, use the non-mutating \( \text{apply()} \) instead.

For internal performance-specific uses, see \( \_\text{apply\_inplace()} \).

**as\_vector** (**\text{kwargs}**)

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

**Returns** vector ((\( N \)) \( \text{ndarray} \)) – 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** (**\text{transform}**)

A \( \text{Transform} \) that represents this transform composed \text{after} the given transform:

\[
\begin{align*}
c &= a.\text{compose\_after}(b) \\
c.\text{apply}(p) &= a.\text{apply}(b.\text{apply}(p))
\end{align*}
\]

\( a \) and \( b \) are left unchanged.

This corresponds to the usual mathematical formalism for the compose operator, \( \circ \).

An attempt is made to perform native composition, but will fall back to a \( \text{TransformChain} \) as a last resort. See \( \text{composes\_with} \) for a description of how the mode of composition is decided.

**Parameters** \( \text{transform} \) (\( \text{Transform} \)) – Transform to be applied \text{before} \( \text{self} \)

**Returns** transform (\( \text{Transform} \) or \( \text{TransformChain} \)) – If the composition was native, a single new \( \text{Transform} \) will be returned. If not, a \( \text{TransformChain} \) is returned instead.

**compose\_after\_from\_vector\_inplace** (**\text{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** \( \text{vector} \) ((\( n\_\text{parameters}, \)) \( \text{ndarray} \)) – Vector to update the transform state with.

**compose\_after\_inplace** (**\text{transform}**, \( \text{ndarray} \))

Update \( \text{self} \) so that it represents this transform composed \text{after} the given transform:

\[
\begin{align*}
a\_\text{orig} &= a.\text{copy()} \\
a.\text{compose\_after\_inplace}(b) \\
a.\text{apply}(p) &= a\_\text{orig}.\text{apply}(b.\text{apply}(p))
\end{align*}
\]

\( a \) 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:

```py
a = a.compose_before(b)
c = a.compose_before_inplace(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:

```py
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a.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_shear (phi, psi, degrees=True) Convenience constructor for 2D shear transformations about the origin.

Parameters

- • phi (float) – The angle of shearing in the X direction.
- • psi (float) – The angle of shearing in the Y direction.
- • degrees (bool, optional) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

Returns shear_transform (Affine) – A 2D shear transform.

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

classmethod pseudoinverse_vector (vector) The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```python
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 pseudo_inverse_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 NotImplemented – If h_matrix_is_mutable returns False.
property composes_inplace_with
   *Affine* can swallow composition with any other *Affine*.

property composes_with
   Any Homogeneous can compose with any other Homogeneous.

property h_matrix
   The homogeneous matrix defining this transform.
   
   Type (n_dims + 1, n_dims + 1) ndarray

property 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

property has_true_inverse
   The pseudoinverse is an exact inverse.
   
   Type True

property linear_component
   The linear component of this affine transform.
   
   Type (n_dims, n_dims) ndarray

property n_dims
   The dimensionality of the data the transform operates on.
   
   Type int

property n_dims_output
   The output of the data from the transform.
   
   Type int

property n_parameters
   
   Type int
   
   Type The number of parameters

property 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 produced 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.

Returns **scale** *(UniformScale or NonUniformScale)* – The correct type of scale

Raises **ValueError** – 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_().

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:
Menpo Documentation, Release 0.9.0

```python
c = a.compose_after(b)
c.apply(p) == a.apply(b.apply(p))
```

\[ a \text{ and } b \text{ are left unchanged.} \]

This corresponds to the usual mathematical formalism for the compose operator, \( \circ \).

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_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.

`compose_after_inplace` (*transform*)

Update `self` so that it represents this transform composed after the given transform:

```python
a_orig = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a_orig.apply(b.apply(p))
```

\[ a \text{ is permanently altered to be the result of the composition. } b \text{ is left unchanged.} \]

**Parameters**

- `transform` (*Transform*) – 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:

```python
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```

\[ a \text{ and } b \text{ are left unchanged.} \]

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:

```python
a_orig = a.copy()
a.compose_before_inplace(b)
a.apply(p) == b.apply(a_orig.apply(p))
```

\[ a \text{ is permanently altered to be the result of the composition. } b \text{ is left unchanged.} \]
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_shear (phi, psi, degrees=True)`  
Convenience constructor for 2D shear transformations about the origin.

Parameters

- `phi (float)` – The angle of shearing in the X direction.
- `psi (float)` – The angle of shearing in the Y direction.
- `degrees (bool, optional)` – If `True` `phi` and `psi` are interpreted as degrees. If `False`, `phi` and `psi` are interpreted as radians.

Returns `shear_transform (Affine)` – A 2D shear transform.

classmethod `init_identity (n_dims)`  
Creates an identity transform.

Parameters `n_dims (int)` – The number of dimensions.

Returns `identity (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:

```python
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 NotImplemetedError – If h_matrix_is_mutable returns False.

property composes_inplace_with

UniformScale can swallow composition with any other UniformScale.

property composes_with

Any Homogeneous can compose with any other Homogeneous.

property h_matrix

The homogeneous matrix defining this transform.

Type (n_dims + 1, n_dims + 1) ndarray

property 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 NotImplemented Error.

Type bool

property has_true_inverse

The pseudoinverse is an exact inverse.

Type True

property linear_component

The linear component of this affine transform.

Type (n_dims, n_dims) ndarray

property n_dims

The dimensionality of the data the transform operates on.
Type \textit{int}

\textbf{property n_dims_output}

The output of the data from the transform.

Type \textit{int}

\textbf{property n_parameters}

1

Type \textit{int}

Type The number of parameters

\textbf{property scale}

The single scale value.

Type \textit{float}

\textbf{property translation_component}

The translation component of this affine transform.

Type \((n\text{\_dims,})\) \textit{ndarray}

\textbf{NonUniformScale}

\texttt{class menpo.transform.NonUniformScale(scale, skip\_checks=False)}

\texttt{Bases: DiscreteAffine, Affine}

An n\_dims scale transform, with a scale component for each dimension.

Parameters

- \texttt{scale} \((n\text{\_dims,})\) \textit{ndarray} – A scale for each axis.
- \texttt{skip\_checks} \(\text{bool}, \text{optional}\) – If True avoid sanity checks on \textit{h\_matrix} for performance.

\textbf{apply} \((x, \text{batch\_size=}\text{None}, **\text{kwargs})\)

Applies this transform to \(x\).

If \(x\) is \texttt{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 \textit{ndarray}. The transformation will be non-destructive, returning the transformed version.

Any \texttt{kwargs} will be passed to the specific transform \texttt{\_apply()} method.

Parameters

- \texttt{x (Transformable or (n\_points, n\_dims) ndarray)} – The array or object to be transformed.
- \texttt{batch\_size (int, optional)} – If not \texttt{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.
- \texttt{kwargs (dict)} – Passed through to \texttt{\_apply()}.

Returns transformed \((\text{type}(x))\) – 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_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.

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:

```python
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_shear (phi, psi, degrees=True)
Convenience constructor for 2D shear transformations about the origin.

Parameters

- phi (float) – The angle of shearing in the X direction.
- psi (float) – The angle of shearing in the Y direction.
• **degrees** (*bool*, optional) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

**Returns** **shear_transform** (**Affine**) – A 2D shear transform.

**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:

```python
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.

**property** **composes_inplace_with**  
**NonUniformScale** can swallow composition with any other **NonUniformScale** and **UniformScale**.

**property** **composes_with**  
Any Homogeneous can compose with any other Homogeneous.

**property** **h_matrix**  
The homogeneous matrix defining this transform.

**Type** ((n_dims + 1, n_dims + 1) *ndarray*)

**property** **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**
property has_true_inverse
  The pseudoinverse is an exact inverse.
  Type True

property linear_component
  The linear component of this affine transform.
  Type (n_dims, n_dims) ndarray

property n_dims
  The dimensionality of the data the transform operates on.
  Type int

property n_dims_output
  The output of the data from the transform.
  Type int

property n_parameters
  n_dims. They have the form [scale_x, scale_y, ...] representing the scale across each axis.
  Type list of int

property scale
  The scale vector.
  Type (n_dims,) ndarray

property translation_component
  The translation component of this affine transform.
  Type (n_dims,) ndarray

1.9.3 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 R2LogR2BF 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).

Raises ValueError – 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, nDims) 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

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.

property has_true_inverse
False

Type type

property n_dims
The number of dimensions of the target.

Type int

property 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

property n_points
The number of points on the target.

Type int

property source
The source PointCloud that is used in the alignment.

The source is not mutable.

Type PointCloud

property 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.

\[
(Ma)' = b' \\
(a' M') = b' \\
a a' M' = a b'
\]

$a a'$ 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

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 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:
   ```python
   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_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.

compose_after_inplace(transform)
   Update self so that it represents this transform composed after the given transform:
   ```python
   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:
   ```python
   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:

```python
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

```python
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** *(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** *(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_shear**(\(\phi, \psi, \text{degrees}=\text{True}\))

Convenience constructor for 2D shear transformations about the origin.

**Parameters**

- **\(\phi\) (float)** – The angle of shearing in the X direction.
- **\(\psi\) (float)** – The angle of shearing in the Y direction.
- **\text{degrees} (bool, optional)** – If \text{True} \(\phi\) and \(\psi\) are interpreted as degrees. If \text{False}, \(\phi\) and \(\psi\) are interpreted as radians.

**Returns** \text{shear_transform} (**Affine**) – A 2D shear transform.

**classmethod init_identity**(\(n_{\text{dims}}\))

Creates an identity matrix Affine transform.

**Parameters** \(n_{\text{dims}}\) (**int**) – The number of dimensions.

**Returns** \text{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** \text{transform} (**type(self)**) – The inverse of this transform.

**pseudoinverse_vector**(\(\text{vector}\))

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```python
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** \(\text{vector}\) (**(n_{\text{parameters}},) ndarray**) – A vectorized version of self

**Returns** \text{pseudoinverse_vector} (**(n_{\text{parameters}},) ndarray**) – The pseudoinverse of the vector provided

**set_h_matrix**(\(\text{value}, \text{copy}=\text{True}, \text{skip_checks}=\text{False}\))

Deprecated<br>Deprecated - do not use this method - you are better off just creating a new transform!

Updates \text{h_matrix}, optionally performing sanity checks.

Note that it won’t always be possible to manually specify the \text{h_matrix} through this method, specifically if changing the \text{h_matrix} could change the nature of the transform. See \text{h_matrix_is_mutable} for how you can discover if the \text{h_matrix} is allowed to be set for a given class.

**Parameters**

- **\text{value}** (**ndarray**) – The new homogeneous matrix to set.
- **\text{copy}** (**bool**, optional) – If \text{False}, do not copy the \text{h_matrix}. Useful for performance.

**Raises** \text{NotImplementedError} – If \text{h_matrix_is_mutable} returns \text{False}.

**set_target**(\(\text{new_target}\))

Update this object so that it attempts to recreate the \text{new_target}.

**Parameters** \text{new_target} (**PointCloud**) – The new target that this object should try and regenerate.
property composes_inplace_with

    Affine can swallow composition with any other Affine.

property composes_with

    Any Homogeneous can compose with any other Homogeneous.

property h_matrix

    The homogeneous matrix defining this transform.

    Type (n_dims + 1, n_dims + 1) ndarray

property 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

property has_true_inverse

    The pseudoinverse is an exact inverse.

    Type True

property linear_component

    The linear component of this affine transform.

    Type (n_dims, n_dims) ndarray

property n_dims

    The number of dimensions of the target.

    Type int

property n_dims_output

    The output of the data from the transform.

    Type int

property 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]
```

property n_points

    The number of points on the target.

    Type int
property source
The source :obj:`PointCloud` that is used in the alignment.

The source is not mutable.

Type :obj:`PointCloud`

property target
The current :obj:`PointCloud` that this object produces.

To change the target, use :meth:`set_target`.

Type :obj:`PointCloud`

property translation_component
The translation component of this affine transform.

Type :obj:`(n_dims,) ndarray`

AlignmentSimilarity

```
class menpo.transform.AlignmentSimilarity(source, target, rotation=True, allow_mirror=False)
```

Infers the similarity transform relating two vectors with the same dimensionality. This is simply the procrustes alignment of the :obj:`source` to the :obj:`target`.

Parameters

- **source** (:obj:`PointCloud`) – The source pointcloud instance used in the alignment
- **target** (:obj:`PointCloud`) – The target pointcloud instance used in the alignment
- **rotation** (bool, optional) – If :obj:`False`, the rotation component of the similarity transform is not inferred.
- **allow_mirror** (bool, optional) – If :obj:`True`, the Kabsch algorithm check is not performed, and mirroring of the Rotation matrix is permitted.

aligned_source()

The result of applying :obj:`self` to :obj:`source`

Type :obj:`PointCloud`

alignment_error()

The Frobenius Norm of the difference between the target and the aligned source.

Type :obj:`float`

```
apply (x, batch_size=None, **kwargs)
```

Applies this transform to :obj:`x`.

If :obj:`x` is :obj:`Transformable`, :obj:`x` will be handed this transform object to transform itself non-destructively (a transformed copy of the object will be returned).

If not, :obj:`x` is assumed to be an :obj:`ndarray`. The transformation will be non-destructive, returning the transformed version.

Any :obj:`kwargs` will be passed to the specific transform :meth:`_apply` method.

Parameters

- **x** (:obj:`Transformable` or (:obj:`n_points`, :obj:`n_dims`) :obj:`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*)

Deprecates 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:

```python
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, \( \circ \).

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_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.

**compose_after_inplace** (*transform*)

Update **self** so that it represents this transform composed after the given transform:

```python
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:

```python
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:

```python
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 (*shape(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 (*shape(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_shear (phi, psi, degrees=True)
Convenience constructor for 2D shear transformations about the origin.

Parameters

• phi (float) – The angle of shearing in the X direction.
• psi (float) – The angle of shearing in the Y direction.
• degrees (bool, optional) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

Returns shear_transform (Affine) – A 2D shear transform.

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.

Returns transform (type(self)) – The inverse of this transform.

pseudoinverse_vector (vector)
The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```python
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 NotImplementedeError – 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.

property composes_inplace_with
Affine can swallow composition with any other Affine.

property composes_with
Any Homogeneous can compose with any other Homogeneous.

property h_matrix
The homogeneous matrix defining this transform.

Type (n_dims + 1, n_dims + 1) ndarray

property 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

property has_true_inverse
The pseudoinverse is an exact inverse.

Type True

property linear_component
The linear component of this affine transform.

Type (n_dims, n_dims) ndarray

property n_dims
The number of dimensions of the target.

Type int

property n_dims_output
The output of the data from the transform.

Type int

property n_parameters
Number of parameters of Similarity

2D Similarity - 4 parameters

\[
\begin{bmatrix}
(l + a), & -b, & tx \\
(b, & (l + a), & ty \\
\end{bmatrix}
\]

3D Similarity: Currently not supported

Returns n_parameters (int) – The transform parameters

Raises DimensionalityError, NotImplementedError – Only 2D transforms are supported.

property n_points
The number of points on the target.

Type int
property source
The source `PointCloud` that is used in the alignment.

The source is not mutable.

Type `PointCloud`

property target
The current `PointCloud` that this object produces.

To change the target, use `set_target()`.

Type `PointCloud`

property 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.

**as_vector** (**kwargs)

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

**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:

```python
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, \( \circ \).

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_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.

**compose_after_inplace** (transform)

Update self so that it represents this transform composed after the given transform:

```python
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:

```python
    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:

```python
    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 \textit{nan} values or not. This is particularly useful for objects with unknown values that have been mapped to \textit{nan} values.

Returns \textit{has\_nan\_values} (\textit{bool}) – If the vectorized object contains \textit{nan} values.

classmethod init_3d_from_quaternion(q)
Convenience constructor for 3D rotations based on quaternion parameters.

Parameters \textit{q} ((4,) \textit{ndarray}) – The quaternion parameters.

Returns \textit{rotation} (\textit{Rotation}) – A 3D rotation transform.

classmethod init_from_2d_ccw_angle(theta, degrees=False)
Convenience constructor for 2D CCW rotations about the origin.

Parameters

\begin{itemize}
  \item \textit{theta} (\textit{float}) – The angle of rotation about the origin
  \item \textit{degrees} (\textit{bool}, optional) – If \textit{True} theta is interpreted as a degree. If \textit{False}, theta is interpreted as radians.
\end{itemize}

Returns \textit{rotation} (\textit{Rotation}) – A 2D rotation transform.

classmethod init_from_2d_shear(phi, psi, degrees=False)
Convenience constructor for 2D shear transformations about the origin.

Parameters

\begin{itemize}
  \item \textit{phi} (\textit{float}) – The angle of shearing in the X direction.
  \item \textit{psi} (\textit{float}) – The angle of shearing in the Y direction.
  \item \textit{degrees} (\textit{bool}, optional) – If \textit{True} phi and psi are interpreted as degrees. If \textit{False}, phi and psi are interpreted as radians.
\end{itemize}

Returns \textit{shear\_transform} (\textit{Affine}) – A 2D shear transform.

classmethod init_from_3d_ccw_angle_around_x(theta, degrees=False)
Convenience constructor for 3D CCW rotations around the x axis

Parameters

\begin{itemize}
  \item \textit{theta} (\textit{float}) – The angle of rotation about the origin
  \item \textit{degrees} (\textit{bool}, optional) – If \textit{True} theta is interpreted as a degree. If \textit{False}, theta is interpreted as radians.
\end{itemize}

Returns \textit{rotation} (\textit{Rotation}) – A 3D rotation transform.

classmethod init_from_3d_ccw_angle_around_y(theta, degrees=False)
Convenience constructor for 3D CCW rotations around the y axis

Parameters

\begin{itemize}
  \item \textit{theta} (\textit{float}) – The angle of rotation about the origin
  \item \textit{degrees} (\textit{bool}, optional) – If \textit{True} theta is interpreted as a degree. If \textit{False}, theta is interpreted as radians.
\end{itemize}

Returns \textit{rotation} (\textit{Rotation}) – A 3D rotation transform.

classmethod init_from_3d_ccw_angle_around_z(theta, degrees=False)
Convenience constructor for 3D CCW rotations around the z axis

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 3D 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 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 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 - 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.

**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.

**property** `composes_inplace_with`

*Rotation* can swallow composition with any other *Rotation*.

**property** `composes_with`

Any Homogeneous can compose with any other Homogeneous.

**property** `h_matrix`

The homogeneous matrix defining this transform.

Type *(n_dims + 1, n_dims + 1)* ndarray

**property** `h_matrix_ismutable`

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

**property** `has_true_inverse`

The pseudoinverse is an exact inverse.

Type True

**property** `linear_component`

The linear component of this affine transform.

Type *(n_dims, n_dims)* ndarray

**property** `n_dims`

The number of dimensions of the *target*.

Type int

**property** `n_dims_output`

The output of the data from the transform.

Type int

**property** `n_parameters`

Number of parameters of Rotation. Only 3D rotations are currently supported.

Returns *n_parameters* (int) – The transform parameters. Only 3D rotations are currently supported which are parametrized with quaternions.

Raises *DimensionalityError*, *NotImplementedError* – Non-3D Rotations are not yet vectorizable

**property** `n_points`

The number of points on the *target*.

Type int

**property** `rotation_matrix`

The rotation matrix.

Type *(n_dims, n_dims)* ndarray

**property** `source`

The source *PointCloud* that is used in the alignment.

The source is not mutable.

Type *PointCloud*
property target
    The current PointCloud that this object produces.
    To change the target, use set_target().
    Type PointCloud

property 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 trans-
        formed 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:

```python
import numpy as np
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, \( \circ \).

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_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.

compose_after_inplace (transform)

Update self so that it represents this transform composed after the given transform:

```python
import numpy as np
a = a.copy()
a.compose_after_inplace(b)
a.apply(p) == a.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:

```python
import numpy as np
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:

```python
da_orig = a.copy()
da.compose_before_inplace(b)
da.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.

**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_shear** *(phi, psi, degrees=True)*
Convenience constructor for 2D shear transformations about the origin.

**Parameters**
- **phi** *(float)* – The angle of shearing in the X direction.
- **psi** *(float)* – The angle of shearing in the Y direction.
- **degrees** *(bool, optional)* – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

**Returns**
- **shear_transform** *(Affine)* – A 2D shear transform.
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 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 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.

property composes_inplace_with
    Affine can swallow composition with any other Affine.

property composes_with
    Any Homogeneous can compose with any other Homogeneous.

property h_matrix
    The homogeneous matrix defining this transform.

    Type (n_dims + 1, n_dims + 1) ndarray

property 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`

**property** has_true_inverse

The pseudoinverse is an exact inverse.

Type `True`

**property** linear_component

The linear component of this affine transform.

Type `(n_dims, n_dims) ndarray`

**property** n_dims

The number of dimensions of the `target`.

Type `int`

**property** n_dims_output

The output of the data from the transform.

Type `int`

**property** n_parameters

`n_dims`

Type `int`

Type The number of parameters

**property** n_points

The number of points on the `target`.

Type `int`

**property** source

The source `PointCloud` that is used in the alignment.

The source is not mutable.

Type `PointCloud`

**property** target

The current `PointCloud` that this object produces.

To change the target, use `set_target()`.

Type `PointCloud`

**property** 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().

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 uniform scale without it’s alignment nature.

Returns transform (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.

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_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.

compose_after_inplace(transform)
Update self so that it represents this transform composed after the given transform:

```python
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:

```python
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:

```python
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.

Returns

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

classmethod init_from_2d_shear (phi, psi, degrees=True)

Convenience constructor for 2D shear transformations about the origin.

Parameters

• phi (float) – The angle of shearing in the X direction.
• psi (float) – The angle of shearing in the Y direction.
• degrees (bool, optional) – If True phi and psi are interpreted as degrees. If False, phi and psi are interpreted as radians.

Returns

shear_transform (Affine) – A 2D shear transform.

classmethod init_identity (n_dims)

Creates an identity transform.

Parameters

n_dims (int) – The number of dimensions.

Returns

identity (UniformScale) – The identity matrix transform.

classmethod 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.

classmethod pseudoinverse_vector (vector)

The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

```python
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.

**property composes_inplace_with**

UniformScale can swallow composition with any other UniformScale.

**property composes_with**

Any Homogeneous can compose with any other Homogeneous.

**property h_matrix**

The homogeneous matrix defining this transform.

**Type** *(n_dims + 1, n_dims + 1) ndarray*

**property 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*

**property has_true_inverse**

The pseudoinverse is an exact inverse.

**Type** *True*

**property linear_component**

The linear component of this affine transform.

**Type** *(n_dims, n_dims) ndarray*

**property n_dims**

The number of dimensions of the target.

**Type** *int*

**property n_dims_output**

The output of the data from the transform.

**Type** *int*

**property n_parameters**

1

**Type** *int*
Type  The number of parameters

**property n_points**
The number of points on the `target`.

Type  `int`

**property scale**
The single scale value.

Type  `float`

**property source**
The source `PointCloud` that is used in the alignment.

The source is not mutable.

Type  `PointCloud`

**property target**
The current `PointCloud` that this object produces.

To change the target, use `set_target()`.

Type  `PointCloud`

**property translation_component**
The translation component of this affine transform.

Type  `(n_dims,) ndarray`

### 1.9.4 Group Alignments

**GeneralizedProcrustesAnalysis**

```python
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

**meanAlignedShape**

Returns the mean of the aligned shapes.

Type  `PointCloud`

**meanAlignmentError**

Returns the average error of the recursive procrustes alignment.

Type  `float`
1.9.5 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().

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)
    A Transform that represents this transform composed after the given transform:

    ```python
    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:

```python
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:

```python
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:

```python
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

**property** *composes_inplace_with*

The *Transforms* 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 *Transforms*
**property** **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

**property** **n_dims**

The dimensionality of the data the transform operates on.

None if the transform is not dimension specific.

*Type* *int* or *None*

**property** **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*

### 1.9.6 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:** $r = \|x - c\|$

**Parameters**

- **c** ($(n_centres, n_dims) ndarray$) – The set of centers that make the basis. Usually represents a set of source landmarks.

**apply** $(x, \text{batch_size=None, **kwargs})$

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

**property n_centres**

The number of centres.

**Type**

- **int**

**property n_dims**

The RBF can only be applied on points with the same dimensionality as the centres.

**Type**

- **int**

**property 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:

```python
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:

```python
c = a.compose_before(b)
c.apply(p) == b.apply(a.apply(p))
```
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

**property n_centres**

- The number of centres.

  **Type** int

**property n_dims**

- The RBF can only be applied on points with the same dimensionality as the centres.

  **Type** int

**property n_dims_output**

- The result of the transform has a dimension (weight) for every centre.

  **Type** int

### 1.9.7 Abstract Bases

**Transform**

```python
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:

```python
(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
```

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()`.

**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))
```

**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:
```python
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

**property n_dims**
The dimensionality of the data the transform operates on.

None if the transform is not dimension specific.

**Type**
- `int` or `None`

**property 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 in-place.

**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.
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.

_def_compose_after_inplace_(transform)

Specialised inplace composition. This should be overridden to provide specific cases of composition as defined in composes_inplace_with.

Parameters:

transform (composes_inplace_with) – Transform to be applied before self

_def_compose_before_inplace_(transform)

Specialised inplace composition. This should be overridden to provide specific cases of composition as defined in composes_inplace_with.

Parameters:

transform (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().

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)

A Transform that represents this transform composed after the given transform:

```python
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 \texttt{TransformChain} as a last resort. See \texttt{composes_with} for a description of how the mode of composition is decided.

**Parameters** \texttt{transform (Transform)} – Transform to be applied \texttt{before} \texttt{self}

**Returns** \texttt{transform (Transform or TransformChain)} – If the composition was native, a single new \texttt{Transform} will be returned. If not, a \texttt{TransformChain} is returned instead.

\textbf{compose_after_inplace} (\texttt{transform})

Update \texttt{self} so that it represents \texttt{this} transform composed \texttt{after} the given transform:

\begin{verbatim}
  a_orig = a.copy()
  a.compose_after_inplace(b)
  a.apply(p) == a_orig.apply(b.apply(p))
\end{verbatim}

\texttt{a} is permanently altered to be the result of the composition. \texttt{b} is left unchanged.

**Parameters** \texttt{transform (composes_inplace_with)} – Transform to be applied \texttt{before} \texttt{self}

**Raises** \texttt{ValueError} – If \texttt{transform} isn’t an instance of \texttt{composes_inplace_with}

\textbf{compose_before} (\texttt{transform})

A \texttt{Transform} that represents \texttt{this} transform composed \texttt{before} the given transform:

\begin{verbatim}
  c = a.compose_before(b)
  c.apply(p) == b.apply(a.apply(p))
\end{verbatim}

\texttt{a} and \texttt{b} are left unchanged.

An attempt is made to perform native composition, but will fall back to a \texttt{TransformChain} as a last resort. See \texttt{composes_with} for a description of how the mode of composition is decided.

**Parameters** \texttt{transform (Transform)} – Transform to be applied \texttt{after} \texttt{self}

**Returns** \texttt{transform (Transform or TransformChain)} – If the composition was native, a single new \texttt{Transform} will be returned. If not, a \texttt{TransformChain} is returned instead.

\textbf{compose_before_inplace} (\texttt{transform})

Update \texttt{self} so that it represents \texttt{this} transform composed \texttt{before} the given transform:

\begin{verbatim}
  a_orig = a.copy()
  a.compose_before_inplace(b)
  a.apply(p) == b.apply(a_orig.apply(p))
\end{verbatim}

\texttt{a} is permanently altered to be the result of the composition. \texttt{b} is left unchanged.

**Parameters** \texttt{transform (composes_inplace_with)} – Transform to be applied \texttt{after} \texttt{self}

**Raises** \texttt{ValueError} – If \texttt{transform} isn’t an instance of \texttt{composes_inplace_with}

\textbf{copy} ()

Generate an efficient copy of this object.

Note that Numpy arrays and other \texttt{Copyable} objects on \texttt{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** \texttt{type(self)} – A copy of this object
property 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 Transforms

property 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 Transforms

property n_dims
   The dimensionality of the data the transform operates on.

   None if the transform is not dimension specific.

   Type int or None

property 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 pseudo-inverse 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)

property 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.

**property n_dims**

The number of dimensions of the `target`.

Type `int`

**property n_points**

The number of points on the `target`.

Type `int`

**property source**

The source `PointCloud` that is used in the alignment.

The source is not mutable.

Type `PointCloud`

**property target**

The current `PointCloud` that this object produces.

To change the target, use `set_target()`.

Type `PointCloud`
**MultipleAlignment**

```python
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**

```python
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.

```python
def decompose()
    A `DiscreteAffine` is already maximally decomposed - return a copy of self in a list.

    Returns transform (`DiscreteAffine`) – Deep copy of self.
```

### 1.9.8 Performance Specializations

Mix-ins that provide fast vectorized variants of methods.

**VComposable**

```python
class menpo.transform.base.composable.VComposable
Bases: object
```

Mix-in for `VectorizableComposableTransforms`.

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.

```python
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**

```python
class menpo.transform.base.invertible.VInvertible
Bases: Invertible
```

Mix-in for `VectorizableInvertibleTransforms`.

### 1.9. menpo.transform
Prefer this mix-in over \texttt{Invertible} if the \texttt{Transform} in question is \texttt{Vectorizable} as this adds \texttt{from\_vector()} variants to the \texttt{Invertible} interface. These can be tuned for performance, and are, for instance, needed by some of the machinery of fit.

\texttt{pseudoinverse()} \\
\hspace{10mm} The pseudoinverse of the transform - that is, the transform that results from swapping \texttt{source} and \texttt{target}, or more formally, negating the transforms parameters. If the transform has a true inverse this is returned instead.

\hspace{10mm} \textbf{Type} \hspace{1mm} \texttt{type(self)}

\texttt{pseudoinverse\_vector(vector)} \\
\hspace{10mm} The vectorized pseudoinverse of a provided vector instance. Syntactic sugar for:

\begin{quote}
\texttt{self.from\_vector(vector).pseudoinverse().as\_vector()}
\end{quote}

Can be much faster than the explicit call as object creation can be entirely avoided in some cases.

\hspace{10mm} \textbf{Parameters} \hspace{1mm} \texttt{vector((n\_parameters,)) ndarray} – A vectorized version of \texttt{self}

\hspace{10mm} \textbf{Returns} \hspace{1mm} \texttt{pseudoinverse\_vector((n\_parameters,)) ndarray} – The pseudoinverse of the vector provided

\textbf{property} \hspace{1mm} \texttt{has\_true\_inverse}

\hspace{10mm} \textbf{True} if the pseudoinverse is an exact inverse.

\hspace{10mm} \textbf{Type} \hspace{1mm} \texttt{bool}

1.10 \texttt{menpo.visualize}

1.10.1 Abstract Classes

\textbf{Renderer}

\begin{verbatim}
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 \texttt{figure\_id} or a \texttt{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 \texttt{figure\_id} and \texttt{new\_figure == True} is not a valid state.

  \textbf{Parameters}

  \hspace{10mm} • \texttt{figure\_id (object)} – A figure id. Could be any valid object that identifies a figure in a given framework (\textit{str}, \textit{int}, \textit{float}, etc.).

  \hspace{10mm} • \texttt{new\_figure (bool)} – Whether the rendering engine should create a new figure.

  \textbf{Raises} \hspace{3mm} \texttt{ValueError} – It is not valid to provide a figure id AND request a new figure to be rendered on.

  \texttt{clear\_figure()}

  Abstract method for clearing the current figure.
\end{verbatim}
force_draw()
Abstract method for forcing the current figure to render.

get_figure()
Abstract method for getting the correct figure to render on. Should also set the correct figure_id for the figure.

Returns figure (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.

Parameters

kwargs (dict) – Passed through to specific rendering engine.

Returns
viewer (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.

Parameters

kwargs (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.

clear_figure()
Method for clearing the current figure.

force_draw()
Method for forcing the current figure to render. This is useful for the widgets animation.
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

kwgs (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

```python
('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

```python
('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

```python
['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.

### 1.10.2 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, boxes_line_colour='r', boxes_line_style='-', boxes_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, 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=(7,7))
```

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

```python
{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:

```python
{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** (bool, optional) – If True, the edges will be rendered.

• **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

```python
{., ,, 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

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

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

```python
{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

```python
{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

```python
{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

```python
{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

```python
{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
• **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.

Returns viewer (ImageViewer) – The image viewing object.

### 1.10.3 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, min_seconds_between_updates=0.1)

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.
• **min_seconds_between_updates** (*float*, optional) – The number of seconds that have to pass between two print updates. This allows `print_progress` to be used on fast iterations without incurring a significant overhead. Set to 0 to disable this throttling.

**Raises** `ValueError` – 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:
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:

```python
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

**1.10.4 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, render_lines=True, line_colour=None, line_style='-', line_width=1, render_markers=False, 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, figure_size=(7, 7), 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:

```python
{'r', 'g', 'b', 'c', 'm', 'k', 'w'}
```

• **line_style (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 are:

```python
{'.', ',', 'o', 'v', '^', '<', '>', '+', 'x', 'D', 'd', 's', 'p', 's', 'h', 'H', 'l', '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

```python
{'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

```python
{'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

```python
{'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

```python
{'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:
• \textbf{legend_bbox_to_anchor} ((float, float), optional) – The bbox that the legend will be anchored.

• \textbf{legend_border_axes_pad} (float, optional) – The pad between the axes and legend border.

• \textbf{legend_n_columns} (int, optional) – The number of the legend’s columns.

• \textbf{legend_horizontal_spacing} (float, optional) – The spacing between the columns.

• \textbf{legend_vertical_spacing} (float, optional) – The vertical space between the legend entries.

• \textbf{legend_border} (bool, optional) – If True, a frame will be drawn around the legend.

• \textbf{legend_border_padding} (float, optional) – The fractional whitespace inside the legend border.

• \textbf{legend_shadow} (bool, optional) – If True, a shadow will be drawn behind legend.

• \textbf{legend_rounded_corners} (bool, optional) – If True, the frame’s corners will be rounded (fancybox).

• \textbf{render_axes} (bool, optional) – If True, the axes will be rendered.

• \textbf{axes_font_name} (See below, optional) – The font of the axes. Example options

\begin{verbatim}
{'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
\end{verbatim}

• \textbf{axes_font_size} (int, optional) – The font size of the axes.


• \textbf{axes_font_weight} (See below, optional) – The font weight of the axes. Example options

\begin{verbatim}
 ‘extra bold’, ‘black’}
\end{verbatim}

• \textbf{figure_size} ((float, float) or None, optional) – The size of the figure in inches.

• \textbf{render_grid} (bool, optional) – If True, the grid will be rendered.

• \textbf{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=(7, 7))

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’ point-cloud.

• `figure_size ((float, float) tuple or None optional)` – The size of the figure in inches.

## 1.11 Changelog

### 1.11.1 0.9.0 (2019/07/10)

Remove Python 2.x from the build matrix as it is to difficult to continue support as all major packages are sunsetting Python 2.x support in 2020. At this time no regressions have been added but Python 2.x compatibility will not be guaranteed from this release onwards. This release also adds Python 3.7 support and builds.

Furthermore, the build system has moved to using conda-forge as its package repository. This is in preparation for adding Menpo to conda-forge. Many requirements were updated in turn and this the major version bump.
This also adds support for the LJSONv3 format (https://github.com/menpo/landmarker.io/wiki/LEP-0001:-LJSON-v3)

Github Pull Requests

- #785 Migrate towards LJSONv3. (@grigorisg9gr)
- #802 minor .lms fix in test (@jabooth)
- #803 3D LabelledPointUndirectedGraph visualization (@nontas)
- #805 Update base.py (@georgesterpu)
- #807 Fall-back to querying numpy for inc dir (@jabooth)
- #811 Pytest and upgrade to Matplotlib 2.x (@patricksnape)
- #814 Bump deps, fix or disable problematic tests (@jabooth)
- #815 Pure-Python mesh normal calculation fallback (@jabooth)
- #817 remove (unused) Cython PWA (@jabooth)
- #829 Move to conda-forge, update dependencies and drop Python 2.7 from build config (@patricksnape)

1.11.2 0.8.1 (2017/05/06)

Menpo 0.8.1 includes a few new minor features, and Python 3.6 support.

Github Pull Requests

- #753 .view_widget() on LazyList (@jabooth)
- #777 convenience constructors for 3D rotations (@jabooth)

1.11.3 0.8.0 (2017/05/04)

Menpo 0.8.0 includes a variety of minor bug fixes and a few major features:

A large simplification is made to how Menpo handles landmarks. In the past, Landmarks were a special type in Menpo, they weren’t shapes. That meant we frequently had to access the underlying shape information (at .lms), which was always a little clunky and confusing, especially to newcomers.

In this release, we instead change the dynamic so that any shape can be attached as a landmark directly. If you only need to store a list of points with no groups or labels, you can now just use a PointCloud, which is totally natural in Menpo. The traditional features of LandmarkGroup (handling groups etc) are now available in a specialization Shape called LabelledPointUndirectedGraph.

The migration is simple - just remove .lms anywhere from your codebase (a warning will be raised when you do use .lms - this will be deprecated in the future).

A smaller set of additional features in 0.8:

1. Most menpo objects print a sensible string for their __repr__, which is helpful in the notebook.
2. you can now tab complete landmark keys in the notebook.
Github Pull Requests

- #766 upgrades for menpowidgets (ipywidgets 6) (@nontas)
- #798 move back to making macOS builds with travis (@patricksnape)
- #799 move back to making Win builds with appveyor (@patricksnape)
- #792 add throttling to print_progress for fast iterations (@jabooth)
- #790 fix __setstate__ for old landmarks (@jabooth)
- #787 add a default __str__ implementation to avoid inf recursion on __repr__ (@jabooth)
- #782 __repr__ return __str__ for all Copyable objects (@jabooth)
- #780 add support for ipython tab complete landmarks (@jabooth)
- #675 LandmarkGroups are now Shapes (@patricksnape, @jabooth, @nontas)
- #778 Minor documentation clarification in image rasterize utility (@grigorisg9gr)
- #761 Add clip_pixels to Image and automatically clip RGB visualisations (@grigorisg9gr)

1.11.4 0.7.7 (2017/01/05)

Minor bug fixes

Github Pull Requests

- #767 Minor fixes (@patricksnape)
- #774 Fix pip install by properly including source files (@patricksnape)
- #775 Allow Pillow 4.x (@patricksnape)
- #776 Manifest includes should be recursive (@jabooth)

1.11.5 0.7.6 (2016/12/10)

Minor bug fixes and three new pieces of functionality:

- 3D visualization improvements (@nontas)
- Bounding cubiod method for 3D shapes (@nontas)
- New transforms to change dimensionality of shapes (@jabooth)

Github Pull Requests

- #745 Documentation change in image gradient for int pixels dtype. (@grigorisg9gr)
- #750 Copy landmarks and path (@jabooth)
- #751 WithDims transform and with_dims method (@jabooth)
- #754 Restrict the video channels in exports(@grigorisg9gr)
- #755 fix ndarray slicing of lazylist, cleaner __add__ implementation (@jabooth)
- #756 remove duplicate import_pickles def (!) (@jabooth)
• #757 don’t print_dynamic size report in as_matrix (@jabooth)
• #759 Bounding cuboid (@nontas)
• #760 Quaternions for 3D Rotations (@nontas)
• #762 3D Visualization Upgrade (@nontas)
• #764 Fix NUMPY_INC_PATH detection for multiple dirs found (@jabooth)

1.11.6 0.7.5 (2016/11/17)

Minor bug fixes and three new pieces of functionality:

• Fix bug in video importing when importing long videos (closed pipe) (@patricksnape)
• Update MANIFEST.in to include the LICENSE.txt and AUTHORS.txt
• Add new transform_about_centre method on images. Allow easily performing operations such as rotating an image about it’s centre or shearing an image about it’s centre.
• Allow path only exporters (rather than hard requiring a buffer). The exporting logic was also updated to support multiple kwargs to be passed through to the exporters easily.
• New transforms to move between image and texture coordinates. (@jabooth).

Github Pull Requests

• #724 add transforms for tcoords -> image coords (@jabooth)
• #733 Remove user guide as it now lives at www.menpo.org (@nontas)
• #729 Fix PCA docs w.r.t centre parameter (@jabooth)
• #736 Fix bug importing boolean images with pillow (@grigorisg9gr)
• #739 Fix PointGraph printing to mention dimensionality (@nontas)
• #737 Allow path only exporters. Fix landmarking exporting on Python 3. (@patricksnape)
• #735 Fix init_from_channels_at_back to support 2D arrays. (@grigorisg9gr, @patricksnape)
• #738 Add transform_about_centre to images (@nontas, @patricksnape)
• #743 Set nan values to None in video importing (@JeanKossaifi)
• #744 Fix Regression: Allow None for landmark_resolver (@grigorisg9gr, @patricksnape)

1.11.7 0.7.4 (2016/08/18)

Minor fixes and additions including improved compatibility with loading older menpo PCAModel pickles and importing Python 2 pickles in Python 3.

Github Pull Requests

• #723 Add optional ‘encoding’ argument to import_pickle. (@patricksnape)
• #728 Allow for unpickling of the older PCAModel. (@patricksnape)
• #726 Improve pip install. (@patricksnape)
• #731 Pin setuptools to 23.x. (@jabooth)

1.11.8 0.7.3 (2016/08/05)

Minor fixes and additions including allowing more flexibility in FFmpeg exporting options, supporting more types with normal calculations, and tidying up the conda build recipe.

Github Pull Requests

• #716 allow for forced inexact frame count in FFMpegVideoReader. (@san-bil)
• #714 Utilise the kwargs in exporting video with ffmpeg. (@grigoris9gr)
• #720 Simplify the conda build. (@patricksnape)
• #719 Support more types for normals. (@patricksnape)

1.11.9 0.7.2 (2016/06/22)

Minor fixes including allowing exporting grayscale videos and fixing a minor bug in PointGraph masking. The Menpo logo has also been updated.

Github Pull Requests

• #709 Minor change in documentation of pickle, contrain_landmarks in image. (@grigoris9gr)
• #713 Remove zero edge adjacency check. (@patricksnape)
• #711 Enable greyscale video to be exported. (@grigoris9gr, @patricksnape)

1.11.10 0.7.1 (2016/06/10)

We now ship our own FFmpeg video importer based on piping, thus removing the dependency on imageio. A couple of further minor improvements were also introduced:

• Added register_* methods to the importing packages to make it simpler to add custom importers. For example, use menpo.io.register_image_importer('.extension', your_method) to register a new importer.
• Fix rasterization bug for matplotlib on Python 3.x
• normalise keyword arguments are now deprecated in favour of normalize to make spelling consistent across project.
• LazyList is now copyable -> LazyList.copy
• LazyList map method now accepts a list of callables as well as a single callable.
• Add LazyList.init_from_iterable for easily creating lazy lists from existing iterables.
• Fix small visualisation bug for viewing of LandmarkGroup that contain PointClouds
• New pixel_with_channels_at_back method for images
• Deprecate init_from_rolled_channels in favour of new method init_from_channels_at_back
Deprecate `as_imageio`

Finally, as of this release we no longer use Appveyor, in favour of our own Windows Jenkins build boxes.

Github Pull Requests

- #694 Functional IO Package. (@patricksnape)
- #703 Fix the bug with rasterize landmarks with matplotlib backend. (@grigorisg9gr)
- #700 Standardise the normalize spelling in importers. (@grigorisg9gr)
- #702 Now reading videos using subprocess and ffmpeg. Drop ImageIO. (@JeanKossaifi, @patricksnape)
- #706 Autoscale PointCloud if no limits set. (@patricksnape)
- #707 LazyList init methods and are now Copyable. (@patricksnape)
- #708 Remove appveyor in favour of Jenkins. (@patricksnape)

1.11.11 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:

```python
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

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

```python
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, `MaskedImage`es 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](#698) Video importing warnings. (@patricksnape)
- [#697](#697) Relex version constraints on dependencies. (@jabooth)
- [#695](#695) condaci fixes. (@patricksnape)
- [#692](#692) new **OverwriteError** raised specifically for overwrite errors in io.export. (@jabooth)
- [#691](#691) Add mio.pickle_paths(glob). (@jabooth)
- [#690](#690) Fix init_2d_grid for TriMesh subclasses + add init_from_depth_image. (@patricksnape)
- [#687](#687) WIP: BREAKING: Various release fixes. (@patricksnape)
- [#685](#685) GMRF mahalanobis computation with sparse precision. (@nontas)
- [#684](#684) Video importer docs and negative max_images. (@grigoris9gr)
- [#683](#683) Bugfix: Widget imports. (@nontas)
- [#682](#682) Update the view_patches to show only the selected landmarks. (@grigoris9gr)
- [#680](#680) Expose file extension to exporters (Fix PIL exporter bug). (@patricksnape)
- [#678](#678) Deprecate set_h_matrix and fix #677. (@patricksnape)
- [#676](#676) Implement LazyList __add__. (@patricksnape)
- [#673](#673) Fix the widgets in PCA. (@grigoris9gr)
- [#672](#672) Use Conda environment.yml on RTD. (@patricksnape)
- [#670](#670) Rasterize 2D Landmarks Method. (@patricksnape)
- [#669](#669) BREAKING: Add LazyList - default importing is now Lazy. (@patricksnape)
- [#668](#668) Speedup as_greyscale. (@patricksnape)
- [#666](#666) Add the protocol option in exporting pickle. (@grigoris9gr)
- [#665](#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)

1.11.12 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)

1.11.13 0.6.1 (2015/12/09)

Fix a nasty bug pertaining to a Diamond inheritance problem in PCA. Add the Gaussian 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)

1.11.14 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.
Menpo Documentation, Release 0.9.0

- 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.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)

1.11.15 0.5.3 (2015/08/12)

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

1.11.16 0.5.2 (2015/08/04)

Minor bug fixes and improvements 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)

1.11.17 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 cvylfeat (@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)

1.11.18 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:

```python
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. (@grigorisd9gr)
• #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 (@grigorisd9gr)
• #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 convinience method, quick lm ﬁlter (@jabooth)
• #554 Bugﬁx: visualize_images (@nontas)
• #553 Transform docs ﬁxes (@nontas)
• #533 LandmarkGroup.init_with_all_label, init_* convinience 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)

1.11.19 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)

1.11.20 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)

1.11.21 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)

1.11.22 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)

1.11.23 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. (@grigoris9gr)
• #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)

1.11.24 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)

1.11.25 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)

1.11. Changelog
• #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 variant 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)

1.11.26 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)
Symbols

_index

compos_after_inplace() (menpo.transform.base.composable.ComposableTransform method), 391
_compose_before_inplace() (menpo.transform.base.composable.ComposableTransform method), 391
_transform_inplace() (menpo.transform.base.Transformable method), 390
_view_2d() (menpo.image.Image method), 19
_view_2d() (menpo.image.MaskedImage method), 68
_view_2d() (menpo.shape.ColouredTriMesh method), 295
_view_2d() (menpo.shape.LabelledPointUndirectedGraph method), 262
_view_2d() (menpo.shape.PointCloud method), 182
_view_2d() (menpo.shape.PointDirectedGraph method), 228
_view_2d() (menpo.shape.PointTree method), 246
_view_2d() (menpo.shape.TexturedTriMesh method), 308
_view_2d() (menpo.shape.TriMesh method), 283
_view_landmarks_2d() (menpo.image.Image method), 20
_view_landmarks_2d() (menpo.image.MaskedImage method), 69
_view_landmarks_2d() (menpo.shape.ColouredTriMesh method), 297
_view_landmarks_2d() (menpo.shape.LabelledPointUndirectedGraph method), 266
_view_landmarks_2d() (menpo.shape.PointCloud method), 184
_view_landmarks_2d() (menpo.shape.PointDirectedGraph method), 230
_view_landmarks_2d() (menpo.shape.PointTree method), 248
_view_landmarks_2d() (menpo.shape.TexturedTriMesh method), 310
_view_landmarks_2d() (menpo.shape.TriMesh method), 285

A

add_label() (menpo.shape.LabelledPointUndirectedGraph method), 270
Affine (class in menpo.transform), 327
aligned_source() (menpo.transform.AlignmentAffine method), 357
aligned_source() (menpo.transform.AlignmentRotation method), 367
aligned_source() (menpo.transform.AlignmentSimilarity method), 362
aligned_source() (menpo.transform.AlignmentTranslation method), 373
aligned_source() (menpo.transform.AlignmentUniformScale method), 377
aligned_source() (menpo.transform.base.alignment.Alignment method), 394
aligned_source() (menpo.transform.ThinPlateSplines method), 354
Alignment (class in menpo.transform.base.alignment), 393
alignment_error() (menpo.transform.AlignmentAffine method), 357
alignment_error() (menpo.transform.AlignmentRotation method), 367
alignment_error() (menpo.transform.AlignmentSimilarity method), 362
alignment_error() (menpo.transform.AlignmentTranslation method), 373
alignment_error()
as_non_alignment()  
(menpo.transform.AlignmentSimilarity method), 363
as_non_alignment()  
(menpo.transform.AlignmentTranslation method), 373
as_non_alignment()  
(menpo.transform.AlignmentUniformScale method), 378
as_PILImage()  
(menpo.image.BooleanImage method), 45
as_PILImage()  
(menpo.image.Image method), 24
as_PILImage()  
(menpo.image.MaskedImage method), 73
as_pointgraph()  
(menpo.shape.ColouredTriMesh method), 301
as_pointgraph()  
(menpo.shape.TexturedTriMesh method), 314
as_pointgraph()  
(menpo.shape.TriMesh method), 289
as_unmasked()  
(menpo.image.MaskedImage method), 75
as_vector()  
(menpo.base.Vectorizable method), 3
as_vector()  
(menpo.image.BooleanImage method), 47
as_vector()  
(menpo.image.Image method), 26
as_vector()  
(menpo.image.MaskedImage method), 75
as_vector()  
(menpo.shape.base.Shape method), 180
as_vector()  
(menpo.shape.ColouredTriMesh method), 301
as_vector()  
(menpo.shape.ColouredPointUndirectedGraph method), 270
as_vector()  
(menpo.shape.PointCloud method), 188
as_vector()  
(menpo.shape.PointDirectedGraph method), 235
as_vector()  
(menpo.shape.PointTree method), 252
as_vector()  
(menpo.shape.PointUndirectedGraph method), 218
as_vector()  
(menpo.shape.TexturedTriMesh method), 314
as_vector()  
(menpo.shape.TriMesh method), 289
as_vector()  
(menpo.transform.Affine method), 328
as_vector()  
(menpo.transform.AlignmentAffine method), 358
as_vector()  
(menpo.transform.AlignmentRotation method), 368
as_vector()  
(menpo.transform.AlignmentSimilarity method), 363
as_vector()  
(menpo.transform.AlignmentTranslation method), 374
as_vector()  
(menpo.transform.AlignmentUniformScale method), 378
as_vector()  
(menpo.transform.Homogeneous method), 324
as_vector()  
(menpo.transform.NonUniformScale method), 351
as_vector()  
(menpo.transform.Rotation method), 337
as_vector()  
(menpo.transform.Similarity method), 332
as_vector()  
(menpo.transform.Translation method), 342
as_vector()  
(menpo.transform.UniformScale method), 346
axis_and_angle_of_rotation()  
(menpo.transform.AlignmentRotation method), 368
axis_and_angle_of_rotation()  
(menpo.transform.Rotation method), 337

B

BooleanImage (class in menpo.image), 45
boundary_tri_index()  
(menpo.shape.ColouredTriMesh method), 301
boundary_tri_index()  
(menpo.shape.TexturedTriMesh method), 314
boundary_tri_index()  
(menpo.shape.TriMesh method), 289
bounding_box()  (in module menpo.shape), 321
bounding_box()  (menpo.shape.ColouredTriMesh method), 302
bounding_box()  (menpo.shape.LabelledPointUndirectedGraph method), 270
bounding_box()  (menpo.shape.PointCloud method), 188
bounding_box()  (menpo.shape.PointDirectedGraph method), 235
bounding_box()  (menpo.shape.PointTree method), 252
bounding_box()  (menpo.shape.PointUndirectedGraph method), 218
bounding_box()  (menpo.shape.TexturedTriMesh method), 314
bounding_box()  (menpo.shape.TriMesh method), 289
bounding_box()  (module menpo.landmark), 114
bounding_box_to_bounding_box()  (module menpo.landmark), 115
bounding_cuboid()  (module menpo.shape), 322
bounds()  (menpo.image.BooleanImage method), 47
bounds()  (menpo.image.Image method), 26
bounds()  (menpo.image.MaskedImage method), 75
bounds()  (menpo.shape.ColouredTriMesh method), 302
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounds()</td>
<td>menpo.shape.LabelledPointUndirectedGraph method</td>
<td>271</td>
</tr>
<tr>
<td>bounds()</td>
<td>menpo.shape.PointCloud method</td>
<td>188</td>
</tr>
<tr>
<td>bounds()</td>
<td>menpo.shape.PointDirectedGraph method</td>
<td>235</td>
</tr>
<tr>
<td>bounds()</td>
<td>menpo.shape.PointTree method</td>
<td>252</td>
</tr>
<tr>
<td>bounds()</td>
<td>menpo.shape.TexturedTriMesh method</td>
<td>315</td>
</tr>
<tr>
<td>bounds()</td>
<td>menpo.shape.TriMesh method</td>
<td>290</td>
</tr>
<tr>
<td>bounds_false()</td>
<td>menpo.image.BooleanImage method</td>
<td>47</td>
</tr>
<tr>
<td>bounds_true()</td>
<td>menpo.image.BooleanImage method</td>
<td>47</td>
</tr>
<tr>
<td>build_mask_around_landmarks()</td>
<td>menpo.image.MaskedImage method</td>
<td>75</td>
</tr>
<tr>
<td>bytes_str()</td>
<td>in module menpo.visualize</td>
<td>404</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.image.BooleanImage method</td>
<td>48</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.image.Image method</td>
<td>26</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.image.MaskedImage method</td>
<td>75</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.ColouredTriMesh method</td>
<td>302</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.LabelledPointUndirectedGraph method</td>
<td>271</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.PointCloud method</td>
<td>189</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.PointDirectedGraph method</td>
<td>236</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.PointTree method</td>
<td>253</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.TexturedTriMesh method</td>
<td>315</td>
</tr>
<tr>
<td>centre()</td>
<td>menpo.shape.TriMesh method</td>
<td>290</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.LabelledPointUndirectedGraph method</td>
<td>271</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.PointCloud method</td>
<td>189</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.PointDirectedGraph method</td>
<td>236</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.PointTree method</td>
<td>253</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.TexturedTriMesh method</td>
<td>315</td>
</tr>
<tr>
<td>centre_of_bounds()</td>
<td>menpo.shape.TriMesh method</td>
<td>290</td>
</tr>
<tr>
<td>chain_graph()</td>
<td>(in module menpo.shape)</td>
<td>280</td>
</tr>
<tr>
<td>children()</td>
<td>menpo.shape.PointCloud method</td>
<td>236</td>
</tr>
<tr>
<td>children()</td>
<td>menpo.shape.PointDirectedGraph method</td>
<td>236</td>
</tr>
<tr>
<td>children()</td>
<td>menpo.shape.PointTree method</td>
<td>253</td>
</tr>
<tr>
<td>children()</td>
<td>menpo.shape.TriMesh method</td>
<td>290</td>
</tr>
<tr>
<td>clear()</td>
<td>menpo.landmark.LandmarkManager method</td>
<td>113</td>
</tr>
<tr>
<td>clear_figure()</td>
<td>menpo.visualize.MatplotlibRenderer method</td>
<td>397</td>
</tr>
<tr>
<td>clear_figure()</td>
<td>menpo.visualize.Renderer method</td>
<td>396</td>
</tr>
<tr>
<td>clip_pixels()</td>
<td>menpo.image.BooleanImage method</td>
<td>48</td>
</tr>
<tr>
<td>clip_pixels()</td>
<td>menpo.image.Image method</td>
<td>26</td>
</tr>
<tr>
<td>clip_pixels()</td>
<td>menpo.image.MaskedImage method</td>
<td>75</td>
</tr>
<tr>
<td>clip_texture()</td>
<td>menpo.shape.ColouredTriMesh method</td>
<td>302</td>
</tr>
<tr>
<td>clip_texture()</td>
<td>menpo.shape.TexturedTriMesh method</td>
<td>315</td>
</tr>
<tr>
<td>ColouredTriMesh</td>
<td>(class in menpo.shape)</td>
<td>295</td>
</tr>
<tr>
<td>complete_graph()</td>
<td>(in module menpo.shape)</td>
<td>280</td>
</tr>
<tr>
<td>component()</td>
<td>menpo.model.LinearVectorModel method</td>
<td>150</td>
</tr>
<tr>
<td>component()</td>
<td>menpo.model.MeanLinearVectorModel method</td>
<td>152</td>
</tr>
<tr>
<td>component()</td>
<td>menpo.model.PCAModel method</td>
<td>155</td>
</tr>
<tr>
<td>component()</td>
<td>menpo.model.PCAVectorModel method</td>
<td>166</td>
</tr>
<tr>
<td>component()</td>
<td>menpo.model.PCAModel method</td>
<td>155</td>
</tr>
<tr>
<td>components()</td>
<td>menpo.model.LinearVectorModel property</td>
<td>151</td>
</tr>
</tbody>
</table>
components() (menpo.model.MeanLinearVectorModel property), 154
components() (menpo.model.PCAModel property), 165
components() (menpo.model.PCAVectorModel property), 176
ComposableTransform (class in menpo.transform.base.composable), 391
compose_after() (menpo.transform.Affine method), 328
compose_after() (menpo.transform.AlignmentAffine method), 358
compose_after() (menpo.transform.AlignmentRotation method), 368
compose_after() (menpo.transform.AlignmentSimilarity method), 363
compose_after() (menpo.transform.AlignmentTranslation method), 374
compose_after() (menpo.transform.AlignmentUniformScale method), 378
compose_after() (menpo.transform.Alignments translation method), 374
compose_after() (menpo.transform.Alignments uniform scale method), 378
compose_after() (menpo.transform.base.composable.ComposableTransform method), 391
compose_after_from_vector_inplace() (menpo.transform.AlignmentTranslation method), 374
compose_after_from_vector_inplace() (menpo.transform.AlignmentUniformScale method), 379
compose_after_from_vector_inplace() (menpo.transform.base.composable.VComposable method), 395
compose_after_from_vector_inplace() (menpo.transform.Homogeneous method), 324
compose_after_from_vector_inplace() (menpo.transform.NonUniformScale method), 351
compose_after_from_vector_inplace() (menpo.transform.Transform method), 389
compose_after_from_vector_inplace() (menpo.transform.TransformChain method), 383
compose_after() (menpo.transform.R2LogR2RBF method), 387
compose_after() (menpo.transform.R2LogRRBF method), 387
compose_after() (menpo.transform.Rotation method), 337
compose_after() (menpo.transform.Similarity method), 333
compose_after() (menpo.transform.ThinPlateSplines method), 355
compose_after() (menpo.transform.Translation method), 342
compose_after() (menpo.transform.UniformScale method), 346
compose_after_inplace() (menpo.transform.Affine method), 328
compose_after_inplace() (menpo.transform.AlignmentAffine method), 358
compose_after_inplace() (menpo.transform.AlignmentRotation method), 368
compose_after_inplace() (menpo.transform.AlignmentSimilarity method), 363
compose_after_inplace() (menpo.transform.AlignmentTranslation method), 374
compose_after_inplace() (menpo.transform.AlignmentUniformScale method), 379
compose_after_inplace() (menpo.transform.base.composable.ComposableTransform method), 392
compose_after_inplace() (menpo.transform.Homogeneous method), 324
compose_after_inplace() (menpo.transform.NonUniformScale method), 351
compose_after_inplace() (menpo.transform.Rotation method), 337
compose_after_inplace() (menpo.transform.Similarity method), 333
compose_after_inplace() (menpo.transform.Translation method), 342
compose_after_inplace() (menpo.transform.UniformScale method), 347
(menpo.transform.TransformChain method), 383

compose_after_inplace()
(menpo.transform.Translation method), 342

compose_after_inplace()
(menpo.transform.UniformScale method), 347

compose_before()
(menpo.transform.Affine method), 328

compose_before()
(menpo.transform.AlignmentAffine method), 358

compose_before()
(menpo.transform.AlignmentRotation method), 369

compose_before()
(menpo.transform.AlignmentSimilarity method), 363

compose_before()
(menpo.transform.AlignmentTranslation method), 375

compose_before()
(menpo.transform.AlignmentUniformScale method), 379

compose_before_inplace()
(menpo.transform.Affine method), 330

compose_before_inplace()
(menpo.transform.AlignmentAffine method), 360

compose_before_inplace()
(menpo.transform.AlignmentRotation method), 372

compose_before_inplace()
(menpo.transform.AlignmentUniformScale method), 381

compose_before_inplace()
(menpo.transform.base.composable.ComposableTransform method), 392

compose_before_inplace()
(menpo.transform.Homogeneous method), 325

compose_before_inplace()
(menpo.transform.NonUniformScale method), 353

compose_before_inplace()
(menpo.transform.R2LogR2RBF method), 338

compose_before_inplace()
(menpo.transform.R2LogRRBF method), 343

compose_before_inplace()
(menpo.transform.Rotation method), 347

compose_before_inplace()
(menpo.transform.Similarity method), 355

compose_before_inplace()
(menpo.transform.ThinPlateSplines method), 369

compose_inplace_with()
(menpo.transform.Affine property), 330

compose_inplace_with()
(menpo.transform.AlignmentAffine property), 360

compose_inplace_with()
(menpo.transform.AlignmentRotation property), 372

compose_inplace_with()
(menpo.transform.AlignmentUniformScale property), 381

compose_inplace_with()
(menpo.transform.ALIGNMENTSIMILARITY property), 366

compose_inplace_with()
(menpo.transform.ALIGNMENTTRANSLATION property), 376

compose_inplace_with()
(menpo.transform.ALIGNMENTUNIFORMSCALE property), 384

compose_inplace_with()
(menpo.transform.Homogeneous property), 326

compose_inplace_with()
(menpo.transform.NonUniformScale property), 353

compose_inplace_with()
(menpo.transform.Rotation property), 340
composes_inplace_with()
(menpo.transform.Similarity property), 335
composes_inplace_with()
(menpo.transform.TransformChain property), 384
composes_inplace_with()
(menpo.transform.Translation property), 344
composes_inplace_with()
(menpo.transform.UniformScale property), 349
composes_with()
(menpo.transform.Affine property), 331
composes_with()
(menpo.transform.AlignmentAffine property), 361
composes_with()
(menpo.transform.AlignmentRotation property), 372
composes_with()
(menpo.transform.AlignmentSimilarity property), 366
composes_with()
(menpo.transform.AlignmentTranslation property), 376
composes_with()
(menpo.transform.AlignmentUniformScale property), 381
composes_with()
(menpo.transform.base.composable.ComposableTransform property), 393
composes_with()
(menpo.transform.Homogeneous property), 326
composes_with()
(menpo.transform.NonUniformScale property), 353
composes_with()
(menpo.transform.Rotation property), 340
composes_with()
(menpo.transform.Similarity property), 335
composes_with()
(menpo.transform.TransformChain property), 384
composes_with()
(menpo.transform.Translation property), 345
composes_with()
(menpo.transform.UniformScale property), 349
constrain_landmarks_to_bounds()
(menpo.image.BooleanImage method), 48
constrain_landmarks_to_bounds()
(menpo.image.Image method), 26
constrain_landmarks_to_bounds()
(menpo.image.MaskedImage method), 76
constrain_mask_to_landmarks()
(menpo.image.MaskedImage method), 76
constrain_mask_to_points_around_landmarks()
(menpo.image.MaskedImage method), 76
constrain_points_to_bounds()
(menpo.image.MaskedImage method), 76
constrain_points_to_bounds()
(menpo.image.Image method), 26
constrain_points_to_bounds()
(menpo.image.MaskedImage method), 76
constrain_points_to_bounds()
(menpo.image.MaskedImage method), 76
constrain_points_to_bounds()
(menpo.image.Image method), 26
constrain_to_bounds()
(menpo.shape.ColouredTriMesh method), 302
constrain_to_bounds()
(menpo.shape.LabelledPointUndirectedGraph method), 271
constrain_to_bounds()
(menpo.shape.PointCloud method), 189
constrain_to_bounds()
(menpo.shape.PointDirectedGraph method), 236
constrain_to_bounds()
(menpo.shape.PointTree method), 253
constrain_to_bounds()
(menpo.shape.PointUndirectedGraph method), 219
constrain_to_bounds()
(menpo.shape.TexturedTriMesh method), 315
constrain_to_bounds()
(menpo.shape.TriMesh method), 290
constrain_to_landmarks()
(menpo.image.BooleanImage method), 48
constrain_to_pointcloud()
(menpo.image.BooleanImage method), 49
copy()
(menpo.base.Copyable method), 3
copy()
(menpo.base.LazyList method), 5
copy()
(menpo.base.Targetable method), 5
copy()
(menpo.base.Vectorizable method), 4
copy()
(menpo.image.BooleanImage method), 49
copy()
(menpo.image.Image method), 26
copy()
(menpo.image.MaskedImage method), 77
copy()
(menpo.landmark.Landmarkable method), 112
copy()
(menpo.landmark.LandmarkManager method), 113
copy()
(menpo.model.LinearVectorModel method), 150
copy()
(menpo.model.MeanLinearVectorModel method), 152
copy()
(menpo.model.PCAModel method), 155
copy()
(menpo.model.PCAVectorModel method), 166
copy()
(menpo.shape.base.Shape method), 181
copy()
(menpo.shape.ColouredTriMesh method), 303
copy()
(menpo.shape.LabelledPointUndirectedGraph method), 271
copy()
(menpo.shape.PointCloud method), 189
copy()
(menpo.shape.PointDirectedGraph method), 236
copy()
(menpo.shape.PointTree method), 253
copy()
(menpo.shape.PointUndirectedGraph method), 219
copy()
(menpo.shape.TexturedTriMesh method), 315
copy()
(menpo.shape.TriMesh method), 290
distance_to() (menpo.shape.PointDirectedGraph method), 236
distance_to() (menpo.shape.PointTree method), 253
distance_to() (menpo.shape.PointUndirectedGraph method), 219
distance_to() (menpo.shape.TexturedTriMesh method), 316
dot_inplace_left() (in module menpo.math), 147
dot_inplace_right() (in module menpo.math), 146
double_igo() (in module menpo.feature), 106
dsift() (in module menpo.feature), 104
edge_indices() (menpo.shape.ColouredTriMesh method), 303
edge_indices() (menpo.shape.TexturedTriMesh method), 316
edge_indices() (menpo.shape.TriMesh method), 291
edge_lengths() (menpo.shape.ColouredTriMesh method), 303
edge_lengths() (menpo.shape.TexturedTriMesh method), 316
edge_lengths() (menpo.shape.TriMesh method), 291
edge_vectors() (menpo.shape.ColouredTriMesh method), 303
edge_vectors() (menpo.shape.TexturedTriMesh method), 316
edge_vectors() (menpo.shape.TriMesh method), 291
dges() (menpo.shape.DirectedGraph property), 203
dges() (menpo.shape.LabelledPointUndirectedGraph property), 278
dges() (menpo.shape.PointDirectedGraph property), 243
dges() (menpo.shape.PointTree property), 260
dges() (menpo.shape.PointUndirectedGraph property), 225
dges() (menpo.shape.TriMesh property), 209
dges() (menpo.shape.UnirectedGraph property), 197
eigenvalue_decomposition() (in module menpo.math), 144
eigenvalues() (menpo.model.PCAModel property), 165
eigenvalues() (menpo.model.PCAVectorModel property), 176
eigenvalues_cumulative_ratio() (menpo.model.PCAVectorModel method), 155
eigenvalues_ratio() (menpo.model.PCAModel method), 155
eigenvalues_ratio() (menpo.model.PCAVectorModel method), 167
empty_graph() (in module menpo.shape), 279
erode() (menpo.image.MaskedImage method), 80
es() (in module menpo.feature), 99
export_image() (in module menpo.io), 15
export_landmark_file() (in module menpo.io), 16
export_pickle() (in module menpo.io), 17
export_video() (in module menpo.io), 16
extract_channels() (menpo.image.BooleanImage method), 52
extract_channels() (menpo.image.Image method), 29
extract_channels() (menpo.image.MaskedImage method), 80
extract_patches() (menpo.image.BooleanImage method), 52
extract_patches() (menpo.image.Image method), 29
extract_patches() (menpo.image.MaskedImage method), 80
extract_patches_around_landmarks() (menpo.image.BooleanImage method), 53
extract_patches_around_landmarks() (menpo.image.Image method), 30
extract_patches_around_landmarks() (menpo.image.MaskedImage method), 81
eye_ibug_close_17_to_eye_ibug_close_17() (in module menpo.landmark), 128
eye_ibug_close_17_to_eye_ibug_close_17_trimesh() (in module menpo.landmark), 128
eye_ibug_open_38_to_eye_ibug_open_38() (in module menpo.landmark), 130
eye_ibug_open_38_to_eye_ibug_open_38_trimesh() (in module menpo.landmark), 130

F

face_bu3dge_83_to_face_bu3dge_83() (in module menpo.landmark), 127
face_ibug_49_to_face_ibug_49() (in module menpo.landmark), 124
face_ibug_68_mirrored_to_face_ibug_68() (in module menpo.landmark), 123
face_ibug_68_to_face_ibug_49() (in module menpo.landmark), 116
face_ibug_68_to_face_ibug_49_trimesh() (in module menpo.landmark), 117

Index 437
from_vector() (menpo.base.Vectorizable method), 4
from_vector() (menpo.image.BooleanImage method), 53
from_vector() (menpo.image.Image method), 30
from_vector() (menpo.image.MaskedImage method), 81
from_vector() (menpo.shape.base.Shape method), 181
from_vector() (menpo.shape.ColouredTriMesh method), 304
from_vector() (menpo.shape.LabelledPointUndirectedGraph method), 273
from_vector() (menpo.shape.PointCloud method), 189
from_vector() (menpo.shape.PointDirectedGraph method), 238
from_vector() (menpo.shape.PointTree method), 255
from_vector() (menpo.shape.PointUndirectedGraph method), 221
from_vector() (menpo.shape.TexturedTriMesh method), 317
from_vector() (menpo.shape.TriMesh method), 291
from_vector() (menpo.transform.Affine method), 329
from_vector() (menpo.transform.AlignmentAffine method), 359
from_vector() (menpo.transform.AlignmentRotation method), 369
from_vector() (menpo.transform.AlignmentSimilarity method), 364
from_vector() (menpo.transform.AlignmentTranslation method), 375
from_vector() (menpo.transform.AlignmentUniformScale method), 379
from_vector() (menpo.transform.Homogeneous method), 325
from_vector() (menpo.transform.NonUniformScale method), 352
from_vector() (menpo.transform.Rotation method), 338
from_vector() (menpo.transform.Similarity method), 334
from_vector() (menpo.transform.Translation method), 343
from_vector() (menpo.base.Vectorizable method), 4
from_vector_inplace() (menpo.image.Image method), 31
from_vector_inplace() (menpo.image.MaskedImage method), 81
from_vector_inplace() (menpo.image.BooleanImage method), 54
from_vector_inplace() (menpo.shape.base.Shape method), 181
from_vector_inplace() (menpo.shape.ColouredTriMesh method), 304
from_vector_inplace() (menpo.shape.LabelledPointUndirectedGraph method), 273
from_vector_inplace() (menpo.shape.PointCloud method), 189
from_vector_inplace() (menpo.shape.PointDirectedGraph method), 238
from_vector_inplace() (menpo.shape.PointTree method), 255
from_vector_inplace() (menpo.shape.PointUndirectedGraph method), 221
from_vector_inplace() (menpo.shape.TexturedTriMesh method), 317
from_vector_inplace() (menpo.shape.TriMesh method), 292
from_vector_inplace() (menpo.transform.Affine method), 329
from_vector_inplace() (menpo.transform.AlignmentAffine method), 359
from_vector_inplace() (menpo.transform.AlignmentRotation method), 369
from_vector_inplace() (menpo.transform.AlignmentSimilarity method), 364
from_vector_inplace() (menpo.transform.AlignmentTranslation method), 375
from_vector_inplace() (menpo.transform.AlignmentUniformScale method), 380
from_vector_inplace() (menpo.transform.Homogeneous method), 325
from_vector_inplace() (menpo.transform.NonUniformScale method), 352
from_vector_inplace() (menpo.transform.Rotation method), 338
from_vector_inplace() (menpo.transform.Similarity method), 334
from_vector_inplace() (menpo.transform.Translation method), 343
has_nan_values() (menpo.transform.AlignmentUniformScale method), 380
has_nan_values() (menpo.transform.Homogeneous method), 326
has_nan_values() (menpo.transform.NonUniformScale method), 352
has_nan_values() (menpo.transform.Rotation method), 338
has_nan_values() (menpo.transform.Similarity method), 334
has_nan_values() (menpo.transform.Translation method), 344
has_nan_values() (menpo.transform.UniformScale method), 348
has_true_inverse() (menpo.transform.Affine property), 331
has_true_inverse() (menpo.transform.AlignmentAffine property), 361
has_true_inverse() (menpo.transform.AlignmentRotation property), 372
has_true_inverse() (menpo.transform.AlignmentSimilarity property), 366
has_true_inverse() (menpo.transform.AlignmentTranslation property), 377
has_true_inverse() (menpo.transform.AlignmentUniformScale property), 381
has_true_inverse() (menpo.transform.base.invertible.Invertible property), 393
has_true_inverse() (menpo.transform.base.invertible.VInvertible property), 396
has_true_inverse() (menpo.transform.Homogeneous property), 327
has_true_inverse() (menpo.transform.NonUniformScale property), 353
has_true_inverse() (menpo.transform.Rotation property), 341
has_true_inverse() (menpo.transform.Similarity property), 335
has_true_inverse() (menpo.transform.ThinPlateSplines property), 356
has_true_inverse() (menpo.transform.Translation property), 345
has_true_inverse() (menpo.transform.UniformScale property), 349
height() (menpo.image.BooleanImage property), 67
height() (menpo.image.Image property), 44
height() (menpo.image.MaskedImage property), 96
hellinger_vector_128_d sift() (in module menpo.feature), 106
hog() (in module menpo.feature), 101
Homogeneous (class in menpo.transform), 323
igo() (in module menpo.feature), 99
Image (class in menpo.image), 18
image_paths() (in module menpo.io), 17
ImageBoundaryError (class in menpo.image), 97
import_builtin_asset() (in module menpo.io), 14
import_image() (in module menpo.io), 8
import_images() (in module menpo.io), 8
import_landmark_file() (in module menpo.io), 12
import_landmark_files() (in module menpo.io), 12
import_pickles() (in module menpo.io), 13
import_videos() (in module menpo.io), 11
increment() (menpo.model.GMRFModel method), 177
increment() (menpo.model.GMRFVectorModel method), 179
increment() (menpo.model.PCAModel method), 155
increment() (menpo.model.PCAVectorModel method), 167
index() (menpo.base.LazyList method), 5
indices() (menpo.image.BooleanImage method), 54
indices() (menpo.image.Image method), 31
indices() (menpo.image.MaskedImage method), 82
init_2d_grid() (menpo.shape.ColouredTriMesh class method), 304
init_2d_grid() (menpo.shape.LabelledPointUndirectedGraph class method), 274
init_2d_grid() (menpo.shape.PointCloud class method), 190
init_2d_grid() (menpo.shape.PointDirectedGraph class method), 239
init_2d_grid() (menpo.shape.PointTree class method), 256
init_2d_grid() (menpo.shape.TexturedTriMesh class method), 317
init_2d_grid() (menpo.shape.TriMesh class method), 292
init_3d_from_quaternion()
(menpo.transform.AlignmentRotation class method), 370
init_3d_from_quaternion()
(menpo.transform.Rotation class method), 339
init_blank() (menpo.image.BooleanImage class method), 54
init_blank() (menpo.image.Image class method), 31
init_blank() (menpo.image.MaskedImage class method), 82
init_from_2d_ccw_angle()
(menpo.transform.AlignmentRotation class method), 370
init_from_2d_ccw_angle()
(menpo.transform.Rotation class method), 339
init_from_2d_shear() (menpo.transform.Affine class method), 330
init_from_2d_shear() (menpo.transform.AlignmentAffine class method), 359
init_from_2d_shear() (menpo.transform.AlignmentRotation class method), 370
init_from_2d_shear() (menpo.transform.AlignmentSimilarity class method), 365
init_from_2d_shear() (menpo.transform.AlignmentTranslation class method), 375
init_from_2d_shear() (menpo.transform.AlignmentUniformScale class method), 380
init_from_2d_shear() (menpo.transform.NonUniformScale class method), 352
init_from_2d_shear() (menpo.transform.Rotation class method), 339
init_from_2d_shear() (menpo.transform.Similarity class method), 334
init_from_2d_shear() (menpo.transform.Translation class method), 344
init_from_2d_shear() (menpo.transform.UniformScale class method), 348
init_from_3d_ccw_angle_around_x()
(menpo.transform.AlignmentRotation class method), 370
init_from_3d_ccw_angle_around_y()
(menpo.transform.AlignmentRotation class method), 370
init_from_3d_ccw_angle_around_y()
(menpo.transform.Rotation class method), 339
init_from_3d_ccw_angle_around_z()
(menpo.transform.AlignmentRotation class method), 370
init_from_3d_ccw_angle_around_z()
(menpo.transform.Rotation class method), 339
init_from_channels_at_back()
(menpo.image.BooleanImage class method), 54
init_from_channels_at_back()
(menpo.image.Image class method), 31
init_from_channels_at_back()
(menpo.image.MaskedImage class method), 83
init_from_components()
(menpo.model.PCAModel class method), 156
init_from_components()
(menpo.model.PCAVectorModel class method), 167
init_from_covariance_matrix()
(menpo.model.PCAModel class method), 156
init_from_covariance_matrix()
(menpo.model.PCAVectorModel class method), 168
init_from_depth_image()
(menpo.shape.ColouredTriMesh class method), 305
init_from_depth_image()
(menpo.shape.LabelledPointUndirectedGraph class method), 274
init_from_depth_image()
(menpo.shape.PointCloud class method), 190
init_from_depth_image()
(menpo.shape.PointDirectedGraph class method), 239
init_from_depth_image()
(menpo.shape.PointTree class method), 256
init_from_depth_image()
(menpo.shape.PointUndirectedGraph class method), 222
init_from_depth_image()
(menpo.shape.TexturedTriMesh class method), 317
init_from_depth_image()
(menpo.shape.TriMesh class method), 292
init_from_edges() (menpo.shape.DirectedGraph
init_from_edges()

class method, 201
init_from_edges()

(menpo.shape.LabelledPointUndirectedGraph

class method), 275
init_from_edges()

(menpo.shape.PointDirectedGraph

class method), 240
init_from_edges()

(menpo.shape.PointTree

class method), 257
init_from_edges()

(menpo.shape.PointUndirectedGraph

class method), 223
init_from_edges()

(menpo.shape.Tree

class method), 207
init_from_edges()

(menpo.shape.UndirectedGraph

class method), 195
init_from_index_callable()

class method, 6
init_from_indices_mapping()

(menpo.shape.LabelledPointUndirectedGraph

class method), 275
init_from_iterable()

(menpo.base.LazyList

class method), 6
init_from_pointcloud()

(menpo.image.BooleanImage

class method), 54
init_from_pointcloud()

(menpo.image.Image

class method), 32
init_from_pointcloud()

(menpo.image.MaskedImage

class method), 83
init_from_rolled_channels()

(menpo.image.BooleanImage

class method), 55
init_from_rolled_channels()

(menpo.image.Image

class method), 32
init_from_rolled_channels()

(menpo.image.MaskedImage

class method), 83
init_identity()

(menpo.transform.Affine

class method), 330
init_identity()

(menpo.transform.AlignmentAffine

class method), 360
init_identity()

(menpo.transform.AlignmentRotation

class method), 371
init_identity()

(menpo.transform.AlignmentSimilarity

class method), 365
init_identity()

(menpo.transform.AlignmentTranslation

edge)

(menpo.shape.PointTree

method), 258
init_identity()

(menpo.transform.AlignmentUniformScale

class method), 380
init_identity()

(menpo.transform.Homogeneous

class method), 326
init_identity()

(menpo.transform.NonUniformScale

class method), 353
init_identity()

(menpo.transform.Rotation

class method), 340
init_identity()

(menpo.transform.Similarity

class method), 334
init_identity()

(menpo.transform.Translation

class method), 344
init_identity()

(menpo.transform.UniformScale

class method), 348
init_identity()

(menpo.transform.AlignmentAffine

class method), 360
init_identity()

(menpo.transform.AlignmentRotation

class method), 375
init_identity()

(menpo.transform.AlignmentSimilarity

class method), 365
init_identity()

(menpo.transform.AlignmentTranslation

edge)

(menpo.shape.PointTree

method), 258
init_identity()

(menpo.transform.AlignmentUniformScale

class method), 380
init_identity()

(menpo.transform.Homogeneous

class method), 326
init_identity()

(menpo.transform.NonUniformScale

class method), 353
init_identity()

(menpo.transform.Rotation

class method), 340
init_identity()

(menpo.transform.Similarity

class method), 334
init_identity()

(menpo.transform.Translation

class method), 344
init_identity()

(menpo.transform.UniformScale

class method), 348
init_identity()

(menpo.transform.AlignmentAffine

class method), 360
init_identity()

(menpo.transform.AlignmentRotation

class method), 375
init_identity()

(menpo.transform.AlignmentSimilarity

class method), 365
init_identity()

(menpo.transform.AlignmentTranslation

edge)

(menpo.shape.PointTree

method), 258
init_identity()

(menpo.transform.AlignmentUniformScale

class method), 380
init_identity() (menpo.transform.Similarity class method), 334
init_identity() (menpo.transform.Translation class method), 344
init_identity() (menpo.transform.UniformScale class method), 348
init_with_all_label()

(menpo.shape.LabelledPointUndirectedGraph

class method), 276
instance()

(menpo.model.LinearVectorModel

method), 150
instance()

(menpo.model.MeanLinearVectorModel

method), 152
instance() (menpo.model.PCAModel method), 156
instance() (menpo.model.PCAVectorModel method), 168
instance_vectors()

(menpo.model.LinearVectorModel

method), 150
instance_vectors()

(menpo.model.MeanLinearVectorModel

method), 153
instance_vectors() (menpo.model.PCAModel method), 157
instance_vectors()

(menpo.model.PCAVectorModel

method), 168
inverse_noise_variance()

(menpo.model.PCAModel method), 157
inverse_noise_variance()

(menpo.model.PCAVectorModel

method), 169
invert() (menpo.image.BooleanImage method), 55
Invertible

(class in
menpo.transform.base.invertible), 393
ipca() (in module menpo.math), 146
is_edge() (menpo.shape.DirectedGraph method), 202
is_edge() (menpo.shape.LabelledPointUndirectedGraph

method), 276
is_edge() (menpo.shape.PointDirectedGraph

method), 241
is_edge() (menpo.shape.PointTree method), 258
is_edge() (menpo.shape.PointUndirectedGraph

method), 224
is_edge() (menpo.shape.Tree method), 208
is_edge() (menpo.shape.Un directedGraph

method), 196
is_leaf() (menpo.shape.PointTree method), 258
is_leaf() (menpo.shape.Tree method), 208
is_tree() (menpo.shape.DirectedGraph method), 202
is_tree() (menpo.shape.LabelledPointUndirectedGraph

method), 276
is_tree() (menpo.shape.PointDirectedGraph method), 241
is_tree() (menpo.shape.PointTree method), 258
is_tree() (menpo.shape.PointUndirectedGraph method), 224
is_tree() (menpo.shape.Tree method), 208
is_tree() (menpo.shape.UndirectedGraph method), 196
isolated_vertices() (menpo.shape.DirectedGraph method), 202
isolated_vertices() (menpo.shape.LabelledPointUndirectedGraph method), 276
isolated_vertices() (menpo.shape.PointDirectedGraph method), 241
isolated_vertices() (menpo.shape.PointTree method), 258
isolated_vertices() (menpo.shape.PointUndirectedGraph method), 224
isolated_vertices() (menpo.shape.Tree method), 208
isolated_vertices() (menpo.shape.UndirectedGraph method), 196
items() (menpo.landmark.LandmarkManager method), 113
items_matching() (menpo.landmark.LandmarkManager method), 113

K
keys() (menpo.landmark.LandmarkManager method), 113
keys_matching() (menpo.landmark.LandmarkManager method), 113

L
Labellable() (in module menpo.landmark), 114
LabellingError (class in menpo.landmark), 112
labels() (menpo.shape.LabelledPointUndirectedGraph property), 278
landmark_file_paths() (in module menpo.io), 18
Landmarkable (class in menpo.landmark), 112
LandmarkableViewable (class in menpo.visualize), 397
LandmarkManager (class in menpo.landmark), 113
landmarks() (menpo.image.BooleanImage property), 67
landmarks() (menpo.image.Image property), 44
landmarks() (menpo.image.MaskedImage property), 96
landmarks() (menpo.landmark.Landmarkable property), 112
landmarks() (menpo.shape.base.Shape property), 181
landmarks() (menpo.shape.ColouredTriMesh property), 307
landmarks() (menpo.shape.LabelledPointUndirectedGraph property), 278
landmarks() (menpo.shape.PointCloud property), 191
landmarks() (menpo.shape.PointDirectedGraph property), 243
landmarks() (menpo.shape.PointTree property), 261
landmarks() (menpo.shape.PointUndirectedGraph property), 226
landmarks() (menpo.shape.TexturedTriMesh property), 320
LazyList (class in menpo.base), 5
lbp() (in module menpo.feature), 100
leaves() (menpo.shape.PointTree property), 261
leaves() (menpo.shape.Tree property), 209
linear_component() (menpo.transform.Affine property), 331
linear_component() (menpo.transform.AlignmentAffine property), 361
linear_component() (menpo.transform.AlignmentRotation property), 372
linear_component() (menpo.transform.AlignmentSimilarity property), 366
linear_component() (menpo.transform.AlignmentTranslation property), 377
linear_component() (menpo.transform.AlignmentUniformScale property), 381
linear_component() (menpo.transform.NonUniformScale property), 354
linear_component() (menpo.transform.Rotation property), 341
linear_component() (menpo.transform.Similarity property), 335
linear_component() (menpo.transform.Translation property), 345
linear_component() (menpo.transform.UniformScale property), 349
LinearModel (in module menpo.model), 149
LinearVectorModel (class in menpo.model), 150
lms() (menpo.shape.ColouredTriMesh property), 307
lms() (menpo.shape.LabelledPointUndirectedGraph property), 278
lms() (menpo.shape.PointCloud property), 191
lms() (menpo.shape.PointDirectedGraph property), 243
lms() (menpo.shape.TexturedTriMesh property), 307
lms() (menpo.shape.TriMesh property), 294
log_gabor() (in module menpo.math), 148
ls_builtin_assets() (in module menpo.io), 18
mahalanobis_distance() (menpo.model.GMRFModel method), 178
mahalanobis_distance() (menpo.model.GMRFVectorModel method), 180
map() (menpo.base.LazyList method), 6
mask() (menpo.image.BooleanImage property), 67
masked_pixels() (menpo.image.MaskedImage method), 83
MaskedImage (class in menpo.image), 68
MatplotlibRenderer (class in menpo.visualize), 397
maximum_depth() (menpo.shape.PointTree property), 261
maximum_depth() (menpo.shape.PointUndirectedGraph property), 276
mean() (menpo.model.GMRFModel method), 178
mean() (menpo.model.GMRFVectorModel method), 180
mean() (menpo.model.MeanLinearVectorModel method), 153
mean() (menpo.model.PCAModel method), 157
mean() (menpo.model.PCAVectorModel method), 169
mean_aligned_shape() (menpo.transform.GeneralizedProcrustesAnalysis method), 382
mean_alignment_error() (menpo.transform.GeneralizedProcrustesAnalysis method), 382
mean_edge_length() (menpo.shape.ColouredTriMesh method), 305
mean_edge_length() (menpo.shape.TexturedTriMesh method), 318
mean_edge_length() (menpo.shape.TriMesh method), 292
mean_pointcloud() (in module menpo.shape), 321
mean_tri_area() (menpo.shape.TexturedTriMesh method), 318
mean_tri_area() (menpo.shape.TriMesh method), 293
mean_vector() (menpo.model.PCAModel property), 165
MeanLinearModel (in module menpo.model), 152
MeanLinearVectorModel (class in menpo.model), 152
menpo_src_dir_path() (in module menpo.base), 7
MenpoDeprecationWarning (class in menpo.base), 7
MenpoMissingDependencyError (class in menpo.base), 7
minimum_spanning_tree() (menpo.shape.LabelledPointUndirectedGraph method), 276
minimum_spanning_tree() (menpo.shape.PointUndirectedGraph method), 224
minimum_spanning_tree() (menpo.shape.PointDirectedGraph method), 196
mirror() (menpo.image.BooleanImage method), 55
mirror() (menpo.image.Image method), 32
mirror() (menpo.image.MaskedImage method), 83
msg (menpo.base.MenpoMissingDependencyError attribute), 7
MultipleAlignment (class in menpo.transform.groupalign.base), 395

n_active_components() (menpo.model.PCAModel property), 165
n_active_components() (menpo.model.PCAVectorModel property), 176
n_centres() (menpo.transform.R2LogR2RBFP property), 386
n_centres() (menpo.transform.R2LogRRBF property), 388
n_channels() (menpo.image.BooleanImage property), 67
n_channels() (menpo.image.Image property), 44
n_channels() (menpo.image.MaskedImage property), 96
n_channels() (menpo.shape.ColouredTriMesh property), 307
n_channels() (menpo.shape.TexturedTriMesh property), 320
n_children() (menpo.shape.DirectedGraph method), 202
n_children() (menpo.shape.PointDirectedGraph method), 242

446 Index
n_children() (menpo.shape.PointTree method), 258
n_children() (menpo.shape.Tree method), 208
n_components() (menpo.model.LinearVectorModel property), 151
n_components() (menpo.model.MeanLinearVectorModel property), 154
n_components() (menpo.model.PCAModel property), 166
n_dims() (menpo.model.PCAVectorModel property), 176
n_dims() (menpo.base.Targetable property), 5
n_dims() (menpo.image.BooleanImage property), 67
n_dims() (menpo.image.Image property), 44
n_dims() (menpo.image.MaskedImage property), 96
n_dims() (menpo.landmark.Landmarkable method), 112
n_dims() (menpo.landmark.LandmarkManager property), 114
n_dims() (menpo.shape.base.Shape method), 181
n_dims() (menpo.shape.ColouredTriMesh property), 307
n_dims() (menpo.shape.LabelledPointUndirectedGraph property), 278
n_dims() (menpo.shape.PointCloud property), 191
n_dims() (menpo.shape.PointDirectedGraph property), 244
n_dims() (menpo.shape.PointTree property), 261
n_dims() (menpo.shape.PointUndirectedGraph property), 226
n_dims() (menpo.shape.TexturedTriMesh property), 320
n_dims() (menpo.shape.TriMesh property), 294
n_dims() (menpo.transform.Affine property), 331
n_dims() (menpo.transform.AlignmentAffine property), 361
n_dims() (menpo.transform.AlignmentRotation property), 372
n_dims() (menpo.transform.AlignmentSimilarity property), 366
n_dims() (menpo.transform.AlignmentTranslation property), 377
n_dims() (menpo.transform.AlignmentUniformScale property), 381
n_dims() (menpo.transform.R2LogR2RBF property), 386
n_dims() (menpo.transform.R2LogRRBF property), 388
n_dims() (menpo.transform.Rotation property), 341
n_dims() (menpo.transform.Similarity property), 335
n_dims() (menpo.transform.Transform property), 390
n_dims() (menpo.transform.TransformChain property), 385
n_dims_output() (menpo.transform.Affine property), 331
n_dims_output() (menpo.transform.AlignmentAffine property), 361
n_dims_output() (menpo.transform.AlignmentRotation property), 372
n_dims_output() (menpo.transform.AlignmentSimilarity property), 366
n_dims_output() (menpo.transform.AlignmentTranslation property), 377
n_dims_output() (menpo.transform.AlignmentUniformScale property), 381
n_dims_output() (menpo.transform.R2LogR2RBF property), 386
n_dims_output() (menpo.transform.R2LogRRBF property), 388
n_dims_output() (menpo.transform.Rotation property), 341
n_dims_output() (menpo.transform.Similarity property), 335
n_dims_output() (menpo.transform.Transform property), 390
n_dims_output() (menpo.transform.TransformChain property), 385
n_dims_output() (menpo.transform.Transform property), 345
n_dims_output() (menpo.transform.Transform property), 350
n_dims_output() (menpo.transform.Homogeneous property), 327
n_dims_output() (menpo.transform.NonUniformScale property), 354
n_dims_output() (menpo.transform.R2LogR2RBF property), 386
n_dims_output() (menpo.transform.R2LogRRBF property), 388
n_edges() (menpo.shape.DirectedGraph property), 203
n_edges() (menpo.shape.LabelledPointUndirectedGraph property), 278
n_edges() (menpo.shape.PointDirectedGraph property), 244
n_edges() (menpo.shape.PointTree property), 261
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_edges()</td>
<td>(menpo.shape.PointUndirectedGraph property), 226</td>
</tr>
<tr>
<td>n_edges()</td>
<td>(menpo.shape.Tree property), 210</td>
</tr>
<tr>
<td>n_edges()</td>
<td>(menpo.shape.UndirectedGraph property), 197</td>
</tr>
<tr>
<td>n_elements()</td>
<td>(menpo.image.BooleanImage property), 67</td>
</tr>
<tr>
<td>n_elements()</td>
<td>(menpo.image.Image property), 44</td>
</tr>
<tr>
<td>n_elements()</td>
<td>(menpo.image.MaskedImage property), 96</td>
</tr>
<tr>
<td>n_false()</td>
<td>(menpo.image.BooleanImage method), 55</td>
</tr>
<tr>
<td>n_false_elements()</td>
<td>(menpo.image.MaskedImage method), 84</td>
</tr>
<tr>
<td>n_false_pixels()</td>
<td>(menpo.image.MaskedImage method), 84</td>
</tr>
<tr>
<td>n_features()</td>
<td>(menpo.model.LinearVectorModel property), 151</td>
</tr>
<tr>
<td>n_features()</td>
<td>(menpo.model.MeanLinearVectorModel property), 154</td>
</tr>
<tr>
<td>n_features()</td>
<td>(menpo.model.PCAModel property), 166</td>
</tr>
<tr>
<td>n_features()</td>
<td>(menpo.model.PCAVectorModel property), 176</td>
</tr>
<tr>
<td>n_groups()</td>
<td>(menpo.landmark.LandmarkManager property), 114</td>
</tr>
<tr>
<td>n_labels()</td>
<td>(menpo.shape.LabelledPointUndirectedGraph property), 278</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.image.BooleanImage property), 67</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.image.Image property), 44</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.image.MaskedImage property), 96</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.landmark.Landmarkable property), 112</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.base.Shape property), 181</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.ColouredTriMesh property), 307</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.LabelledPointUndirectedGraph property), 279</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.PointCloud property), 191</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.PointDirectedGraph property), 244</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.PointTree property), 261</td>
</tr>
<tr>
<td>n_landmark_groups()</td>
<td>(menpo.shape.PointUndirectedGraph property), 226</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.base.Vectorizable property), 4</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.image.BooleanImage property), 67</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.image.Image property), 44</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.image.MaskedImage property), 96</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.base.Shape property), 181</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.ColouredTriMesh property), 307</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.LabelledPointUndirectedGraph property), 279</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.PointCloud property), 191</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.PointDirectedGraph property), 244</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.PointTree property), 261</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.shape.PointUndirectedGraph property), 226</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.Affine property), 331</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.AlignmentAffine property), 361</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.AlignmentRotation property), 372</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.AlignmentSimilarity property), 366</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.AlignmentTranslation property), 377</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.AlignmentUniformScale property), 381</td>
</tr>
<tr>
<td>n_parameters()</td>
<td>(menpo.transform.Homogeneous property), 349</td>
</tr>
</tbody>
</table>

Index
property), 327
n_parameters() (menpo.transform.NonUniformScale
property), 354
n_parameters() (menpo.transform.Rotation property), 341
n_parameters() (menpo.transform.Similarity property), 336
n_parameters() (menpo.transform.Translation property), 345
n_parameters() (menpo.transform.UniformScale property), 350
n_parents() (menpo.shape.DirectedGraph method), 203
n_parents() (menpo.shape.PointDirectedGraph method), 242
n_parents() (menpo.shape.PointTree method), 259
n_parents() (menpo.shape.Tree method), 208
n_paths() (menpo.shape.DirectedGraph method), 203
n_paths() (menpo.shape.LabelledPointUndirectedGraph
method), 277
n_paths() (menpo.shape.PointDirectedGraph
method), 242
n_paths() (menpo.shape.PointTree method), 259
n_paths() (menpo.shape.PointUndirectedGraph
method), 225
n_paths() (menpo.shape.Tree method), 209
n_paths() (menpo.shape.Un directedGraph method), 196
n_pixels() (menpo.image.BooleanImage property), 67
n_pixels() (menpo.image.Image property), 44
n_pixels() (menpo.image.MaskedImage property), 97
n_points() (menpo.base.Targetable property), 5
n_points() (menpo.shape.ColouredTriMesh property), 307
n_points() (menpo.shape.LabelledPointUndirectedGraph
property), 279
n_points() (menpo.shape.PointCloud property), 191
n_points() (menpo.shape.PointDirectedGraph property), 244
n_points() (menpo.shape.PointTree property), 261
n_points() (menpo.shape.PointUndirectedGraph
property), 226
n_points() (menpo.shape.TexturedTriMesh property), 320
n_points() (menpo.shape.TriMesh property), 294
n_points() (menpo.transform.AlignmentAffine property), 361
n_points() (menpo.transform.AlignmentRotation
property), 372
n_points() (menpo.transform.AlignmentSimilarity
property), 366
n_points() (menpo.transform.AlignmentTranslation
property), 377
n_points() (menpo.transform.AlignmentUniformScale
property), 382
n_points() (menpo.transform.base.alignment.Alignment
property), 394
n_points() (menpo.transform.ThinPlateSplines property), 356
n_tris() (menpo.shape.ColouredTriMesh property), 307
n_tris() (menpo.shape.TexturedTriMesh property), 321
n_tris() (menpo.shape.TriMesh property), 295
n_true() (menpo.image.BooleanImage method), 56
n_true_elements() (menpo.image.MaskedImage method), 84
n_true_pixels() (menpo.image.MaskedImage method), 84
n_vertices() (menpo.shape.DirectedGraph property), 203
n_vertices() (menpo.shape.LabelledPointUndirectedGraph
property), 279
n_vertices() (menpo.shape.PointDirectedGraph
property), 244
n_vertices() (menpo.shape.PointTree property), 261
n_vertices() (menpo.shape.PointUndirectedGraph
property), 226
n_vertices() (menpo.shape.Tree property), 210
n_vertices() (menpo.shape.Un directedGraph property), 197
n_vertices_at_depth() (menpo.shape.PointTree
method), 259
name (menpo.base.MenpoMissingDependencyError attribute), 7
name_of_callable() (in module menpo.base), 7
neighbours() (menpo.shape.LabelledPointUndirectedGraph
method), 277
neighbours() (menpo.shape.PointUndirectedGraph
method), 225
no_op() (in module menpo.feature), 98
noise_variance() (menpo.model.PCAModel
method), 157
noise_variance() (menpo.model.PCAVectorModel
method), 169
noise_variance_ratio() (menpo.model.PCAModel
method), 158
noise_variance_ratio() (menpo.model.PCA VectorModel
method), 169
NonUniformScale (class in menpo.transform), 350
norm() (menpo.shape.ColouredTriMesh property), 305

Index 449
Menpo Documentation, Release 0.9.0

norm() (menpo.shape.LabelledPointUndirectedGraph method), 277
norm() (menpo.shape.PointCloud method), 190
norm() (menpo.shape.PointDirectedGraph method), 242
norm() (menpo.shape.PointTree method), 259
norm() (menpo.shape.TexturedTriMesh method), 318

normalize() (in module menpo.feature), 109
normalize_norm() (in module menpo.feature), 109
normalize_norm() (menpo.image.BooleanImage method), 56
normalize_norm() (menpo.image.MaskedImage method), 84
normalize_std() (menpo.image.BooleanImage method), 56
normalize_std() (menpo.image.Image method), 33
normalize_std() (menpo.image.MaskedImage method), 85
normalize_var() (in module menpo.feature), 110

original_variance() (menpo.model.PCAModel method), 158
original_variance() (menpo.model.PCAVectorModel method), 169
orthonormalize_against_inplace() (menpo.model.LinearVectorModel method), 150
orthonormalize_against_inplace() (menpo.model.MeanLinearVectorModel method), 153
orthonormalize_against_inplace() (menpo.model.PCAModel method), 158
orthonormalize_against_inplace() (menpo.model.PCAVectorModel method), 169
orthonormalize_inplace() (menpo.model.LinearVectorModel method), 150
orthonormalize_inplace() (menpo.model.MeanLinearVectorModel method), 153
orthonormalize_inplace() (menpo.model.PCAModel method), 158
orthonormalize_inplace() (menpo.model.PCAVectorModel method), 169

OutOfMaskSampleError (class in menpo.image), 97

P
parent() (menpo.shape.PointTree method), 259
parent() (menpo.shape.Tree method), 209
parents() (menpo.shape.DirectedGraph method), 203
parents() (menpo.shape.PointDirectedGraph method), 242
parents() (menpo.shape.PointTree method), 259
parents() (menpo.shape.Tree method), 209

OutOfMaskSampleError (class in menpo.image), 97

P

.pickle_paths() (in module menpo.io), 18

PiecewiseAffine (in module menpo.transform), 356

pixels_range() (menpo.image BOOLEAN Image method), 56
pixels_range() (menpo.image.Image method), 33
pixels_range() (menpo.image.MaskedImage method), 85

pixels_with_channels_at_back() (menpo.image.BooleanImage method), 56
pixels_with_channels_at_back() (menpo.image.Image method), 33
pixels_with_channels_at_back() (menpo.image.MaskedImage method), 85

plot_curve() (in module menpo.visualize), 404
plot_eigenvalues() (menpo.model.PCAModel method), 158
plot_eigenvalues() (menpo.model.PCAVectorModel method), 169
plot_eigenvalues_cumulative_ratio() (menpo.model.PCAModel method), 159
plot_eigenvalues_cumulative_ratio() (menpo.model.PCAVectorModel method), 171
plot_eigenvalues_cumulative_ratio_widget() (menpo.model.PCAModel method), 161
plot_eigenvalues_cumulative_ratio_widget() (menpo.model.PCAVectorModel method), 172
plot_eigenvalues_ratio() (menpo.model.PCAModel method), 161
plot_eigenvalues_ratio() (menpo.model.PCAVectorModel method), 172
plot_eigenvalues_ratio_widget() (menpo.model.PCAModel method), 163
plot_eigenvalues_ratio_widget() (menpo.model.PCAVectorModel method), 174
plot_eigenvalues_widget()
(menpo.model.PCAModel method), 163

plot_eigenvalues_widget()
(menpo.model.PCAVectorModel method), 174

plot_gaussian_ellipses()  (in module menpo.visualize), 408

PointCloud (class in menpo.shape), 182
PointDirectedGraph (class in menpo.shape), 226
PointTree (class in menpo.shape), 244
PointUndirectedGraph (class in menpo.shape), 210

pop() (menpo.landmark.LandmarkManager method), 113

popitem()  (menpo.landmark.LandmarkManager method), 113

pose_flic_11_to_pose_flic_11()  (in module menpo.landmark), 132
pose_human36M_32_to_pose_human36M_17()  (in module menpo.landmark), 133
pose_human36M_32_to_pose_human36M_32()  (in module menpo.landmark), 134
pose_lsp_14_to_pose_lsp_14()  (in module menpo.landmark), 135
pose_stickmen_12_to_pose_stickmen_12()  (in module menpo.landmark), 136

principal_components_analysis()
(menpo.model.GMRFModel method), 178
principal_components_analysis()
(menpo.model.GMRFVectorModel method), 180

print_dynamic()  (in module menpo.visualize), 403
print_progress()  (in module menpo.visualize), 402
progress_bar_str()  (in module menpo.visualize), 403

project()  (menpo.model.LinearVectorModel method), 151
project()  (menpo.model.MeanLinearVectorModel method), 153
project()  (menpo.model.PCAModel method), 163
project()  (menpo.model.PCAVectorModel method), 175
project()  (menpo.model.PCAVectorModel method), 175

project_out()  (menpo.model.LinearVectorModel method), 151
project_out()  (menpo.model.MeanLinearVectorModel method), 153
project_out()  (menpo.model.PCAModel method), 163
project_out()  (menpo.model.PCAVectorModel method), 175
project_out()  (menpo.model.PCAVectorModel method), 175

project_out_vector()  (menpo.model.PCAModel method), 164
project_out_vector()  (menpo.model.PCAVectorModel method), 175
project_out_vectors()  (menpo.model.LinearVectorModel method), 151
project_out_vectors()  (menpo.model.MeanLinearVectorModel method), 153
project_out_vectors()  (menpo.model.PCAModel method), 163
project_out_vectors()  (menpo.model.PCAVectorModel method), 175
project_out_vectors()  (menpo.model.PCAVectorModel method), 175

project_out_whitened()  (menpo.model.PCAModel method), 164
project_out_whitened()  (menpo.model.PCAVectorModel method), 175
project_out_whitened_vector()  (menpo.model.PCAModel method), 164
project_out_whitened_vector()  (menpo.model.PCAVectorModel method), 175
proportion_false()  (menpo.image.BooleanImage method), 56
proportion_true()  (menpo.image.BooleanImage method), 56

pseudoinverse()  (menpo.transform.Affine method), 330
pseudoinverse()  (menpo.transform.AlignmentAffine method), 360
pseudoinverse()  (menpo.transform.AlignmentRotation method), 371
pseudoinverse()  (menpo.transform.AlignmentSimilarity method), 365
pseudoinverse()  (menpo.transform.AlignmentTranslation method), 376
pseudoinverse()  (menpo.transform.AlignmentUniformScale method), 380
pseudoinverse()  (menpo.transform.base.invertible.Invertible method), 393
pseudoinverse()  (menpo.transform.base.invertible.VInvertible method), 396
pseudoinverse()  (menpo.transform.Homogeneous method), 326
pseudoinverse()  (menpo.transform.NonUniformScale method), 353
pseudoinverse()  (menpo.transform.Rotation method), 451
Menpo Documentation, Release 0.9.0

Method), 277
render() (menpo.visualize.MatplotlibRenderer method), 398
render() (menpo.visualize.Renderer method), 397
Renderer (class in menpo.visualize), 396
repeat() (menpo.base.LazyList method), 6
rescale() (menpo.image.BooleanImage method), 57
rescale() (menpo.image.Image method), 34
rescale() (menpo.image.MaskedImage method), 86
rescale_landmarks_to_diagonal_range() (menpo.image.BooleanImage method), 58
rescale_landmarks_to_diagonal_range() (menpo.image.Image method), 35
rescale_landmarks_to_diagonal_range() (menpo.image.MaskedImage method), 87
rescale_pixels() (menpo.image.BooleanImage method), 59
rescale_pixels() (menpo.image.Image method), 35
rescale_pixels() (menpo.image.MaskedImage method), 87
rescale_texture() (menpo.shape.ColouredTriMesh method), 305
rescale_texture() (menpo.shape.TexturedTriMesh method), 318
rescale_to_diagonal() (menpo.image.BooleanImage method), 59
rescale_to_diagonal() (menpo.image.Image method), 36
rescale_to_diagonal() (menpo.image.MaskedImage method), 87
rescale_to_pointcloud() (menpo.image.BooleanImage method), 59
rescale_to_pointcloud() (menpo.image.Image method), 36
rescale_to_pointcloud() (menpo.image.MaskedImage method), 88
resize() (menpo.image.BooleanImage method), 60
resize() (menpo.image.Image method), 37
resize() (menpo.image.MaskedImage method), 88
rolled_channels() (menpo.image.BooleanImage method), 60
rolled_channels() (menpo.image.Image method), 37
rolled_channels() (menpo.image.MaskedImage method), 89
rotate_ccw_about_centre() (in module menpo.transform), 322
rotate_ccw_about_centre() (menpo.image.BooleanImage method), 60
rotate_ccw_about_centre() (menpo.image.Image method), 37
rotate_ccw_about_centre() (menpo.image.MaskedImage method), 89
Rotation (class in menpo.transform), 336
rotation_matrix() (menpo.transform.AlignmentRotation property), 372
rotation_matrix() (menpo.transform.Rotation property), 341
save_figure() (menpo.visualize.MatplotlibRenderer method), 398
save_figure() (menpo.visualize.Renderer method), 397
save_figure_widget() (menpo.visualize.MatplotlibRenderer method), 399
Scale() (in module menpo.transform), 345
scale() (menpo.transform.AlignmentUniformScale property), 382
scale() (menpo.transform.NonUniformScale property), 354
scale() (menpo.transform.UniformScale property), 350
scale_about_centre() (in module menpo.transform), 322
set_boundary_pixels() (menpo.image.MaskedImage method), 90
set_h_matrix() (menpo.transform.Affine method), 330
set_h_matrix() (menpo.transform.AlignmentAffine method), 360
set_h_matrix() (menpo.transform.AlignmentRotation method), 371
set_h_matrix() (menpo.transform.AlignmentSimilarity method), 365
set_h_matrix() (menpo.transform.AlignmentTranslation method), 376
set_h_matrix() (menpo.transform.AlignmentUniformScale method), 380
set_h_matrix() (menpo.transform.Homogeneous method), 326
set_h_matrix() (menpo.transform.NonUniformScale method), 353
set_h_matrix() (menpo.transform.Rotation method), 340
set_h_matrix() (menpo.transform.Similarity method), 335
set_h_matrix() (menpo.transform.Translation method), 344
set_h_matrix() (menpo.transform.UniformScale method), 349

S
sample() (menpo.image.BooleanImage method), 61
sample() (menpo.image.Image method), 38
sample() (menpo.image.MaskedImage method), 89
save_figure() (menpo.visualize.MatplotlibRenderer method), 398
save_figure() (menpo.visualize.Renderer method), 397
save_figure_widget() (menpo.visualize.MatplotlibRenderer method), 399
Scale() (in module menpo.transform), 345
scale() (menpo.transform.AlignmentUniformScale property), 382
scale() (menpo.transform.NonUniformScale property), 354
scale() (menpo.transform.UniformScale property), 350
scale_about_centre() (in module menpo.transform), 322
set_boundary_pixels() (menpo.image.MaskedImage method), 90
set_h_matrix() (menpo.transform.Affine method), 330
set_h_matrix() (menpo.transform.AlignmentAffine method), 360
set_h_matrix() (menpo.transform.AlignmentRotation method), 371
set_h_matrix() (menpo.transform.AlignmentSimilarity method), 365
set_h_matrix() (menpo.transform.AlignmentTranslation method), 376
set_h_matrix() (menpo.transform.AlignmentUniformScale method), 380
set_h_matrix() (menpo.transform.Homogeneous method), 326
set_h_matrix() (menpo.transform.NonUniformScale method), 353
set_h_matrix() (menpo.transform.Rotation method), 340
set_h_matrix() (menpo.transform.Similarity method), 335
set_h_matrix() (menpo.transform.Translation method), 344
set_h_matrix() (menpo.transform.UniformScale method), 349

Index 453
set_masked_pixels() (menpo.image.MaskedImage method), 91
set_patches() (menpo.image.BooleanImage method), 61
set_patches() (menpo.image.Image method), 38
set_patches() (menpo.image.MaskedImage method), 91
set_patches_around_landmarks() (menpo.image.BooleanImage method), 62
set_patches_around_landmarks() (menpo.image.Image method), 39
set_rotation_matrix() (menpo.image.MaskedImage method), 91
set_rotation_matrix() (menpo.transform.AlignmentRotation method), 371
set_rotation_matrix() (menpo.transform.Rotation method), 340
set_target() (menpo.base.Targetable method), 5
set_target() (menpo.transform.AlignmentAffine method), 360
set_target() (menpo.transform.AlignmentRotation method), 371
set_target() (menpo.transform.AlignmentSimilarity method), 365
set_target() (menpo.transform.AlignmentTranslation method), 376
set_target() (menpo.transform.AlignmentUniformScale method), 381
set_target() (menpo.transform.base.alignment.Alignment method), 394
set_target() (menpo.transform.ThinPlateSplines property), 356
setdefault() (menpo.landmark.LandmarkManager method), 113
Shape (class in menpo.shape.base), 180
shape() (menpo.image.Image method), 67
shape() (menpo.image.MaskedImage method), 44
shape() (menpo.image.BufferedImage method), 97
shear_about_centre() (in module menpo.transform), 323
Similarity (class in menpo.transform), 332
source() (menpo.transform.AlignmentAffine property), 361
source() (menpo.transform.AlignmentRotation property), 372
source() (menpo.transform.AlignmentSimilarity property), 366
source() (menpo.transform.AlignmentTranslation property), 377
source() (menpo.transform.AlignmentUniformScale property), 382
source() (menpo.transform.base.alignment.Alignment property), 394
source() (menpo.transform.ThinPlateSplines property), 356
sparse_hog() (in module menpo.feature), 107
star_graph() (in module menpo.shape), 279
Stencil_grid() (in module menpo.shape), 281
sum_channels() (in module menpo.feature), 111
T

target() (menpo.base.Targetable property), 5
target() (menpo.transform.AlignmentAffine property), 362
target() (menpo.transform.AlignmentRotation property), 372
target() (menpo.transform.AlignmentSimilarity property), 367
target() (menpo.transform.AlignmentTranslation property), 377
target() (menpo.transform.AlignmentUniformScale property), 382
target() (menpo.transform.base.alignment.Alignment property), 394
target() (menpo.transform.ThinPlateSplines property), 356
Targetable (class in menpo.base), 4	
tcoords_pixel_scaled() (menpo.shape.TexturedTriMesh method), 318
TexturedTriMesh (class in menpo.shape), 307
ThinPlateSplines (class in menpo.transform), 354
tjson() (menpo.shape.ColouredTriMesh method), 306
tjson() (menpo.shape.LabelledPointUndirectedGraph method), 277
tjson() (menpo.shape.PointCloud method), 190
tjson() (menpo.shape.PointDirectedGraph method), 243
tjson() (menpo.shape.PointTree method), 260
tjson() (menpo.shape.PointUndirectedGraph method), 225
tjson() (menpo.shape.TexturedTriMesh method), 319
tjson() (menpo.shape.TriMesh method), 293
tongue_ibug_19_to_tongue_ibug_19() (in module menpo.landmark), 143
Transform (class in menpo.transform), 388
transform_about_centre() (in module menpo.transform), 323
transform_about_centre() (menpo.image.BooleanImage method), 63
transform_about_centre() (menpo.image.Image method), 39
transform_about_centre() (menpo.image.MaskedImage method), 92
Transformable (class in menpo.transform.base), 390
<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransformChain (class in menpo.transform)</td>
<td>383</td>
</tr>
<tr>
<td>Translation (class in menpo.transform)</td>
<td>341</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.Affine property)</td>
<td>331</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.AlignmentAffine property)</td>
<td>362</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.AlignmentRotation property)</td>
<td>373</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.AlignmentSimilarity property)</td>
<td>367</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.AlignmentTranslation property)</td>
<td>377</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.AlignmentUniformScale property)</td>
<td>382</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.NonUniformScale property)</td>
<td>354</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.Rotation property)</td>
<td>341</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.Similarity property)</td>
<td>336</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.Translation property)</td>
<td>345</td>
</tr>
<tr>
<td>translation_component()</td>
<td></td>
</tr>
<tr>
<td>(menpo.transform.UniformScale property)</td>
<td>350</td>
</tr>
<tr>
<td>Tree (class in menpo.shape)</td>
<td>204</td>
</tr>
<tr>
<td>tri_areas()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>tri_areas()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>319</td>
</tr>
<tr>
<td>tri_areas()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>293</td>
</tr>
<tr>
<td>tri_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>tri_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>319</td>
</tr>
<tr>
<td>tri_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>293</td>
</tr>
<tr>
<td>trim_components()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>175</td>
</tr>
<tr>
<td>trim_components()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAModel method)</td>
<td>165</td>
</tr>
<tr>
<td>trim_components()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAVectorModel method)</td>
<td>176</td>
</tr>
<tr>
<td>TriMesh (class in menpo.shape)</td>
<td>283</td>
</tr>
<tr>
<td>true_indices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.image.BooleanImage method)</td>
<td>65</td>
</tr>
<tr>
<td>UniformScale (class in menpo.transform)</td>
<td>346</td>
</tr>
<tr>
<td>unique_edge_indices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>unique_edge_indices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>319</td>
</tr>
<tr>
<td>unique_edge_indices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>293</td>
</tr>
<tr>
<td>unique_edge_lengths()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>unique_edge_lengths()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>319</td>
</tr>
<tr>
<td>unique_edge_lengths()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>293</td>
</tr>
<tr>
<td>unique_edge_vectors()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>unique_edge_vectors()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>320</td>
</tr>
<tr>
<td>unique_edge_vectors()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>294</td>
</tr>
<tr>
<td>update()</td>
<td></td>
</tr>
<tr>
<td>(menpo.landmark.LandmarkManager method)</td>
<td>113</td>
</tr>
<tr>
<td>values()</td>
<td></td>
</tr>
<tr>
<td>(menpo.landmark.LandmarkManager method)</td>
<td>113</td>
</tr>
<tr>
<td>variance()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAModel method)</td>
<td>165</td>
</tr>
<tr>
<td>variance()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAVectorModel method)</td>
<td>175</td>
</tr>
<tr>
<td>variance_ratio()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAModel method)</td>
<td>165</td>
</tr>
<tr>
<td>variance_ratio()</td>
<td></td>
</tr>
<tr>
<td>(menpo.model.PCAVectorModel method)</td>
<td>176</td>
</tr>
<tr>
<td>VComposable</td>
<td></td>
</tr>
<tr>
<td>(class in menpo.transform.base.composable)</td>
<td>395</td>
</tr>
<tr>
<td>vector_128_dsift()</td>
<td></td>
</tr>
<tr>
<td>(in module menpo.feature)</td>
<td>106</td>
</tr>
<tr>
<td>Vectorizable (class in menpo.base)</td>
<td>3</td>
</tr>
<tr>
<td>vertex_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.ColouredTriMesh method)</td>
<td>306</td>
</tr>
<tr>
<td>vertex_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TexturedTriMesh method)</td>
<td>320</td>
</tr>
<tr>
<td>vertex_normals()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.TriMesh method)</td>
<td>294</td>
</tr>
<tr>
<td>vertices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.DirectedGraph property)</td>
<td>203</td>
</tr>
<tr>
<td>vertices()</td>
<td></td>
</tr>
<tr>
<td>(menpo.shape.LabelledPointUndirectedGraph property)</td>
<td>279</td>
</tr>
</tbody>
</table>
vertices() (menpo.shape.PointDirectedGraph property), 244
vertices() (menpo.shape.PointTree property), 261
vertices() (menpo.shape.PointUndirectedGraph property), 226
vertices() (menpo.shape.Tree property), 210
vertices() (menpo.shape.UndirectedGraph property), 197
vertices_at_depth() (menpo.shape.PointTree method), 260
vertices_at_depth() (menpo.shape.Tree method), 209
video_paths() (in module menpo.io), 18
view_patches() (in module menpo.visualize), 399
view_widget() (menpo.base.LazyList method), 6
view_widget() (menpo.image.BooleanImage method), 65
view_widget() (menpo.image.Image method), 42
view_widget() (menpo.image.MaskedImage method), 94
view_widget() (menpo.model.PCAModel method), 165
Viewable (class in menpo.visualize), 397
VInvertible (class in menpo.transform.base.invertible), 395

W
warp_to_mask() (menpo.image.BooleanImage method), 65
warp_to_mask() (menpo.image.Image method), 42
warp_to_mask() (menpo.image.MaskedImage method), 94
warp_to_shape() (menpo.image.BooleanImage method), 65
warp_to_shape() (menpo.image.Image method), 42
warp_to_shape() (menpo.image.MaskedImage method), 95
whitened_components() (menpo.model.PCAModel method), 165
whitened_components() (menpo.model.PCAVectorModel method), 176
width() (menpo.image.BooleanImage property), 67
width() (menpo.image.Image property), 44
width() (menpo.image.MaskedImage property), 97
with_dims() (menpo.shape.ColouredTriMesh method), 307
with_dims() (menpo.shape.LabelledPointUndirectedGraph method), 225
with_dims() (menpo.shape.TexturedTriMesh method), 320
with_dims() (menpo.shape.TriMesh method), 294
with_labels() (menpo.shape.LabelledPointUndirectedGraph method), 278
with_traceback() (menpo.base.MenpoDeprecationWarning method), 7
with_traceback() (menpo.base.MenpoMissingDependencyError method), 7
with_traceback() (menpo.image.ImageBoundaryError method), 97
with_traceback() (menpo.imageOutOfBoundsSampleError method), 97
with_traceback() (menpo.landmark.LabelError method), 112
without_labels() (menpo.shape.LabelledPointUndirectedGraph method), 278

Z
zoom() (menpo.image.BooleanImage method), 66
zoom() (menpo.image.Image method), 43
zoom() (menpo.image.MaskedImage method), 96