Module: transform

skimage.transform.downscale_local_mean (...)	Down-sample N-dimensional image by local averaging.
skimage.transform.estimate_transform (ttype, ...)	Estimate 2D geometric transformation parameters.
skimage.transform.frt2 (a)	Compute the 2-dimensional finite radon transform (FRT) for an n x n integer array.
skimage.transform.hough_circle (image, radius)	Perform a circular Hough transform.
skimage.transform.hough_circle_peaks (...[, ...])	Return peaks in a circle Hough transform.
skimage.transform.hough_ellipse (image[, ...])	Perform an elliptical Hough transform.
skimage.transform.hough_line (image[, theta])	Perform a straight line Hough transform.
skimage.transform.hough_line_peaks (hspace, ...)	Return peaks in a straight line Hough transform.
skimage.transform.ifrt2 (a)	Compute the 2-dimensional inverse finite radon transform (iFRT) for an (n+1) x n integer array.
skimage.transform.integral_image (image)	Integral image / summed area table.
skimage.transform.integrate (ii, start, end)	Use an integral image to integrate over a given window.
skimage.transform.iradon (radon_image[, ...])	Inverse radon transform.
skimage.transform.iradon_sart (radon_image[, ...])	Inverse radon transform
skimage.transform.matrix_transform (coords, ...)	Apply 2D matrix transform.
skimage.transform.order_angles_golden_ratio (theta)	Order angles to reduce the amount of correlated information in subsequent projections.
skimage.transform.probabilistic_hough_line (image)	Return lines from a progressive probabilistic line Hough transform.
skimage.transform.pyramid_expand (image[, ...])	Upsample and then smooth image.
skimage.transform.pyramid_gaussian (image[, ...])	Yield images of the Gaussian pyramid formed by the input image.
skimage.transform.pyramid_laplacian (image[, ...])	Yield images of the laplacian pyramid formed by the input image.
skimage.transform.pyramid_reduce (image[, ...])	Smooth and then downsample image.
skimage.transform.radon (image[, theta, circle])	Calculates the radon transform of an image given specified projection angles.
skimage.transform.rescale (image, scale[, ...])	Scale image by a certain factor.
skimage.transform.resize (image, output_shape)	Resize image to match a certain size.
skimage.transform.rotate (image, angle[, ...])	Rotate image by a certain angle around its center.
skimage.transform.seam_carve (image, ...[, ...])	Carve vertical or horizontal seams off an image.
skimage.transform.swirl (image[, center, ...])	Perform a swirl transformation.
skimage.transform.warp (image, inverse_map[, ...])	Warp an image according to a given coordinate transformation.

skimage.transform.warp_coords (coord_map, shape)	Build the source coordinates for the output of a 2-D image warp.
skimage.transform.AffineTransform ([matrix, ...])	2D affine transformation of the form:
skimage.transform.EssentialMatrixTransform ([...])	Essential matrix transformation.
skimage.transform.EuclideanTransform ([...])	2D Euclidean transformation of the form:
skimage.transform.FundamentalMatrixTransform ([...])	Fundamental matrix transformation.
skimage.transform.PiecewiseAffineTransform ()	2D piecewise affine transformation.
skimage.transform.PolynomialTransform ([params])	2D polynomial transformation of the form:
skimage.transform.ProjectiveTransform ([matrix])	Projective transformation.
skimage.transform.SimilarityTransform ([...])	2D similarity transformation of the form:

downscale_local_mean

skimage.transform.**downscale_local_mean**(*image*, *factors*, *cval=0*, *clip=True*) [source]

Down-sample N-dimensional image by local averaging.

The image is padded with cval if it is not perfectly divisible by the integer factors.

In contrast to the 2-D interpolation in skimage.transform.resize and skimage.transform.rescale this function may be applied to N-dimensional images and calculates the local mean of elements in each block of size factors in the input image.

Parameters:

image : ndarray
N-dimensional input image.

factors : array_like
Array containing down-sampling integer factor along each axis.

cval : float, optional
Constant padding value if image is not perfectly divisible by the integer factors.

Returns:

image : ndarray
Down-sampled image with same number of dimensions as input image.

Examples

```
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> downscale_local_mean(a, (2, 3))
array([[ 3.5,  4. ],
       [ 5.5,  4.5]])
```

estimate_transform

skimage.transform.**estimate_transform**(*ttype*, *src*, *dst*, ***kwargs*) [source]

Estimate 2D geometric transformation parameters.

You can determine the over-, well- and under-determined parameters with the total least-squares method.

Number of source and destination coordinates must match.

Parameters:

ttype : {'euclidean', similarity', 'affine', 'piecewise-affine', 'projective', 'polynomial'}
Type of transform.

kwargs : array or int

Function parameters (src, dst, n, angle):

```
NAME / TTYPE        FUNCTION PARAMETERS
'euclidean'         `src, `dst`
'similarity'        `src, `dst`
'affine'            `src, `dst`
'piecewise-affine'  `src, `dst`
'projective'        `src, `dst`
'polynomial'        `src, `dst`, `order` (polynomial orde
r,
                                          default order i
s 2)
```

Also see examples below.

Returns: **tform** : `GeometricTransform`

Transform object containing the transformation parameters and providing access to forward and inverse transformation functions.

Examples

```
>>> import numpy as np
>>> from skimage import transform as tf
```

```
>>> # estimate transformation parameters
>>> src = np.array([0, 0, 10, 10]).reshape((2, 2))
>>> dst = np.array([12, 14, 1, -20]).reshape((2, 2))
```

```
>>> tform = tf.estimate_transform('similarity', src, dst)
```

```
>>> np.allclose(tform.inverse(tform(src)), src)
True
```

```
>>> # warp image using the estimated transformation
>>> from skimage import data
>>> image = data.camera()
```

```
>>> warp(image, inverse_map=tform.inverse)
```

```
>>> # create transformation with explicit parameters
>>> tform2 = tf.SimilarityTransform(scale=1.1, rotation=1,
...     translation=(10, 20))
```

```
>>> # unite transformations, applied in order from left to right
>>> tform3 = tform + tform2
>>> np.allclose(tform3(src), tform2(tform(src)))
True
```

frt2

`skimage.transform.` **`frt2`**(*a*) [source]

Compute the 2-dimensional finite radon transform (FRT) for an n x n integer array.

Parameters: **a** : array_like

A 2-D square n x n integer array.

Returns: **FRT** : 2-D ndarray

Finite Radon Transform array of (n+1) x n integer coefficients.

See also

`ifrt2`

The two-dimensional inverse FRT.

Notes

The FRT has a unique inverse if and only if n is prime. [FRT] The idea for this algorithm is due to Vlad Negnevitski.

References

[FRT] A. Kingston and I. Svalbe, "Projective transforms on periodic discrete image arrays," in P. Hawkes (Ed), Advances in Imaging and Electron Physics, 139 (2006)

Examples

Generate a test image: Use a prime number for the array dimensions

```
>>> SIZE = 59
>>> img = np.tri(SIZE, dtype=np.int32)
```

Apply the Finite Radon Transform:

```
>>> f = frt2(img)
```

hough_circle

`skimage.transform.` **`hough_circle`**(*image*, *radius*, *normalize=True*, *full_output=False*) [source]

Perform a circular Hough transform.

Parameters:

image : (M, N) ndarray
Input image with nonzero values representing edges.

radius : scalar or sequence of scalars
Radii at which to compute the Hough transform. Floats are converted to integers.

normalize : boolean, optional (default True)
Normalize the accumulator with the number of pixels used to draw the radius.

full_output : boolean, optional (default False)
Extend the output size by twice the largest radius in order to detect centers outside the input picture.

Returns:

H : 3D ndarray (radius index, (M + 2R, N + 2R) ndarray)
Hough transform accumulator for each radius. R designates the larger radius if full_output is True. Otherwise, R = 0.

Examples

```
>>> from skimage.transform import hough_circle
>>> from skimage.draw import circle_perimeter
>>> img = np.zeros((100, 100), dtype=np.bool_)
>>> rr, cc = circle_perimeter(25, 35, 23)
>>> img[rr, cc] = 1
>>> try_radii = np.arange(5, 50)
>>> res = hough_circle(img, try_radii)
>>> ridx, r, c = np.unravel_index(np.argmax(res), res.shape)
>>> r, c, try_radii[ridx]
(25, 35, 23)
```

hough_circle_peaks

`skimage.transform.` **`hough_circle_peaks`**(*hspaces*, *radii*, *min_xdistance=1*, *min_ydistance=1*,

threshold=None, *num_peaks=inf*, *total_num_peaks=inf*, *normalize=False*) [source]

Return peaks in a circle Hough transform.

Identifies most prominent circles separated by certain distances in a Hough space. Non-maximum suppression with different sizes is applied separately in the first and second dimension of the Hough space to identify peaks.

Parameters:

hspaces : (N, M) array
Hough spaces returned by the hough_circle function.

radii : (M,) array
Radii corresponding to Hough spaces.

min_xdistance : int, optional
Minimum distance separating centers in the x dimension.

min_ydistance : int, optional
Minimum distance separating centers in the y dimension.

threshold : float, optional
Minimum intensity of peaks in each Hough space. Default is 0.5 * max(hspace).

num_peaks : int, optional
Maximum number of peaks in each Hough space. When the number of peaks exceeds num_peaks, only num_peaks coordinates based on peak intensity are considered for the corresponding radius.

total_num_peaks : int, optional
Maximum number of peaks. When the number of peaks exceeds num_peaks, return num_peaks coordinates based on peak intensity.

normalize : bool, optional
If True, normalize the accumulator by the radius to sort the prominent peaks.

Returns:

accum, cx, cy, rad : tuple of array
Peak values in Hough space, x and y center coordinates and radii.

Examples

```
>>> from skimage import transform as tf
>>> from skimage import draw
>>> img = np.zeros((120, 100), dtype=int)
>>> radius, x_0, y_0 = (20, 99, 50)
>>> y, x = draw.circle_perimeter(y_0, x_0, radius)
>>> img[x, y] = 1
>>> hspaces = tf.hough_circle(img, radius)
>>> accum, cx, cy, rad = hough_circle_peaks(hspaces, [radius,])
```

hough_ellipse

skimage.transform. **hough_ellipse** (*image*, *threshold=4*, *accuracy=1*, *min_size=4*, *max_size=None*) [source]

Perform an elliptical Hough transform.

Parameters:

image : (M, N) ndarray
Input image with nonzero values representing edges.

threshold: int, optional
Accumulator threshold value.

accuracy : double, optional
Bin size on the minor axis used in the accumulator.

min_size : int, optional
Minimal major axis length.

max_size : int, optional
Maximal minor axis length. If None, the value is set to the half of the smaller image dimension.

Returns: **result** : ndarray with fields [(accumulator, y0, x0, a, b, orientation)]

Where `(yc, xc)` is the center, `(a, b)` the major and minor axes, respectively. The orientation value follows skimage.draw.ellipse_perimeter convention.

Notes

The accuracy must be chosen to produce a peak in the accumulator distribution. In other words, a flat accumulator distribution with low values may be caused by a too low bin size.

References

[R471] Xie, Yonghong, and Qiang Ji. "A new efficient ellipse detection method." Pattern Recognition, 2002. Proceedings. 16th International Conference on. Vol. 2. IEEE, 2002

Examples

```
>>> from skimage.transform import hough_ellipse
>>> from skimage.draw import ellipse_perimeter
>>> img = np.zeros((25, 25), dtype=np.uint8)
>>> rr, cc = ellipse_perimeter(10, 10, 6, 8)
>>> img[cc, rr] = 1
>>> result = hough_ellipse(img, threshold=8)
>>> result.tolist()
[(10, 10.0, 10.0, 8.0, 6.0, 0.0)]
```

hough_line

`skimage.transform.` **`hough_line`**(*image*, *theta=None*) [source]

Perform a straight line Hough transform.

Parameters: **image** : (M, N) ndarray

Input image with nonzero values representing edges.

theta : 1D ndarray of double, optional

Angles at which to compute the transform, in radians. Defaults to a vector of 180 angles evenly spaced from -pi/2 to pi/2.

Returns: **hspace** : 2-D ndarray of uint64

Hough transform accumulator.

angles : ndarray

Angles at which the transform is computed, in radians.

distances : ndarray

Distance values.

Notes

The origin is the top left corner of the original image. X and Y axis are horizontal and vertical edges respectively. The distance is the minimal algebraic distance from the origin to the detected line. The angle accuracy can be improved by decreasing the step size in the theta array.

Examples

Generate a test image:

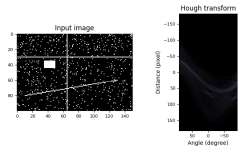
```
>>> img = np.zeros((100, 150), dtype=bool)
>>> img[30, :] = 1
>>> img[:, 65] = 1
>>> img[35:45, 35:50] = 1
>>> for i in range(90):
...     img[i, i] = 1
>>> img += np.random.random(img.shape) > 0.95
```

Apply the Hough transform:

```
>>> out, angles, d = hough_line(img)
```

```
import numpy as np
import matplotlib.pyplot as plt

from skimage.transform import hough_line
from skimage.draw import line

img = np.zeros((100, 150), dtype=bool)
img[30, :] = 1
img[:, 65] = 1
img[35:45, 35:50] = 1
rr, cc = line(60, 130, 80, 10)
img[rr, cc] = 1
img += np.random.random(img.shape) > 0.95

out, angles, d = hough_line(img)

fix, axes = plt.subplots(1, 2, figsize=(7, 4))

axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Input image')

axes[1].imshow(
    out, cmap=plt.cm.bone,
    extent=(np.rad2deg(angles[-1]), np.rad2deg(angles[0]), d[-1], d[0]))
axes[1].set_title('Hough transform')
axes[1].set_xlabel('Angle (degree)')
axes[1].set_ylabel('Distance (pixel)')

plt.tight_layout()
plt.show()
```

(Source code, png, pdf)



hough_line_peaks

skimage.transform. **hough_line_peaks**(*hspace*, *angles*, *dists*, *min_distance=9*, *min_angle=10*, *threshold=None*, *num_peaks=inf*) [source]

Return peaks in a straight line Hough transform.

Identifies most prominent lines separated by a certain angle and distance in a Hough transform. Non-maximum suppression with different sizes is applied separately in the first (distances) and second (angles) dimension of the Hough space to identify peaks.

Parameters:

hspace : (N, M) array

Hough space returned by the hough_line function.

angles : (M,) array

Angles returned by the hough_line function. Assumed to be continuous. (angles[-1] - angles[0] == PI).

dists : (N,) array

Distances returned by the hough_line function.

min_distance : int, optional

Minimum distance separating lines (maximum filter size for first dimension of hough space).

min_angle : int, optional

Minimum angle separating lines (maximum filter size for second dimension of

hough space).

threshold : float, optional

Minimum intensity of peaks. Default is 0.5 * max(hspace).

num_peaks : int, optional

Maximum number of peaks. When the number of peaks exceeds num_peaks, return num_peaks coordinates based on peak intensity.

Returns: **accum, angles, dists** : tuple of array

Peak values in Hough space, angles and distances.

Examples

```
>>> from skimage.transform import hough_line, hough_line_peaks
>>> from skimage.draw import line
>>> img = np.zeros((15, 15), dtype=np.bool_)
>>> rr, cc = line(0, 0, 14, 14)
>>> img[rr, cc] = 1
>>> rr, cc = line(0, 14, 14, 0)
>>> img[cc, rr] = 1
>>> hspace, angles, dists = hough_line(img)
>>> hspace, angles, dists = hough_line_peaks(hspace, angles, dists)
>>> len(angles)
2
```

ifrt2

skimage.transform. **ifrt2** (*a*) [source]

Compute the 2-dimensional inverse finite radon transform (iFRT) for an (n+1) x n integer array.

Parameters: **a** : array_like

A 2-D (n+1) row x n column integer array.

Returns: **iFRT** : 2-D n x n ndarray

Inverse Finite Radon Transform array of n x n integer coefficients.

See also

frt2

The two-dimensional FRT

Notes

The FRT has a unique inverse if and only if n is prime. See [R472] for an overview. The idea for this algorithm is due to Vlad Negnevitski.

References

[R472] (1, 2) A. Kingston and I. Svalbe, “Projective transforms on periodic discrete image arrays,” in P. Hawkes (Ed), Advances in Imaging and Electron Physics, 139 (2006)

Examples

```
>>> SIZE = 59
>>> img = np.tri(SIZE, dtype=np.int32)
```

Apply the Finite Radon Transform:

```
>>> f = frt2(img)
```

Apply the Inverse Finite Radon Transform to recover the input

```
>>> fi = ifrt2(f)
```

Check that it's identical to the original

```
>>> assert len(np.nonzero(img-fi)[0]) == 0
```

integral_image

skimage.transform. **integral_image** (*image*) [source]

Integral image / summed area table.

The integral image contains the sum of all elements above and to the left of it, i.e.:

$$S[m, n] = \sum_{i \leq m} \sum_{j \leq n} X[i, j]$$

Parameters:

image : ndarray
Input image.

Returns:

S : ndarray
Integral image/summed area table of same shape as input image.

References

[R473] F.C. Crow, "Summed-area tables for texture mapping," ACM SIGGRAPH Computer Graphics, vol. 18, 1984, pp. 207-212.

integrate

skimage.transform. **integrate** (*ii*, *start*, *end*) [source]

Use an integral image to integrate over a given window.

Parameters:

ii : ndarray
Integral image.

start : List of tuples, each tuple of length equal to dimension of ii
Coordinates of top left corner of window(s). Each tuple in the list contains the starting row, col, ... index i.e [(row_win1, col_win1, ...), (row_win2, col_win2,...), ...].

end : List of tuples, each tuple of length equal to dimension of ii
Coordinates of bottom right corner of window(s). Each tuple in the list containing the end row, col, ... index i.e [(row_win1, col_win1, ...), (row_win2, col_win2, ...), ...].

Returns:

S : scalar or ndarray
Integral (sum) over the given window(s).

Examples

```
>>> arr = np.ones((5, 6), dtype=np.float)
>>> ii = integral_image(arr)
>>> integrate(ii, (1, 0), (1, 2))  # sum from (1, 0) to (1, 2)
array([ 3.])
>>> integrate(ii, [(3, 3)], [(4, 5)])  # sum from (3, 3) to (4, 5)
array([ 6.])
>>> # sum from (1, 0) to (1, 2) and from (3, 3) to (4, 5)
>>> integrate(ii, [(1, 0), (3, 3)], [(1, 2), (4, 5)])
array([ 3.,  6.])
```

iradon

skimage.transform. **iradon**(*radon_image*, *theta=None*, *output_size=None*, *filter='ramp'*, *interpolation='linear'*, *circle=None*) [source]

Inverse radon transform.

Reconstruct an image from the radon transform, using the filtered back projection algorithm.

Parameters:

radon_image : array_like, dtype=float

Image containing radon transform (sinogram). Each column of the image corresponds to a projection along a different angle. The tomography rotation axis should lie at the pixel index `radon_image.shape[0] // 2` along the 0th dimension of `radon_image`.

theta : array_like, dtype=float, optional

Reconstruction angles (in degrees). Default: m angles evenly spaced between 0 and 180 (if the shape of radon_image is (N, M)).

output_size : int

Number of rows and columns in the reconstruction.

filter : str, optional (default ramp)

Filter used in frequency domain filtering. Ramp filter used by default. Filters available: ramp, shepp-logan, cosine, hamming, hann. Assign None to use no filter.

interpolation : str, optional (default 'linear')

Interpolation method used in reconstruction. Methods available: 'linear', 'nearest', and 'cubic' ('cubic' is slow).

circle : boolean, optional

Assume the reconstructed image is zero outside the inscribed circle. Also changes the default output_size to match the behaviour of `radon` called with `circle=True`. The default behavior (None) is equivalent to False.

Returns:

reconstructed : ndarray

Reconstructed image. The rotation axis will be located in the pixel with indices `(reconstructed.shape[0] // 2, reconstructed.shape[1] // 2)`.

Notes

It applies the Fourier slice theorem to reconstruct an image by multiplying the frequency domain of the filter with the FFT of the projection data. This algorithm is called filtered back projection.

References

[R474] AC Kak, M Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press 1988.

[R475] B.R. Ramesh, N. Srinivasa, K. Rajgopal, "An Algorithm for Computing the Discrete Radon Transform With Some Applications", Proceedings of the Fourth IEEE Region 10 International Conference, TENCON '89, 1989

iradon_sart

skimage.transform. **iradon_sart**(*radon_image*, *theta=None*, *image=None*, *projection_shifts=None*, *clip=None*, *relaxation=0.15*) [source]

Inverse radon transform

Reconstruct an image from the radon transform, using a single iteration of the Simultaneous Algebraic Reconstruction Technique (SART) algorithm.

Parameters:

radon_image : 2D array, dtype=float

Image containing radon transform (sinogram). Each column of the image corresponds to a projection along a different angle. The tomography rotation axis should lie at the pixel index `radon_image.shape[0] // 2` along the 0th dimension of `radon_image`.

theta : 1D array, dtype=float, optional

Reconstruction angles (in degrees). Default: m angles evenly spaced between 0 and

180 (if the shape of radon_image is (N, M)).

image : 2D array, dtype=float, optional

Image containing an initial reconstruction estimate. Shape of this array should be `(radon_image.shape[0], radon_image.shape[0])`. The default is an array of zeros.

projection_shifts : 1D array, dtype=float

Shift the projections contained in `radon_image` (the sinogram) by this many pixels before reconstructing the image. The i'th value defines the shift of the i'th column of `radon_image`.

clip : length-2 sequence of floats

Force all values in the reconstructed tomogram to lie in the range `[clip[0], clip[1]]`

relaxation : float

Relaxation parameter for the update step. A higher value can improve the convergence rate, but one runs the risk of instabilities. Values close to or higher than 1 are not recommended.

Returns: **reconstructed** : ndarray

Reconstructed image. The rotation axis will be located in the pixel with indices `(reconstructed.shape[0] // 2, reconstructed.shape[1] // 2)`.

Notes

Algebraic Reconstruction Techniques are based on formulating the tomography reconstruction problem as a set of linear equations. Along each ray, the projected value is the sum of all the values of the cross section along the ray. A typical feature of SART (and a few other variants of algebraic techniques) is that it samples the cross section at equidistant points along the ray, using linear interpolation between the pixel values of the cross section. The resulting set of linear equations are then solved using a slightly modified Kaczmarz method.

When using SART, a single iteration is usually sufficient to obtain a good reconstruction. Further iterations will tend to enhance high-frequency information, but will also often increase the noise.

References

[R476] AC Kak, M Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press 1988.

[R477] AH Andersen, AC Kak, "Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm", Ultrasonic Imaging 6 pp 81–94 (1984)

[R478] S Kaczmarz, "Angenäherte auflösung von systemen linearer gleichungen", Bulletin International de l'Academie Polonaise des Sciences et des Lettres 35 pp 355–357 (1937)

[R479] Kohler, T. "A projection access scheme for iterative reconstruction based on the golden section." Nuclear Science Symposium Conference Record, 2004 IEEE. Vol. 6. IEEE, 2004.

[R480] Kaczmarz' method, Wikipedia, http://en.wikipedia.org/wiki/Kaczmarz_method

matrix_transform

`skimage.transform.` **`matrix_transform`**(*coords*, *matrix*) [source]

Apply 2D matrix transform.

Parameters: **coords** : (N, 2) array

x, y coordinates to transform

matrix : (3, 3) array

Homogeneous transformation matrix.

Returns: **coords** : (N, 2) array
Transformed coordinates.

order_angles_golden_ratio

`skimage.transform.` **`order_angles_golden_ratio`**(*theta*) [source]

Order angles to reduce the amount of correlated information in subsequent projections.

Parameters: **theta** : 1D array of floats
Projection angles in degrees. Duplicate angles are not allowed.

Returns: **indices_generator** : generator yielding unsigned integers
The returned generator yields indices into `theta` such that `theta[indices]` gives the approximate golden ratio ordering of the projections. In total, `len(theta)` indices are yielded. All non-negative integers < `len(theta)` are yielded exactly once.

Notes

The method used here is that of the golden ratio introduced by T. Kohler.

References

[R481] Kohler, T. "A projection access scheme for iterative reconstruction based on the golden section." Nuclear Science Symposium Conference Record, 2004 IEEE. Vol. 6. IEEE, 2004.

[R482] Winkelmann, Stefanie, et al. "An optimal radial profile order based on the Golden Ratio for time-resolved MRI." Medical Imaging, IEEE Transactions on 26.1 (2007): 68-76.

probabilistic_hough_line

`skimage.transform.` **`probabilistic_hough_line`**(*image*, *threshold=10*, *line_length=50*, *line_gap=10*, *theta=None*) [source]

Return lines from a progressive probabilistic line Hough transform.

Parameters: **image** : (M, N) ndarray
Input image with nonzero values representing edges.

threshold : int, optional
Threshold

line_length : int, optional
Minimum accepted length of detected lines. Increase the parameter to extract longer lines.

line_gap : int, optional
Maximum gap between pixels to still form a line. Increase the parameter to merge broken lines more aggresively.

theta : 1D ndarray, dtype=double, optional
Angles at which to compute the transform, in radians. If None, use a range from -pi/2 to pi/2.

Returns: **lines** : list
List of lines identified, lines in format ((x0, y0), (x1, y1)), indicating line start and end.

References

[R483] C. Galamhos, J. Matas and J. Kittler, "Progressive probabilistic Hough transform for line detection", in IEEE Computer Society Conference on Computer Vision and

Pattern Recognition, 1999.

pyramid_expand

skimage.transform.**pyramid_expand**(*image*, *upscale=2*, *sigma=None*, *order=1*, *mode='reflect'*, *cval=0*, *multichannel=None*) [source]

Upsample and then smooth image.

Parameters:

image : ndarray
Input image.

upscale : float, optional
Upscale factor.

sigma : float, optional
Sigma for Gaussian filter. Default is 2 * upscale / 6.0 which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the Gaussian distribution.

order : int, optional
Order of splines used in interpolation of upsampling. See skimage.transform.warp for detail.

mode : {'reflect', 'constant', 'edge', 'symmetric', 'wrap'}, optional
The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

cval : float, optional
Value to fill past edges of input if mode is 'constant'.

multichannel : bool, optional
Whether the last axis of the image is to be interpreted as multiple channels or another spatial dimension. By default, is set to True for 3D (2D+color) inputs, and False for others. Starting in release 0.16, this will always default to False.

Returns:

out : array
Upsampled and smoothed float image.

References

[R484] http://web.mit.edu/persci/people/adelson/pub_pdfs/pyramid83.pdf

pyramid_gaussian

skimage.transform.**pyramid_gaussian**(*image*, *max_layer=-1*, *downscale=2*, *sigma=None*, *order=1*, *mode='reflect'*, *cval=0*, *multichannel=None*) [source]

Yield images of the Gaussian pyramid formed by the input image.

Recursively applies the pyramid_reduce function to the image, and yields the downscaled images.

Note that the first image of the pyramid will be the original, unscaled image. The total number of images is max_layer + 1. In case all layers are computed, the last image is either a one-pixel image or the image where the reduction does not change its shape.

Parameters:

image : ndarray
Input image.

max_layer : int
Number of layers for the pyramid. 0th layer is the original image. Default is -1 which builds all possible layers.

downscale : float, optional
Downscale factor.

sigma : float, optional
Sigma for Gaussian filter. Default is 2 * downscale / 6.0 which corresponds to a

filter mask twice the size of the scale factor that covers more than 99% of the Gaussian distribution.

order : int, optional

Order of splines used in interpolation of downsampling. See skimage.transform.warp for detail.

mode : {'reflect', 'constant', 'edge', 'symmetric', 'wrap'}, optional

The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

cval : float, optional

Value to fill past edges of input if mode is 'constant'.

multichannel : bool, optional

Whether the last axis of the image is to be interpreted as multiple channels or another spatial dimension. By default, is set to True for 3D (2D+color) inputs, and False for others. Starting in release 0.16, this will always default to False.

Returns:

pyramid : generator

Generator yielding pyramid layers as float images.

References

[R485] http://web.mit.edu/persci/people/adelson/pub_pdfs/pyramid83.pdf

pyramid_laplacian

`skimage.transform.` **`pyramid_laplacian`**(*image*, *max_layer=-1*, *downscale=2*, *sigma=None*, *order=1*, *mode='reflect'*, *cval=0*, *multichannel=None*) [source]

Yield images of the laplacian pyramid formed by the input image.

Each layer contains the difference between the downsampled and the downsampled, smoothed image:

```
layer = resize(prev_layer) - smooth(resize(prev_layer))
```

Note that the first image of the pyramid will be the difference between the original, unscaled image and its smoothed version. The total number of images is max_layer + 1. In case all layers are computed, the last image is either a one-pixel image or the image where the reduction does not change its shape.

Parameters:

image : ndarray

Input image.

max_layer : int

Number of layers for the pyramid. 0th layer is the original image. Default is -1 which builds all possible layers.

downscale : float, optional

Downscale factor.

sigma : float, optional

Sigma for Gaussian filter. Default is 2 * downscale / 6.0 which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the Gaussian distribution.

order : int, optional

Order of splines used in interpolation of downsampling. See skimage.transform.warp for detail.

mode : {'reflect', 'constant', 'edge', 'symmetric', 'wrap'}, optional

The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

cval : float, optional

Value to fill past edges of input if mode is 'constant'.

multichannel : bool, optional

Whether the last axis of the image is to be interpreted as multiple channels or

another spatial dimension. By default, is set to True for 3D (2D+color) inputs, and False for others. Starting in release 0.16, this will always default to False.

Returns: **pyramid** : generator

Generator yielding pyramid layers as float images.

References

[R486] http://web.mit.edu/persci/people/adelson/pub_pdfs/pyramid83.pdf

[R487] http://sepwww.stanford.edu/data/media/public/sep/morgan/texturematch/paper_html/node3.html

pyramid_reduce

`skimage.transform.` **`pyramid_reduce`**(*image*, *downscale=2*, *sigma=None*, *order=1*, *mode='reflect'*, *cval=0*, *multichannel=None*) [source]

Smooth and then downsample image.

Parameters:

image : ndarray

Input image.

downscale : float, optional

Downscale factor.

sigma : float, optional

Sigma for Gaussian filter. Default is 2 * downscale / 6.0 which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the Gaussian distribution.

order : int, optional

Order of splines used in interpolation of downsampling. See skimage.transform.warp for detail.

mode : {'reflect', 'constant', 'edge', 'symmetric', 'wrap'}, optional

The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

cval : float, optional

Value to fill past edges of input if mode is 'constant'.

multichannel : bool, optional

Whether the last axis of the image is to be interpreted as multiple channels or another spatial dimension. By default, is set to True for 3D (2D+color) inputs, and False for others. Starting in release 0.16, this will always default to False.

Returns: **out** : array

Smoothed and downsampled float image.

References

[R488] http://web.mit.edu/persci/people/adelson/pub_pdfs/pyramid83.pdf

radon

`skimage.transform.` **`radon`**(*image*, *theta=None*, *circle=None*) [source]

Calculates the radon transform of an image given specified projection angles.

Parameters:

image : array_like, dtype=float

Input image. The rotation axis will be located in the pixel with indices `(image.shape[0] // 2, image.shape[1] // 2)`.

theta : array_like, dtype=float, optional (default np.arange(180))

Projection angles (in degrees).

circle : boolean, optional

Assume image is zero outside the inscribed circle, making the width of each projection (the first dimension of the sinogram) equal to `min(image.shape)`. The default behavior (None) is equivalent to False.

Returns: **radon_image** : ndarray

Radon transform (sinogram). The tomography rotation axis will lie at the pixel index `radon_image.shape[0] // 2` along the 0th dimension of `radon_image`.

Notes

Based on code of Justin K. Romberg (http://www.clear.rice.edu/elec431/projects96/DSP/bpanalysis.html)

References

[R489] AC Kak, M Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press 1988.

[R490] B.R. Ramesh, N. Srinivasa, K. Rajgopal, "An Algorithm for Computing the Discrete Radon Transform With Some Applications", Proceedings of the Fourth IEEE Region 10 International Conference, TENCON '89, 1989

rescale

`skimage.transform.` **`rescale`**(*`image`*, *`scale`*, *`order=1`*, *`mode=None`*, *`cval=0`*, *`clip=True`*, *`preserve_range=False`*, *`multichannel=None`*) [source]

Scale image by a certain factor.

Performs interpolation to upscale or down-scale images. For down-sampling N-dimensional images with integer factors by applying a function or the arithmetic mean, see skimage.measure.block_reduce and skimage.transform.downscale_local_mean, respectively.

Parameters: **image** : ndarray

Input image.

scale : {float, tuple of floats}

Scale factors. Separate scale factors can be defined as (rows, cols[, ...][, dim]).

Returns: **scaled** : ndarray

Scaled version of the input.

Other Parameters:

order : int, optional

The order of the spline interpolation, default is 1. The order has to be in the range 0-5. See skimage.transform.warp for detail.

mode : {'constant', 'edge', 'symmetric', 'reflect', 'wrap'}, optional

Points outside the boundaries of the input are filled according to the given mode. Modes match the behaviour of numpy.pad. The default mode is 'constant'.

cval : float, optional

Used in conjunction with mode 'constant', the value outside the image boundaries.

clip : bool, optional

Whether to clip the output to the range of values of the input image. This is enabled by default, since higher order interpolation may produce values outside the given input range.

preserve_range : bool, optional

Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of img_as_float.

multichannel : bool, optional

Whether the last axis of the image is to be interpreted as multiple channels or another spatial dimension. By default, is set to True for 3D (2D+color) inputs, and

False for others. Starting in release 0.16, this will always default to False.

Examples

```
>>> from skimage import data
>>> from skimage.transform import rescale
>>> image = data.camera()
>>> rescale(image, 0.1, mode='reflect').shape
(51, 51)
>>> rescale(image, 0.5, mode='reflect').shape
(256, 256)
```

resize

skimage.transform.**resize**(*image*, *output_shape*, *order=1*, *mode=None*, *cval=0*, *clip=True*, *preserve_range=False*) [source]

Resize image to match a certain size.

Performs interpolation to up-size or down-size images. For down-sampling N-dimensional images by applying a function or the arithmetic mean, see skimage.measure.block_reduce and skimage.transform.downscale_local_mean, respectively.

Parameters:

image : ndarray

Input image.

output_shape : tuple or ndarray

Size of the generated output image (rows, cols[, ...][, dim]). If dim is not provided, the number of channels is preserved. In case the number of input channels does not equal the number of output channels a n-dimensional interpolation is applied.

Returns:

resized : ndarray

Resized version of the input.

Other Parameters:

order : int, optional

The order of the spline interpolation, default is 1. The order has to be in the range 0-5. See skimage.transform.warp for detail.

mode : {'constant', 'edge', 'symmetric', 'reflect', 'wrap'}, optional

Points outside the boundaries of the input are filled according to the given mode. Modes match the behaviour of numpy.pad. The default mode is 'constant'.

cval : float, optional

Used in conjunction with mode 'constant', the value outside the image boundaries.

clip : bool, optional

Whether to clip the output to the range of values of the input image. This is enabled by default, since higher order interpolation may produce values outside the given input range.

preserve_range : bool, optional

Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of img_as_float.

Notes

Modes 'reflect' and 'symmetric' are similar, but differ in whether the edge pixels are duplicated during the reflection. As an example, if an array has values [0, 1, 2] and was padded to the right by four values using symmetric, the result would be [0, 1, 2, 2, 1, 0, 0], while for reflect it would be [0, 1, 2, 1, 0, 1, 2].

Examples

```
>>> from skimage import data
>>> from skimage.transform import resize
```

```
>>> image = data.camera()
>>> resize(image, (100, 100), mode='reflect').shape
(100, 100)
```

rotate

skimage.transform.**rotate**(*image*, *angle*, *resize=False*, *center=None*, *order=1*, *mode='constant'*, *cval=0*, *clip=True*, *preserve_range=False*) [source]

Rotate image by a certain angle around its center.

Parameters:

image : ndarray
Input image.

angle : float
Rotation angle in degrees in counter-clockwise direction.

resize : bool, optional
Determine whether the shape of the output image will be automatically calculated, so the complete rotated image exactly fits. Default is False.

center : iterable of length 2
The rotation center. If `center=None`, the image is rotated around its center, i.e. `center=(rows / 2 - 0.5, cols / 2 - 0.5)`.

Returns:

rotated : ndarray
Rotated version of the input.

Other Parameters:

order : int, optional
The order of the spline interpolation, default is 1. The order has to be in the range 0-5. See skimage.transform.warp for detail.

mode : {'constant', 'edge', 'symmetric', 'reflect', 'wrap'}, optional
Points outside the boundaries of the input are filled according to the given mode. Modes match the behaviour of numpy.pad.

cval : float, optional
Used in conjunction with mode 'constant', the value outside the image boundaries.

clip : bool, optional
Whether to clip the output to the range of values of the input image. This is enabled by default, since higher order interpolation may produce values outside the given input range.

preserve_range : bool, optional
Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of img_as_float.

Examples

```
>>> from skimage import data
>>> from skimage.transform import rotate
>>> image = data.camera()
>>> rotate(image, 2).shape
(512, 512)
>>> rotate(image, 2, resize=True).shape
(530, 530)
>>> rotate(image, 90, resize=True).shape
(512, 512)
```

seam_carve

skimage.transform.**seam_carve**(*image*, *energy_map*, *mode*, *num*, *border=1*, *force_copy=True*) [source]

Carve vertical or horizontal seams off an image.

Carves out vertical/horizontal seams from an image while using the given energy map to decide the importance of each pixel.

Parameters:

image : (M, N) or (M, N, 3) ndarray
Input image whose seams are to be removed.

energy_map : (M, N) ndarray
The array to decide the importance of each pixel. The higher the value corresponding to a pixel, the more the algorithm will try to keep it in the image.

mode : str {'horizontal', 'vertical'}
Indicates whether seams are to be removed vertically or horizontally. Removing seams horizontally will decrease the height whereas removing vertically will decrease the width.

num : int
Number of seams are to be removed.

border : int, optional
The number of pixels in the right, left and bottom end of the image to be excluded from being considered for a seam. This is important as certain filters just ignore image boundaries and set them to 0. By default border is set to 1.

force_copy : bool, optional
If set, the image and energy_map are copied before being used by the method which modifies it in place. Set this to False if the original image and the energy map are no longer needed after this opetration.

Returns:

out : ndarray
The cropped image with the seams removed.

References

[R491] Shai Avidan and Ariel Shamir "Seam Carving for Content-Aware Image Resizing" http://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Avidan07.pdf

swirl

`skimage.transform.` **`swirl`**(*image*, *center=None*, *strength=1*, *radius=100*, *rotation=0*, *output_shape=None*, *order=1*, *mode=None*, *cval=0*, *clip=True*, *preserve_range=False*) [source]

Perform a swirl transformation.

Parameters:

image : ndarray
Input image.

center : (row, column) tuple or (2,) ndarray, optional
Center coordinate of transformation.

strength : float, optional
The amount of swirling applied.

radius : float, optional
The extent of the swirl in pixels. The effect dies out rapidly beyond radius.

rotation : float, optional
Additional rotation applied to the image.

Returns:

swirled : ndarray
Swirled version of the input.

Other Parameters:

output_shape : tuple (rows, cols), optional
Shape of the output image generated. By default the shape of the input image is preserved.

order : int, optional

The order of the spline interpolation, default is 1. The order has to be in the range 0-5. See skimage.transform.warp for detail.

mode : {'constant', 'edge', 'symmetric', 'reflect', 'wrap'}, optional

Points outside the boundaries of the input are filled according to the given mode, with 'constant' used as the default. Modes match the behaviour of numpy.pad.

cval : float, optional

Used in conjunction with mode 'constant', the value outside the image boundaries.

clip : bool, optional

Whether to clip the output to the range of values of the input image. This is enabled by default, since higher order interpolation may produce values outside the given input range.

preserve_range : bool, optional

Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of img_as_float.

warp

`skimage.transform.` **`warp`**(*image*, *inverse_map*, *map_args={}*, *output_shape=None*, *order=1*, *mode='constant'*, *cval=0.0*, *clip=True*, *preserve_range=False*) [source]

Warp an image according to a given coordinate transformation.

Parameters:

image : ndarray

Input image.

inverse_map : transformation object, callable `cr = f(cr, **kwargs)`, or ndarray

Inverse coordinate map, which transforms coordinates in the output images into their corresponding coordinates in the input image.

There are a number of different options to define this map, depending on the dimensionality of the input image. A 2-D image can have 2 dimensions for gray-scale images, or 3 dimensions with color information.

- For 2-D images, you can directly pass a transformation object, e.g. skimage.transform.SimilarityTransform, or its inverse.
- For 2-D images, you can pass a `(3, 3)` homogeneous transformation matrix, e.g. skimage.transform.SimilarityTransform.params.
- For 2-D images, a function that transforms a `(M, 2)` array of `(col, row)` coordinates in the output image to their corresponding coordinates in the input image. Extra parameters to the function can be specified through map_args.
- For N-D images, you can directly pass an array of coordinates. The first dimension specifies the coordinates in the input image, while the subsequent dimensions determine the position in the output image. E.g. in case of 2-D images, you need to pass an array of shape `(2, rows, cols)`, where rows and cols determine the shape of the output image, and the first dimension contains the `(row, col)` coordinate in the input image. See scipy.ndimage.map_coordinates for further documentation.

Note, that a `(3, 3)` matrix is interpreted as a homogeneous transformation matrix, so you cannot interpolate values from a 3-D input, if the output is of shape `(3,)`.

See example section for usage.

map_args : dict, optional

Keyword arguments passed to inverse_map.

output_shape : tuple (rows, cols), optional

Shape of the output image generated. By default the shape of the input image is

preserved. Note that, even for multi-band images, only rows and columns need to be specified.

order : int, optional

The order of interpolation. The order has to be in the range 0-5:

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

mode : {'constant', 'edge', 'symmetric', 'reflect', 'wrap'}, optional

Points outside the boundaries of the input are filled according to the given mode. Modes match the behaviour of numpy.pad.

cval : float, optional

Used in conjunction with mode 'constant', the value outside the image boundaries.

clip : bool, optional

Whether to clip the output to the range of values of the input image. This is enabled by default, since higher order interpolation may produce values outside the given input range.

preserve_range : bool, optional

Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of img_as_float.

Returns: **warped** : double ndarray

The warped input image.

Notes

- The input image is converted to a double image.
- In case of a SimilarityTransform, AffineTransform and ProjectiveTransform and order in [0, 3] this function uses the underlying transformation matrix to warp the image with a much faster routine.

Examples

```
>>> from skimage.transform import warp
>>> from skimage import data
>>> image = data.camera()
```

The following image warps are all equal but differ substantially in execution time. The image is shifted to the bottom.

Use a geometric transform to warp an image (fast):

```
>>> from skimage.transform import SimilarityTransform
>>> tform = SimilarityTransform(translation=(0, -10))
>>> warped = warp(image, tform)
```

Use a callable (slow):

```
>>> def shift_down(xy):
...     xy[:, 1] -= 10
...     return xy
>>> warped = warp(image, shift_down)
```

Use a transformation matrix to warp an image (fast):

```
>>> matrix = np.array([[1, 0, 0], [0, 1, -10], [0, 0, 1]])
>>> warped = warp(image, matrix)
>>> from skimage.transform import ProjectiveTransform
>>> warped = warp(image, ProjectiveTransform(matrix=matrix))
```

You can also use the inverse of a geometric transformation (fast):

```
>>> warped = warp(image, tform.inverse)
```

For N-D images you can pass a coordinate array, that specifies the coordinates in the input image for every element in the output image. E.g. if you want to rescale a 3-D cube, you can do:

```
>>> cube_shape = np.array([30, 30, 30])
>>> cube = np.random.rand(*cube_shape)
```

Setup the coordinate array, that defines the scaling:

```
>>> scale = 0.1
>>> output_shape = (scale * cube_shape).astype(int)
>>> coords0, coords1, coords2 = np.mgrid[:output_shape[0],
...                    :output_shape[1], :output_shape[2]]
>>> coords = np.array([coords0, coords1, coords2])
```

Assume that the cube contains spatial data, where the first array element center is at coordinate (0.5, 0.5, 0.5) in real space, i.e. we have to account for this extra offset when scaling the image:

```
>>> coords = (coords + 0.5) / scale - 0.5
>>> warped = warp(cube, coords)
```

warp_coords

skimage.transform.**warp_coords**(*coord_map*, *shape*, *dtype=<class 'numpy.float64'>*) [source]

Build the source coordinates for the output of a 2-D image warp.

Parameters:

coord_map : callable like GeometricTransform.inverse

Return input coordinates for given output coordinates. Coordinates are in the shape (P, 2), where P is the number of coordinates and each element is a `(row, col)` pair.

shape : tuple

Shape of output image `(rows, cols[, bands])`.

dtype : np.dtype or string

dtype for return value (sane choices: float32 or float64).

Returns:

coords : (ndim, rows, cols[, bands]) array of dtype dtype

Coordinates for scipy.ndimage.map_coordinates, that will yield an image of shape (orows, ocols, bands) by drawing from source points according to the coord_transform_fn.

Notes

This is a lower-level routine that produces the source coordinates for 2-D images used by warp().

It is provided separately from warp to give additional flexibility to users who would like, for example, to re-use a particular coordinate mapping, to use specific dtypes at various points along the the image-warping process, or to implement different post-processing logic than warp performs after the call to ndi.map_coordinates.

Examples

Produce a coordinate map that shifts an image up and to the right:

```
>>> from skimage import data
>>> from scipy.ndimage import map_coordinates
>>>
>>> def shift_up10_left20(xy):
...     return xy - np.array([-20, 10])[None, :]
```

```
>>>
>>> image = data.astronaut().astype(np.float32)
>>> coords = warp_coords(shift_up10_left20, image.shape)
>>> warped_image = map_coordinates(image, coords)
```

AffineTransform

---

class skimage.transform. **AffineTransform** (*matrix=None*, *scale=None*, *rotation=None*, *shear=None*, *translation=None*) [source]

Bases: skimage.transform._geometric.ProjectiveTransform

2D affine transformation of the form:

> **X = a0*x + a1*y + a2 =**
>
> = sx*x*cos(rotation) - sy*y*sin(rotation + shear) + a2
>
> **Y = b0*x + b1*y + b2 =**
>
> = sx*x*sin(rotation) + sy*y*cos(rotation + shear) + b2

where sx and sy are scale factors in the x and y directions, and the homogeneous transformation matrix is:

```
[[a0  a1  a2]
 [b0  b1  b2]
 [0   0    1]]
```

Parameters:

matrix : (3, 3) array, optional
Homogeneous transformation matrix.

scale : (sx, sy) as array, list or tuple, optional
Scale factors.

rotation : float, optional
Rotation angle in counter-clockwise direction as radians.

shear : float, optional
Shear angle in counter-clockwise direction as radians.

translation : (tx, ty) as array, list or tuple, optional
Translation parameters.

Attributes

params	((3, 3) array) Homogeneous transformation matrix.

__init__ (*matrix=None*, *scale=None*, *rotation=None*, *shear=None*, *translation=None*) [source]

---

rotation

---

scale

---

shear

---

translation

EssentialMatrixTransform

class skimage.transform. **EssentialMatrixTransform**(*rotation=None*, *translation=None*, *matrix=None*) [source]

Bases: skimage.transform._geometric.FundamentalMatrixTransform

Essential matrix transformation.

The essential matrix relates corresponding points between a pair of calibrated images. The matrix transforms normalized, homogeneous image points in one image to epipolar lines in the other image.

The essential matrix is only defined for a pair of moving images capturing a non-planar scene. In the case of pure rotation or planar scenes, the homography describes the geometric relation between two images (ProjectiveTransform). If the intrinsic calibration of the images is unknown, the fundamental matrix describes the projective relation between the two images (FundamentalMatrixTransform).

Parameters:

rotation : (3, 3) array, optional
Rotation matrix of the relative camera motion.

translation : (3, 1) array, optional
Translation vector of the relative camera motion. The vector must have unit length.

matrix : (3, 3) array, optional
Essential matrix.

References

[R492] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.

Attributes

params	((3, 3) array) Essential matrix.

__init__(*rotation=None*, *translation=None*, *matrix=None*) [source]

estimate(*src*, *dst*) [source]

Estimate essential matrix using 8-point algorithm.

The 8-point algorithm requires at least 8 corresponding point pairs for a well-conditioned solution, otherwise the over-determined solution is estimated.

Parameters:

src : (N, 2) array
Source coordinates.

dst : (N, 2) array
Destination coordinates.

Returns:

success : bool
True, if model estimation succeeds.

EuclideanTransform

class skimage.transform. **EuclideanTransform**(*matrix=None*, *rotation=None*, *translation=None*) [source]

Bases: skimage.transform._geometric.ProjectiveTransform

2D Euclidean transformation of the form:

> **X = a0 * x - b0 * y + a1 =**
>
> = x * cos(rotation) - y * sin(rotation) + a1
>
> **Y = b0 * x + a0 * y + b1 =**

```
= x * sin(rotation) + y * cos(rotation) + b1
```

where the homogeneous transformation matrix is:

```
[[a0  b0  a1]
 [b0  a0  b1]
 [0   0    1]]
```

The Euclidean transformation is a rigid transformation with rotation and translation parameters. The similarity transformation extends the Euclidean transformation with a single scaling factor.

Parameters:

matrix : (3, 3) array, optional
Homogeneous transformation matrix.

rotation : float, optional
Rotation angle in counter-clockwise direction as radians.

translation : (tx, ty) as array, list or tuple, optional
x, y translation parameters.

Attributes

params	((3, 3) array) Homogeneous transformation matrix.

`__init__`(*matrix=None*, *rotation=None*, *translation=None*) [source]

`estimate`(*src*, *dst*) [source]

Estimate the transformation from a set of corresponding points.

You can determine the over-, well- and under-determined parameters with the total least-squares method.

Number of source and destination coordinates must match.

Parameters:

src : (N, 2) array
Source coordinates.

dst : (N, 2) array
Destination coordinates.

Returns:

success : bool
True, if model estimation succeeds.

`rotation`

`translation`

`FundamentalMatrixTransform`

class `skimage.transform.` `FundamentalMatrixTransform`(*matrix=None*) [source]

Bases: `skimage.transform._geometric.GeometricTransform`

Fundamental matrix transformation.

The fundamental matrix relates corresponding points between a pair of uncalibrated images. The matrix transforms homogeneous image points in one image to epipolar lines in the other image.

The fundamental matrix is only defined for a pair of moving images. In the case of pure rotation or planar scenes, the homography describes the geometric relation between two images (ProjectiveTransform). If the intrinsic calibration of the images is known, the essential matrix describes the metric relation between the two images (EssentialMatrixTransform).

Parameters: **matrix** : (3, 3) array, optional
Fundamental matrix.

References

[R493] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.

Attributes

params	((3, 3) array) Fundamental matrix.

`__init__`(*matrix=None*) [source]

`estimate`(*src*, *dst*) [source]

Estimate fundamental matrix using 8-point algorithm.

The 8-point algorithm requires at least 8 corresponding point pairs for a well-conditioned solution, otherwise the over-determined solution is estimated.

Parameters: **src** : (N, 2) array
Source coordinates.
dst : (N, 2) array
Destination coordinates.

Returns: **success** : bool
True, if model estimation succeeds.

`inverse`(*coords*) [source]

Apply inverse transformation.

Parameters: **coords** : (N, 2) array
Destination coordinates.

Returns: **coords** : (N, 3) array
Epipolar lines in the source image.

`residuals`(*src*, *dst*) [source]

Compute the Sampson distance.

The Sampson distance is the first approximation to the geometric error.

Parameters: **src** : (N, 2) array
Source coordinates.
dst : (N, 2) array
Destination coordinates.

Returns: **residuals** : (N,) array
Sampson distance.

PiecewiseAffineTransform

class skimage.transform. **PiecewiseAffineTransform** [source]

Bases: skimage.transform._geometric.GeometricTransform

2D piecewise affine transformation.

Control points are used to define the mapping. The transform is based on a Delaunay triangulation of the points to form a mesh. Each triangle is used to find a local affine transform.

Attributes

affines	(list of AffineTransform objects) Affine transformations for each triangle in the mesh.
inverse_affines	(list of AffineTransform objects) Inverse affine transformations for each triangle in the mesh.

__init__() [source]

estimate(*src*, *dst*) [source]

Estimate the transformation from a set of corresponding points.

Number of source and destination coordinates must match.

Parameters: **src** : (N, 2) array
Source coordinates.
dst : (N, 2) array
Destination coordinates.

Returns: **success** : bool
True, if model estimation succeeds.

inverse(*coords*) [source]

Apply inverse transformation.

Coordinates outside of the mesh will be set to - 1.

Parameters: **coords** : (N, 2) array
Source coordinates.

Returns: **coords** : (N, 2) array
Transformed coordinates.

PolynomialTransform

class skimage.transform. **PolynomialTransform**(*params=None*) [source]

Bases: skimage.transform._geometric.GeometricTransform

2D polynomial transformation of the form:

> X = sum[j=0:order](sum[i=0:j](a_ji * x**(j - i) * y**i)) Y = sum[j=0:order](sum[i=0:j](b_ji * x**(j - i) * y**i))

Parameters: **params** : (2, N) array, optional
Polynomial coefficients where N * 2 = (order + 1) * (order + 2). So, a_ji is defined in

params[0, :] and b_ji in params[1, :].

Attributes

params	((2, N) array) Polynomial coefficients where N * 2 = (order + 1) * (order + 2). So, a_ji is defined in params[0, :] and b_ji in params[1, :].

`__init__`(*params=None*) [source]

`estimate`(*src*, *dst*, *order=2*) [source]

Estimate the transformation from a set of corresponding points.

You can determine the over-, well- and under-determined parameters with the total least-squares method.

Number of source and destination coordinates must match.

The transformation is defined as:

```
X = sum[j=0:order]( sum[i=0:j]( a_ji * x**(j - i) * y**i ))
Y = sum[j=0:order]( sum[i=0:j]( b_ji * x**(j - i) * y**i ))
```

These equations can be transformed to the following form:

```
0 = sum[j=0:order]( sum[i=0:j]( a_ji * x**(j - i) * y**i )) - X
0 = sum[j=0:order]( sum[i=0:j]( b_ji * x**(j - i) * y**i )) - Y
```

which exist for each set of corresponding points, so we have a set of N * 2 equations. The coefficients appear linearly so we can write A x = 0, where:

```
A   = [[1 x y x**2 x*y y**2 ... 0 ...             0 -X]
       [0 ...                 0 1 x y x**2 x*y y**2 -Y]
        ...
        ...
      ]
x.T = [a00 a10 a11 a20 a21 a22 ... ann
       b00 b10 b11 b20 b21 b22 ... bnn c3]
```

In case of total least-squares the solution of this homogeneous system of equations is the right singular vector of A which corresponds to the smallest singular value normed by the coefficient c3.

Parameters:

src : (N, 2) array
Source coordinates.

dst : (N, 2) array
Destination coordinates.

order : int, optional
Polynomial order (number of coefficients is order + 1).

Returns:

success : bool
True, if model estimation succeeds.

`inverse`(*coords*) [source]

ProjectiveTransform

class skimage.transform.`ProjectiveTransform`(*matrix=None*) [source]

Bases: skimage.transform._geometric.GeometricTransform

Projective transformation.

Apply a projective transformation (homography) on coordinates.

For each homogeneous coordinate $\mathbf{x} = [x, y, 1]^T$, its target position is calculated by multiplying with the given matrix, H, to give $H\mathbf{x}$:

```
[[a0 a1 a2]
 [b0 b1 b2]
 [c0 c1 1 ]].
```

E.g., to rotate by theta degrees clockwise, the matrix should be:

```
[[cos(theta) -sin(theta) 0]
 [sin(theta)  cos(theta) 0]
 [0            0         1]]
```

or, to translate x by 10 and y by 20:

```
[[1 0 10]
 [0 1 20]
 [0 0 1 ]].
```

Parameters: **matrix** : (3, 3) array, optional

Homogeneous transformation matrix.

Attributes

params	((3, 3) array) Homogeneous transformation matrix.

`__init__`(*matrix=None*) [source]

`estimate`(*src*, *dst*) [source]

Estimate the transformation from a set of corresponding points.

You can determine the over-, well- and under-determined parameters with the total least-squares method.

Number of source and destination coordinates must match.

The transformation is defined as:

```
X = (a0*x + a1*y + a2) / (c0*x + c1*y + 1)
Y = (b0*x + b1*y + b2) / (c0*x + c1*y + 1)
```

These equations can be transformed to the following form:

```
0 = a0*x + a1*y + a2 - c0*x*X - c1*y*X - X
0 = b0*x + b1*y + b2 - c0*x*Y - c1*y*Y - Y
```

which exist for each set of corresponding points, so we have a set of N * 2 equations. The coefficients appear linearly so we can write A x = 0, where:

```
A   = [[x y 1 0 0 0 -x*X -y*X -X]
       [0 0 0 x y 1 -x*Y -y*Y -Y]
        ...
        ...
      ]
x.T = [a0 a1 a2 b0 b1 b2 c0 c1 c3]
```

In case of total least-squares the solution of this homogeneous system of equations is the right singular vector of A which corresponds to the smallest singular value normed by the coefficient c3.

In case of the affine transformation the coefficients c0 and c1 are 0. Thus the system of equations is:

```
A   = [[x y 1 0 0 0 -X]
       [0 0 0 x y 1 -Y]
        ...
        ...
      ]
x.T = [a0 a1 a2 b0 b1 b2 c3]
```

Parameters:

src : (N, 2) array
Source coordinates.

dst : (N, 2) array
Destination coordinates.

Returns:

success : bool
True, if model estimation succeeds.

`inverse`(*coords*) [source]

Apply inverse transformation.

Parameters:

coords : (N, 2) array
Destination coordinates.

Returns:

coords : (N, 2) array
Source coordinates.

SimilarityTransform

---

class skimage.transform.`SimilarityTransform`(*matrix=None*, *scale=None*, *rotation=None*, *translation=None*) [source]

Bases: skimage.transform._geometric.EuclideanTransform

2D similarity transformation of the form:

> **X = a0 * x - b0 * y + a1 =**
>
> = s * x * cos(rotation) - s * y * sin(rotation) + a1
>
> **Y = b0 * x + a0 * y + b1 =**
>
> = s * x * sin(rotation) + s * y * cos(rotation) + b1

where `s` is a scale factor and the homogeneous transformation matrix is:

```
[[a0  b0  a1]
 [b0  a0  b1]
 [0   0    1]]
```

The similarity transformation extends the Euclidean transformation with a single scaling factor in addition to the rotation and translation parameters.

Parameters:

matrix : (3, 3) array, optional
Homogeneous transformation matrix.

scale : float, optional
Scale factor.

rotation : float, optional
Rotation angle in counter-clockwise direction as radians.

translation : (tx, ty) as array, list or tuple, optional
x, y translation parameters.

Attributes

params	((3, 3) array) Homogeneous transformation matrix.

`__init__`(*matrix=None*, *scale=None*, *rotation=None*, *translation=None*) [source]

`estimate`(*src*, *dst*) [source]

Estimate the transformation from a set of corresponding points.

You can determine the over-, well- and under-determined parameters with the total least-squares method.

Number of source and destination coordinates must match.

Parameters:

src : (N, 2) array
Source coordinates.

dst : (N, 2) array
Destination coordinates.

Returns:

success : bool
True, if model estimation succeeds.

`scale`